EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno
(Guest Editors)

Volume 27 (2008), Number 2

GPU-based Fast Ray Casting for a Large Number of Metaballs

Yoshihiro Kanamori, Zoltan Szego and Tomoyuki Nishita

The University of Tokyo, Japan
{pierrot,szegoz,nis}@nis-lab.is.s.u-tokyo.ac.jp

Abstract

Metaballs are implicit surfaces widely used to model curved objects, represented by the isosurface of a density field defined by a set of points. Recently, the results of particle-based simulations have been often visualized using a large number of metaballs, however, such visualizations have high rendering costs. In this paper we propose a fast technique for rendering metaballs on the GPU. Instead of using polygonization, the isosurface is directly evaluated in a per-pixel manner. For such evaluation, all metaballs contributing to the isosurface need to be extracted along each viewing ray, on the limited memory of GPUs. We handle this by keeping a list of metaballs contributing to the isosurface and efficiently update it. Our method neither requires expensive precomputation nor acceleration data structures often used in existing ray tracing techniques. With several optimizations, we can display a large number of moving metaballs quickly.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Animation I.3.3 [Computer Graphics]: Display Algorithm

1. Introduction

The use of implicit surfaces, such as *metaballs* [NHK*85] (also referred to as *blobs* [Bli82] or *soft objects* [WMW86]), to represent smooth objects has been widespread in computer graphics. In recent years, metaballs have been widely used to visualize the results of particle-based simulations. Since such simulations use thousands or tens of thousands of particles, a fast visualization method that can handle a large number of metaballs at high quality is desirable.

Each metaball has a density function, and a set of metaballs represents a smooth surface as the isosurface of the density field. The isosurface is mainly visualized by polygonization [LC87, Ura06] or ray casting [WT90, NN94].

The visual quality of the surface created by polygonization depends on the resolution of the grid, forming a trade-off in relation to computational cost; with a low-resolution grid, polygonization performs quite fast, however, causes artifacts on the generated surface. Even worse, polygonization can completely miss objects smaller than the grid resolution, such as splashes of fluids. Conversely, with a high-resolution grid, polygonization can keep the fine details and generate high quality surfaces, however, it incurs quite high computational and memory costs, and is not suited for interactive applications.

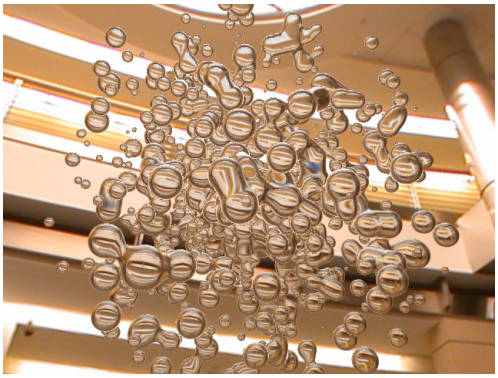


Figure 1: *Screenshot from an animation with 1,000 moving metaballs, rendered in 640×480 at 23.1 fps. Unlike polygonization, even small particles are rendered smoothly.*

On the other hand, in the case of ray casting, the computational cost depends primarily on the screen resolution. Ray casting can produce high quality smooth surfaces even for small objects, however, generally causes an extremely high computational cost. In order to perform a ray-isosurface intersection test, we need to extract the metaballs that contribute to the isosurface along the viewing ray, then construct and solve an equation for the intersection test.

In this paper, we propose a fast ray casting method to render metaballs on the GPU. The basic workflow of our method is simple: First we obtain the intersection points of each viewing ray and the spheres representing the effective ranges of metaballs using *depth peeling* [Mam89, Eve02]. Next, after each pass of depth peeling, we perform a ray-isosurface intersection test using *Bézier Clipping* [NSK90]. When performing this intersection test on the GPU, we need to extract the metaballs contributing to the isosurface along each viewing ray, on the limited amount of memory available. We propose an efficient algorithm to solve this problem. By optimizing the major parts of our rendering process, depth peeling and the isosurface test, our method can display a large number of moving metaballs quickly (see Figure 1). We demonstrate that our method is useful for previewing the results of particle-based simulations such as fluid dynamics and molecular dynamics.

2. Related Work

In this section, we first briefly introduce related work on the density functions of metaballs, then mention recent techniques related to rendering smooth surfaces on the GPU.

2.1. Density Functions of Metaballs

The density function f_i for metaball i is a function of r, the distance from that metaball's center $\mathbf{p}_i \in \mathbb{R}^3$, and monotonically decreases as r increases. If f_i has a finite support R_i, $f_i(r) = 0$ in the range of $r \geq R_i$. R_i represents the effective radius of metaball i. Throughout this paper, we refer to the sphere centered at $\mathbf{p}_i$ with radius R_i as metaball i's "**effective sphere**" (illustrated with solid circles in Figure 2). For N metaballs, the shape of the curved surface is defined by the points $\mathbf{x} \in \mathbb{R}^3$ satisfying the following equation:

$$f(\mathbf{x}) = \sum_{i=0}^{N-1} q_i \, f_i(\|\mathbf{x} - \mathbf{p}_i\|) - T = 0, \qquad (1)$$

where T is a threshold, and $\{q_i\}$ are the density coefficients. The normal vector at $\mathbf{x}$ can be derived from $-\nabla f(\mathbf{x})$.

The first particle-based implicit surface proposed in computer graphics was Blinn's *blob* [Bli82]. Blinn defined the density function as a Gaussian function, however, the calculation of the isosurface becomes expensive since a Gaussian function has an infinite support and thus every blob has to be taken into account. This is why several functions with finite supports have been proposed, such as piecewise quadratic [NHK*85], quartic [MI87] and degree-six polynomials [WMW86]. While these polynomials produce smoother results for higher degrees, the computational cost of solving the equations for isosurface tests also increases with the degree. Nishita and Nakamae [NN94] used Bézier Clipping to solve equations with the quartic and degree-six polynomials quickly. Sherstyuk [She99] used piecewise cubic Hermite polynomials to approximate arbitrary density functions. Our method deals with the above-mentioned quartic and degree-six polynomials, and solves the equations accurately without any approximations.

