Real-Time Metaball Ray Casting with Fragment Lists

L. Szécsi and D. Illés

TU Budapest

Abstract

In this paper we describe a method for rendering particle-based medium representations. The algorithm builds per-pixel lists of relevant metaballs, then incrementally constructs a piecewise polynomial approximation of summed