2.2. GPU-based Surface Rendering

There are several GPU-based ray casting methods, such as NURBS [PSS*06], subdivision surfaces [YK04], *SLIM* surfaces [KOKK06, SGS06] and *point set surfaces* [GG07].

Loop and Blinn [LB06] proposed a GPU-based ray casting method to render algebraic surfaces defined by Bézier tetrahedrons with up to quartic polynomials. For ray-isosurface intersection tests, the coefficients of the equations are efficiently computed by linearly interpolating the vertex attributes of each tetrahedron. Consequently, their method can render just two metaballs much faster than ours. However, the computation of vertex attributes must take the neighboring metaballs into account, which can be a considerable performance hit for a large number of moving metaballs. Our method can efficiently handle such scenes, as well as equations with a degree higher than four.

Iwasaki *et al.* [IYDN06] used metaballs for visualizing the results of a particle-based fluid simulation. They divided the space into a grid, and accelerated the evaluation of the density function at each grid point using the GPU. Similarly to polygonization, however, their method can miss small splashes. Recently, van Kooten *et al.* [vKvdBT07] visualized metaballs on the GPU, using on-surface particles distributed by repulsive forces. Their method can exploit the temporal coherence of animations, however, as they reported, it suffers from small objects due to the limited resolution of the buffer for repulsion computations. Since our method is based on ray casting, it can display small objects.

3. Basic Framework for Rendering Metaballs

Since our method is based on the ray-isosurface intersection test as proposed by Nishita and Nakamae [NN94], we first describe the basic process of their rendering algorithm, then introduce Bézier Clipping, which we use as a solver.

3.1. Nishita and Nakamae's Algorithm

In Nishita and Nakamae's algorithm, the metaballs' density function is defined to have a finite support. Since the equations for the isosurface tests change for each interval defined

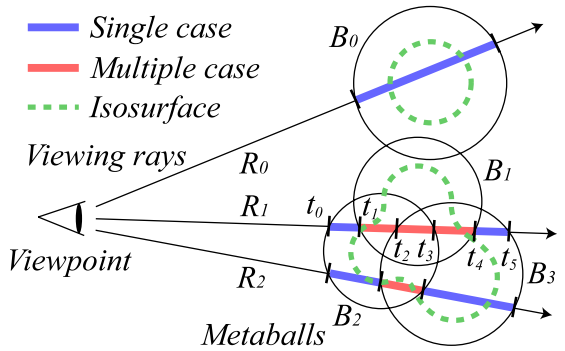


Figure 2: *The basic framework for rendering metaballs. $\{B_0,\ldots,B_3\}$ are the metaballs' IDs, and $\{t_0,\ldots,t_5\}$ the parameters for each intersection point along the ray R_1.*

by two consecutive intersections of the viewing ray and the effective spheres, a ray-isosurface intersection test is performed for each interval. First, their algorithm calculates the intersections of the viewing ray with every effective sphere, and sorts these intersections by increasing distance from the viewpoint. Then, for each interval on the viewing ray, their algorithm proceeds to find the isosurface depending on the number of effective spheres intersected in the interval (Figure 2):

Single Case (only one effective sphere intersected): The isosurface is spherical, therefore just a ray-sphere intersection test is performed. The radius of the sphere can be precomputed for fast runtime evaluation.

Multiple Case (two or more effective spheres intersected): After extracting the metaballs intersected by the viewing ray, an equation in Bézier form (see Appendix A) is constructed, and solved using Bézier Clipping.

Their algorithm performed the above processing scanline by scanline, sequentially for each pixel. Such processing is a good candidate for parallelization, thus we can expect speedups by moving it to the GPU. One challenge of using the GPU is that we have to extract metaballs necessary for isosurface tests with limited memory. For example, along the ray R_1 in Figure 2, in the interval $[t_0,t_1]$ we need only B_2 for the isosurface test, while in $[t_2,t_3]$, we need B_1, B_2 and B_3. We propose an efficient algorithm to do this extraction, enabling ray casting to take advantage of the parallel processing power of GPUs.

3.2. Bézier Clipping

Bézier Clipping was introduced by Nishita *et al.* [NSK90] as a fast method to compute the intersection between the viewing ray and a Bézier surface. See Figure 3 for a simple 1D explanation. A degree-M Bézier curve is denoted with a parameter $u \in [0,1]$ as follows:

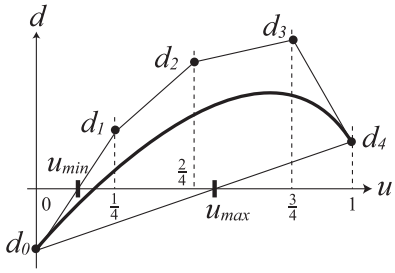


Figure 3: *One-dimensional Bézier Clipping.*

$$\mathbf{D}(u) = (u, d(u)) = \sum_{i=0}^{M} \mathbf{D}_i B_i^M(u), \tag{2}$$

where B represents the Bernstein polynomial, $B_i^M(u) = \binom{M}{i} u^i (1-u)^{M-i}$, $\mathbf{D}_i$ the control points, $\mathbf{D}_i = (i/M, d_i)$. The Bézier curve $\mathbf{D}(u)$ is always inside the convex hull formed by its control points $\mathbf{D}_i$. The intersection point of the curve $\mathbf{D}(u)$ and the u-axis is between the two points where the convex hull intersects the axis, u_{min} and u_{max}. Therefore, by iteratively clipping the Bézier curve in the range $[u_{min}, u_{max}]$ using de Casteljau's algorithm, it will converge to the root of $d(u) = 0$. When the range does not converge past a certain threshold, the Bézier curve is split in two, and the process is repeated for both parts recursively. The above process, unlike Newton's method, does not require an initial value and converges very quickly.

Since any polynomial can be converted into Bézier form [Sch90], Bézier Clipping can be used as a general solver for polynomial equations. Campagna *et al.* [CSS97] pointed out a problem that roots are misdetected when the convex hull barely intersects the axis, and showed a heuristic to mitigate the problem by slightly increasing the clipping range. Pabst *et al.* [PSS*06] proposed an iterative algorithm for Bézier Clipping that works on the limited memory available on GPUs. We propose an even more robust way to solve the problem shown by Campagna *et al.*, as well as a GPU-based algorithm that runs faster than Pabst *et al.*'s method.

4. Algorithm

Our method only finds the isosurface closest to the viewpoint. Throughout this paper, t denotes a parameter of the viewing ray, specifying the distance from the viewpoint to an intersection.

Figure 4 illustrates the outline of our method. Unlike the calculation on the CPU as described in Section 3.1, it is difficult to store many ray-sphere intersections at each pixel in the limited memory available on the GPU. On the other hand, the isosurface can be found in the first few intervals at some pixels, thus storing all the intersections is not always

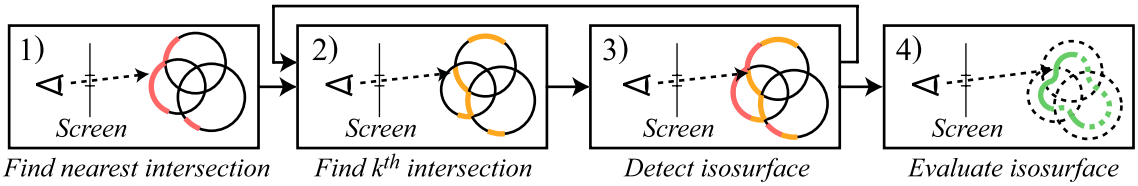


Figure 4: *Outline of our method. 1) Render effective spheres, find the nearest intersection point on the viewing ray. 2) Find the kth intersection using depth peeling (where k is the number of iterations). 3) In the interval between the $(k-1)$th and kth intersection points, perform a ray-isosurface test at each pixel using Bézier Clipping. Repeat steps 2 and 3 until the isosurface is found or no intersection is available. 4) Evaluate and shade the isosurface closest to the viewpoint at every pixel.*

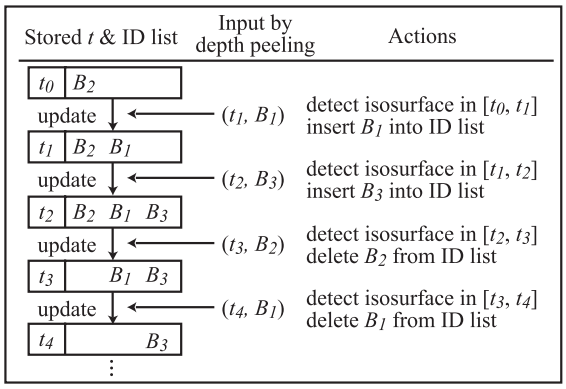


Figure 5: *Updating the list of metaball IDs for the viewing ray R_1 in Figure 2.*

```
// ListBuf_A, ListBuf_B :
//          two sets of textures containing t and the ID list
// TmpTex : a texture containing t and one ID
Initialize ListBuf_A, ListBuf_B and TmpTex with 0;
Render the front face of every effective sphere (step 1 in Figure 4);
Store the resulting first t and ID to ListBuf_A;
do
  Render the front and back faces
    of every effective sphere (step 2 in Figure 4);
  if (t is less than ListBuf_A's t)
    Discard the fragment;
  endif
  Store the resulting t and the ID to TmpTex;
  Detect the isosurface
    using ListBuf_A and TmpTex (step 3 in Figure 4);
  Update the ID list using ListBuf_A and TmpTex;
  Store the updated ID list to ListBuf_B;
  Swap ListBuf_A and ListBuf_B;
until (the isosurface is found or no t available);
Evaluate the isosurface using ListBuf_A (step 4 in Figure 4);
```

Figure 6: *Pseudocode for our rendering process.*

necessary. We keep as small amount of information as possible for ray-isosurface tests at each pixel.

In the rest of this section, we describe how the information for ray-isosurface tests is maintained and updated, and show the pseudocode of our algorithm. Finally, we discuss how to optimize depth peeling and isosurface tests.

4.1. Finding Ray-Sphere Intersections by Depth Peeling

While Nishita and Nakamae's algorithm computes ray-sphere intersections per pixel, we employ depth peeling [Mam89, Eve02] and obtain the intersections in parallel, from front to back. By drawing effective spheres, we store the IDs and ray parameters into a texture. At the same time, we look up another texture which keeps parameters at previous intersections, and discard fragments whose parameters are less than the previous ones. We repeat this procedure and then obtain metaball IDs and parameters at ray-sphere intersections.

4.2. Extracting Metaballs for Isosurface Test

In order to obtain the metaballs required for ray-isosurface tests as described in Section 3.1, we keep a list of metaballs and incrementally update it when a metaball ID is given by depth peeling. Our method to update the list is quite simple: since an effective sphere intersects the viewing ray twice, it only contributes to the isosurface between those two intersections, and thus the metaball in question can be added to the list at the first intersection point, and deleted at the second (see Figure 5). In case that the viewing ray touches a metaball, we can guarantee the ID to appear twice by looking for intersection points twice during depth peeling.

When working on the GPU, we must make do with a fixed sized buffer. Our method keeps this list in a set of textures, using the GPU's ability to render into N_{buf} buffers at once, where the maximum value of N_{buf} depends on the GPU used (4 for shader model 3.0 GPUs, and 8 for shader model 4.0). By grouping N_{buf} screen-sized floating point RGBA textures together and using each channel to store a value, we can treat them as one array that can store $4N_{buf}$ floating point values. In this array, we store the viewing ray parameter t, the number of IDs, and the IDs themselves in the remaining

$4N_{buf} - 2$ slots. In our experiments, with a value of $N_{buf} = 4$ we could therefore save at most 14 IDs.

4.3. Summary of Basic GPU-based Rendering

Using the process described above, we can now render metaballs; In each step of depth peeling, we retrieve a metaball ID and the ray-sphere intersection with each viewing ray from front to back. We then construct an equation in Bézier form (see Appendix A) for a ray-isosurface test, solve it using Bézier Clipping, and update the list required for the ray-isosurface tests. When the isosurface is found at a pixel, we terminate the process at that pixel. Figure 6 contains the pseudocode for the rendering process of our method as shown in Figure 4.

5. Optimizing Depth Peeling and the Isosurface Test

In this section, we will discuss the optimization of the parts that take up most of the rendering process, depth peeling and the isosurface test. Our method requires rendering the effective spheres multiple times. For a large number of metaballs, this computation becomes expensive. Therefore, we employ a fast sphere rendering method, and try to reduce the overall number of spheres to be drawn. Additionally, we reduce fragments to be processed in depth peeling by subdividing the screen into tiles and processing each tile separately. We also optimize Bézier Clipping, which is the most computationally expensive part of the isosurface tests.

5.1. Fast Rendering of Spheres

When rendering spheres in depth peeling, we can use polygons as a possible solution. However, since a fine tessellation is required to accurately find the intersection point on the viewing ray, this method is not so effective, especially when the spheres become small on the screen. In our method, we render spheres using only their radii and their centers. First, in the vertex shader, we specify the rectangular region that tightly fits the perspective-projected sphere on the screen (see Appendix B for details). Then for each fragment in this region, we perform an intersection test for the viewing ray and the sphere in the fragment shader, and discard the fragment if there is no intersection. Similar methods have been used to render perspective projected disks [BSK04] and quadratic curved surfaces [RE05, SWBG06]. Our method is specialized for rendering spheres, and runs much faster than when using polygons. Note that our method only renders either the front or the back faces of the sphere, however, when doing depth peeling there is no need to render both: just the one farther away from the previous intersection point is sufficient.

5.2. Metaball Culling via Occlusion Queries

Next, we will describe how to reduce the number of rendered metaballs. We refer to the sphere making up the isosurface

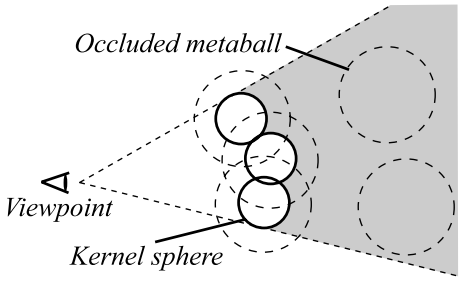


Figure 7: *Culling metaballs using kernel spheres. The metaballs in the gray region are hidden by kernel spheres.*

(as introduced in the Single Case of Section 3.1) as a "**kernel sphere**". We can perform culling based on the premise that metaballs hidden from the viewpoint by kernel spheres do not contribute to the closest isosurface (see Figure 7). To check for this, the GPU's *occlusion query* functionality can be used. First we render the kernel spheres and save the ray parameters to a texture. Next, we disable writes to the depth buffer, and render the effective spheres, using the occlusion query to determine which effective spheres were drawn. By culling the effective spheres that were not rendered in the above test, we can reduce the number of effective spheres to be rendered during depth peeling. This type of culling is especially effective when metaballs are in a dense group, and can reduce the number of metaballs to be rendered to 10-20% in some cases. Furthermore, by looking up the texture storing the ray parameters at the intersections with the kernel spheres, we can obtain the ray-isosurface intersections in the Single Case.

5.3. Optimization with Screen Tiling

While depth peeling computes intersections of viewing rays and spheres in parallel, the number of intersections to be computed varies by each viewing ray. As Bernadon *et al.* [BPCS06] did for volume rendering, we therefore subdivide the screen into uniform tiles and perform depth peeling for each tile until depth peeling terminates (which can be automatically detected using occlusion query [Eve02]). We calculate which spheres intersect with each tile, and register the intersecting spheres to be rendered for every tile. This computation is done on the CPU in our prototype system. Regarding the size of tiles, too small tiles cause many duplicate registrations of spheres among tiles, while too large tiles cannot handle different numbers of intersections efficiently. In all of our experiments, an 8×8 grid of tiles for 640×480 and a 16×16 grid for 1024×768 proved to be good choices.

5.4. Recursive Short-Stack Bézier Clipping

We also speed up Bézier Clipping. For GPU-based rendering of NURBS, Pabst *et al.* [PSS*06] used an iterative version of

Table 1: *Comparison of total rendering times (in milliseconds) between the CPU and GPU versions.*

resolution	metaballs	CPU	GPU	speedup
640x480	10,000	589.5	117.1	5.0
	50,000	1014.8	191.6	5.3
	100,000	1469.5	299.4	4.9
	200,000	2265.4	523.6	4.3
1024x768	10,000	1455.8	230.4	6.3
	50,000	2495.2	366.3	6.8
	100,000	3593.9	571.4	6.3
	200,000	5491.3	980.4	5.6

Bézier Clipping, which works with relatively small number of registers, at the cost of more iterations and slower convergence compared with the recursive algorithm by Nishita *et al.* [NSK90]. We use a recursive algorithm with a fixed size stack on the GPU. We try to keep the stack as small as possible, due to the fact that using too many registers reduces the parallel performance of recent GPUs [NVI]. Because of this, we only push Bézier control points onto the stack after making sure that their convex hull intersects the axis. In our algorithm, we first check how many times the Bézier control points' sign changes (see Figure 3), similar to the original version [NSK90]. If the sign does not change at all, the hull does not intersect the axis, thus we do not have to solve anything. If it changes more than once, there might be two or more solutions. In this case, since we expect the region not to get narrower later on during the clipping, we split the control polygon into two beforehand, while the original version does that only after the region is actually found not to get narrower. This modification adds almost no overhead, and robustly addresses the flaw pointed out by Campagna *et al.* [CSS97]; even when the convex hull barely intersects the axis, the control points' sign changes more than twice, thus the control polygon gets split in two and processed in more detail. Finally, only when the sign changes exactly once do we perform the relatively expensive intersection test between the convex hull and the axis. By using the above improvements, and setting the convergence threshold to $10e^{-4}$, a stack size of 3 proved to be sufficient. Pabst *et al.*'s method used 16.4 iterations on average and at most 29, while our method had an average of 2.3 iterations with a maximum of 8. We achieved an approximately double overall speedup of Bézier Clipping.

6. Results

Our implementation was written in C++, using OpenGL and Cg. We ran our tests on a PC with an Intel Core 2 Quad 2.66 GHz CPU and an NVIDIA GeForce 8800 Ultra graphics card. We used Wyvill *et al.*'s degree-six polynomial [WMW86] for metaballs in all of our experiments.

To evaluate the accuracy of our method, we used an example of randomly arranged metaballs with random effective radii (Figure 1), and compared the results calculated on the GPU to a reference image computed on the CPU. The relative RMS error of the ray parameters at ray-isosurface intersections was 0.7%. The error mainly results from a small number of dots at the edges of isosurfaces.

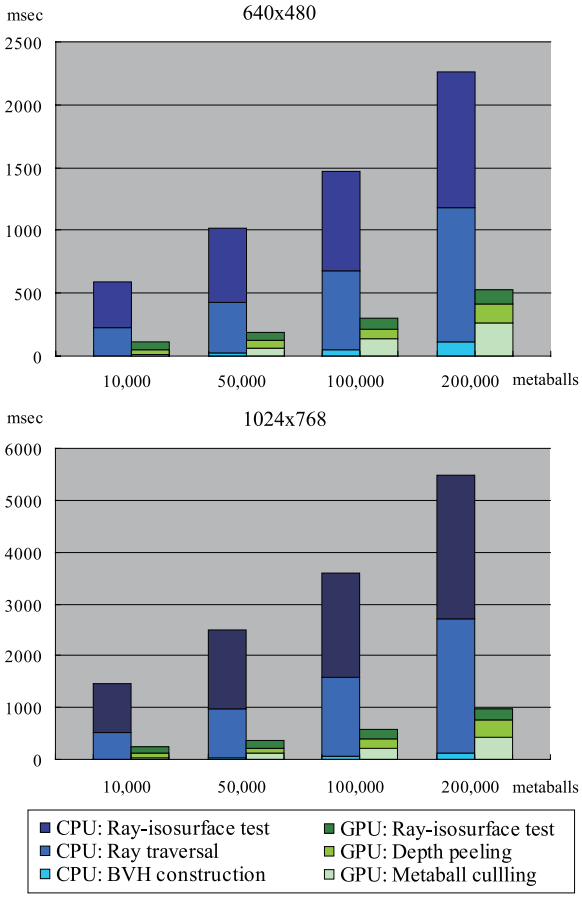
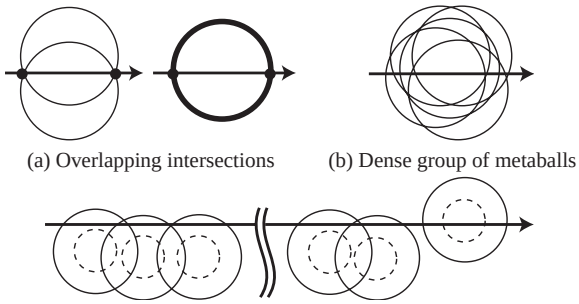


Figure 8: *Graphs of the computational times for the CPU and GPU versions when rendering the scene in Figure 1.*

While our GPU-based ray caster ran more than 75 times faster compared to a naive implemetation of Nishita and Nakamae [NN94] at 640×480 since they did not use any acceration data structure in their work, we further compared our method's performance with an optimized CPU-based ray caster. In this CPU version, Nishita and Nakamae's method [NN94] was used for isosurface extraction, while ray traversal was optimized using bounding volume hierarchies (BVH) built by the split-in-the-middle strategy (see [WMG*07] for details). Ray-sphere intersections were computed four at a time in parallel using SIMD, with one ray for each pixel, and to take advantage of multiple cores, this process was run in four threads. The Intel compiler was



(a) Overlapping intersections (b) Dense group of metaballs

(c) The viewing ray crossing too many metaballs

Figure 9: *Special cases.*

used with optimization options turned on. Packet-based ray traversal [WBS07] was not implemented. Table 1 and Figure 8 show the comparison results without shading calculations. Our GPU-based method had a 4-7x performance advantage compared to this CPU version, while further optimizations to the CPU algorithm might tighten this gap, especially for ray traversal which has been actively studied in the ray tracing community. However, optimizations for ray-isosurface tests, which account for about half the computational time as shown in Figure 8, are not trivial. Moreover, the GPU version still has the advantage of being much easier to integrate with existing polygonal scenes.

As far as we know, there has been no ray tracing method that aims to handle a large number of moving metaballs, thus it is hard to make appropriate comparisons to other existing methods. Loop and Blinn [LB06] could render two metaballs on a GeForce 7800 GTX in 640×480 at close to 300 fps. Our method reaches about 150 fps. Our method seems to be much slower considering the difference in the hardware's abilities, however, as we previously mentioned, their method has difficulties handling many moving metaballs efficiently. Knoll *et al.* [KHH*07], using advanced CPU optimizations, showed 5 metaballs on an Intel Core Duo 2.16 GHz at 512^2 even close up at about 14 fps on the video available on their website. Our method can render a similar scene (with metaballs filling about 80% of the screen area) at more than 50 fps, making it 3.5 times faster. A publicly available demo of GPU-based polygonization [Ura06] renders 25 metaballs at 25 fps, however the surface is not so smooth. Our method can handle even small particles smoothly at almost the same speed at 640×480.

Figure 10 shows an example of using our method to visualize the results of a particle-based fluid simulation. Note that even very small particles are rendered smoothly. Figure 11 shows the visualization of a molecular dynamics simulation. Compared to simply drawing spheres, using metaballs gives a higher quality picture. Figure 12 shows an example of using our method to render a particle-system-based fountain.

7. Discussion

In this section, we discuss the special cases for ray-isosurface tests in our algorithm as well as further optimizations.

Special Cases. We will address a few problematic situations that may arise with our algorithm (Figure 9). When several effective spheres intersect the viewing ray at the same point (Figure 9(a)), by just comparing the ray parameters, only one metaball ID is returned. In this case, we would have to check if the ID is already stored in the list or not, and discard the fragments containing that ID if it is, however, this would drastically increase the cost of depth peeling. Fortunately, in our experiments using 32-bit floating point computations, omitting this check did not lead to any visible artifacts.

For very dense groups of metaballs (Figure 9(b)), the number of items in the ID list might exceed the upper limit of $K = 4N_{buf} - 2$, in which case the intersection test cannot be performed, leading to holes. When this happens, we instead treat the ray-sphere intersection as a ray-isosurface intersection. Using $K = 14$ in our experiments, we did not see any visual artifacts.

There are cases when the viewing ray only intersects the isosurface after crossing a number of other metaballs (Figure 9(c)). As long as we do not exceed K, the upper limit for the ID list's capacity, we can keep the isosurface visible by just repeating depth peeling until it terminates.

Further Optimizations. Depth peeling and isosurface test make up the major part of our method. For the latter, we tried to replace the expensive computation of the Bézier coefficients with precomputation and runtime texture fetches. However, since the texture accesses had little coherence, it actually hurt the performance. As a further possibility for a speedup, future hardware support for depth peeling [BCL*07] would be very helpful.

8. Conclusion and Future Work

We have achieved fast ray casting of metaballs on the GPU. One challenge of using the GPU is that we have to extract the metaballs necessary for isosurface tests with limited memory. Our algorithm efficiently finds the metaballs contributing to the isosurface on the viewing ray by keeping metaball ID lists and updating them after each pass of depth peeling. For faster rendering, we have proposed the following optimizations:

1. fast rendering of spheres via point sprites (Section 5.1),
2. metaball culling via occlusion queries (Section 5.2),
3. tile-based depth peeling (Section 5.3), and
4. recursive short-stack Bézier Clipping (Section 5.4).

We have demonstrated that our method is useful for previewing the results of particle-based simulations such as fluid dynamics and molecular dynamics.

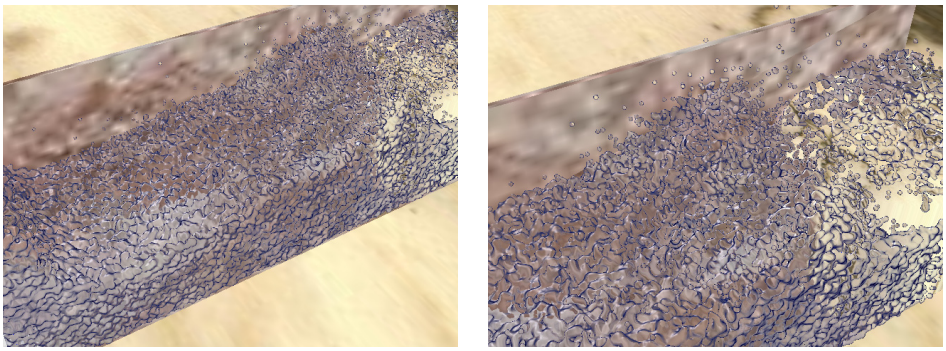


Figure 10: *Visualization of a particle-based fluid simulation with about 120K metaballs. The scene on the left (4.7 fps) is closed up and displayed on the right (4.4 fps). Even small splashes can be seen clearly. The front face of the tank was removed to show the water inside.*

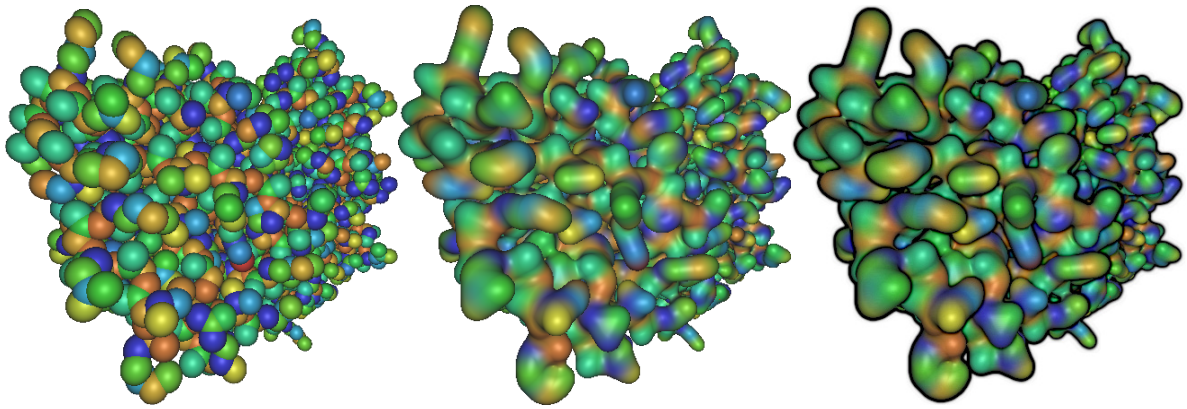


Figure 11: *Visualization of a molecular dynamics simulation with 2,878 metaballs. When simply drawing a sphere for each atom (left, 3971.9 fps), the bonds between the atoms are hard to see. When using our method they become more clear (center, 22.9 fps). Applying Luft et al.'s image enhancement method [LCD06] increases the feeling of depth perception (right, 22.8 fps).*

For future work, we would like to use the GPU to achieve further speed gains. In order to render objects exhibiting complex refraction and transparency effects using image-based approaches [Wym05], we would also like to find an efficient way to calculate not only the isosurface closest to the viewpoint, but also other isosurfaces further along the viewing ray.

Acknowledgements

Our special thanks go to Yonghao Yue for implementing an optimized CPU-based ray caster and Kenichi Yoshida for fruitful discussions. We would also like to thank Tsuneya Kurihara and Takashi Imagire for proofreading the early version of this paper, Prometech Software Inc. and Protein Data Bank for providing the simulation data of fluid dynamics and molecular dynamics, respectively. Finally, we are grateful to the anonymous reviewers for their helpful suggestions, and the members of our laboratory for their great encouragement.

References

[BCL*07] BAVOIL L., CALLAHAN S. P., LEFOHN A., JO A. L. D. C., SILVA C. T.: Multi-fragment effects on the GPU using the k-buffer. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (2007), pp. 97–104.

[Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph. 1*, 3 (1982), 235–256.

[BPCS06] BERNADON F. F., PAGOT C. A., COMBA J. L. D., SILVA C. T.: GPU-based tiled ray casting using depth peeling. *journal of graphics tools 11*, 4 (2006), 1–16.

[BSK04] BOTSCH M., SPERNAT M., KOBBELT L.:

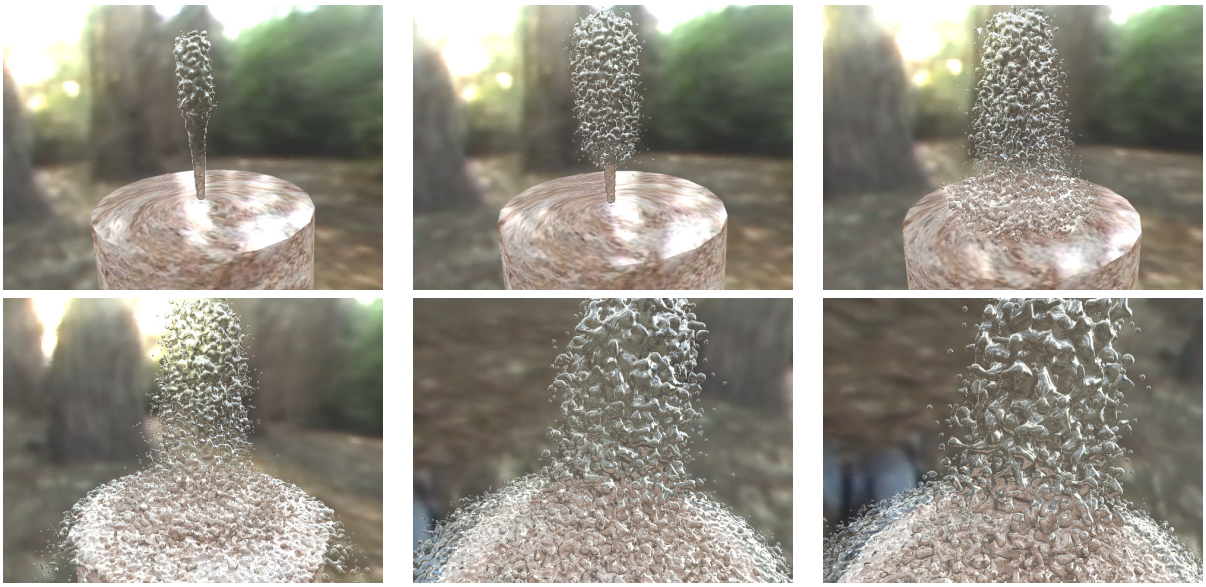


Figure 12: *Screenshots from a particle-system-based animation of a fountain with about 100K metaballs. The rendering speeds were 6.2 fps for the upper-left and 2.3 fps for the lower-right.*

Phong splatting. In *Eurographics Symposium on Point-Based Graphics 2004* (2004), pp. 25–32.

[CSS97] CAMPAGNA S., SLUSALLEK P., , SEIDEL H.-P.: Ray tracing of spline surfaces: Bézier clipping, chebyshev boxing, and bounding volume hierarchy - a critical comparison with new results. *The Visual Computer 13* (1997), 265–282.

[Eve02] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation, 2002. http://developer.nvidia.com.

[GG07] GUENNEBAUD G., GROSS M.: Algebraic point set surfaces. In *Proc. of SIGGRAPH '07* (2007), p. 23.

[IYDN06] IWASAKI K., YOSHIMOTO F., DOBASHI Y., NISHITA T.: Real-time rendering of point-based water surfaces. In *Proc. of Computer Graphics International 2006* (2006), pp. 102–114.

[KHH*07] KNOLL A., HIJAZI Y., HANSEN C. D., WALD I., HAGEN H.: Interactive ray tracing of arbitrary implicit functions. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007).

[KOKK06] KANAI T., OHTAKE Y., KAWATA H., KASE K.: GPU-based rendering of sparse low-degree implicit surfaces. In *Proc. GRAPHITE '06* (2006), pp. 165–171.

[LB06] LOOP C., BLINN J.: Real-time GPU rendering of piecewise algebraic surfaces. In *Proc. SIGGRAPH '06* (2006), pp. 664–670.

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. In *Proc. SIGGRAPH '87* (1987), pp. 163–169.

[LCD06] LUFT T., COLDITZ C., DEUSSEN O.: Image enhancement by unsharp masking the depth buffer. *ACM Transactions on Graphics 25*, 3 (jul 2006), 1206–1213.

[Mam89] MAMMEN A.: Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications 9*, 4 (1989), 43–55.

[MI87] MURAKAMI S., ICHIHARA H.: On a 3D display method by metaball technique. *Journal of papers given by at the Electronics Communication (in Japanese) J70-D*, 8 (1987), 1607–1615.

[NHK*85] NISHIMURA H., HIRAI M., KAWAI T., SHIRAKAWA I., OMURA K.: Object modeling by distribution function and a method of image generation. *Journal of papers given by at the Electronics Communication Conference (in Japanese) J68-D*, 4 (1985), 718–725.

[NN94] NISHITA T., NAKAMAE E.: A method for displaying metaballs by using Bézier Clipping. *Computer Graphics Forum 13*, 3 (1994), 271–280.

[NSK90] NISHITA T., SEDERBERG T. W., KAKIMOTO M.: Ray tracing trimmed rational surface patches. In *Proc. SIGGRAPH '90* (1990), pp. 337–345.

[NVI] NVIDIA: *The CUDA Homepage*. http://developer.nvidia.com/cuda.

[PSS*06] PABST H.-F., SPRINGER J. P., SCHOLLMEYER A., LENHARDT R., LESSIG C., FROEHLICH B.: Ray

casting of trimmed NURBS surfaces on the GPU. In *Proc. of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 151–160.

[RE05] REINA G., ERTL T.: Hardware-accelerated glyphs for mono- and dipoles in molecular dynamics visualization. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization 2005* (2005), pp. 177–182.

[Sch90] SCHNEIDER P. J.: A Bézier curve-based root-finder. *Graphics gems* (1990), 408–415.

[SGS06] STOLL C., GUMHOLD S., SEIDEL H.-P.: Incremental raycasting of piecewise quadratic surfaces on the GPU. In *Proc. of IEEE Symposium on Interactive Ray-tracing* (2006), pp. 141–150.

[She99] SHERSTYUK A.: Fast ray tracing of implicit surfaces. *Computer Graphics Forum 18*, 2 (1999), 139–147.

[SWBG06] SIGG C., WEYRICH T., BOTSCH M., GROSS M.: GPU-based ray-casting of quadratic surfaces. In *Eurographics Symposium on Point-Based Graphics 2006* (2006), pp. 59–65.

[Ura06] URALSKY Y.: *DX10: Practical Metaballs and Implicit Surfaces*. Tech. rep., NVIDIA Corporation, 2006. http://developer.nvidia.com.

[vKvdBT07] VAN KOOTEN K., VAN DEN BERGEN G., TELEA A.: Point-based visualization of metaballs on a GPU. *GPU Gems 3, Chapter 7* (2007), 123–148.

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007).

[WMG*07] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports* (2007).

[WMW86] WYVILL B., MCPHEETERS C., WYVILL G.: Data structure for soft objects. *The Visual Computer 2* (1986), 227–234.

[WT90] WYVILL B., TROTMAN A.: Ray-tracing soft objects. In *CG International '90* (1990), pp. 439–475.

[Wym05] WYMAN C.: An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics 24*, 3 (aug 2005), 1050–1053.

[YK04] YASUI Y., KANAI T.: Surface quality assessment of subdivision surfaces on programmable graphics hardware. In *SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04)* (2004), pp. 129–138.

Appendix A: The Density Functions of Various Degrees in Bézier Form

In Nishita *et al.*'s method [NN94], we need to construct equations in Bézier form for ray-isosurface tests. Here we first describe how to compute the coefficients of the equations for one metaball, then for N metaballs that intersect with the viewing ray in a given interval.

Consider the effective sphere of metaball i that intersects the viewing ray. Let $s \in [0,1]$ be a parameter within the intersected interval. For the degree-M density function $f_i(s)$ in Bézier form, the coefficients $\{d_m^i\}$ at the interval are:

for the quartic polynomial,

$$d_0^i = d_1^i = d_3^i = d_4^i = 0, \quad d_2^i = \frac{8}{3}a_i^2, \tag{3}$$

and for the degree-six polynomial,

$$d_0^i = d_1^i = d_5^i = d_6^i = 0, \tag{4}$$

$$d_2^i = d_4^i = \frac{16}{27}a_i^2, \quad d_3^i = \frac{8(8a_i+5)a_i^2}{45}, \tag{5}$$

where $a_i = \frac{D_i}{R_i}$, D_i is the determinant of the quadratic equation for the ray-sphere intersection test, and R_i is the effective radius of metaball i.

For N metaballs that intersect with the viewing ray in a given interval, the coefficients of the density function for metaball i are computed by clipping the Bézier curve defined by $\{d_m^i\}$ using de Casteljau's algorithm. By putting the density functions in Bézier form into Equation 1, we obtain the equation for the ray-isosurface test at that interval.

Appendix B: Finding a Perspective Projected Sphere's On-screen Region

Consider a sphere i with radius R_i centered at (x_i, y_i, z_i) in the viewing coordinate system. The bounding rectangle for the sphere on the screen can be calculated with the following formula:

$$x = \frac{z_{screen}(x_i z_i \pm R_i\sqrt{x_i^2 + z_i^2 - R_i^2})}{z_i^2 - R_i^2}, \tag{6}$$

$$y = \frac{z_{screen}(y_i z_i \pm R_i\sqrt{y_i^2 + z_i^2 - R_i^2})}{z_i^2 - R_i^2}, \tag{7}$$

where z_{screen} is the distance from the viewpoint to the screen. Using point sprites, we render squares that cover this region.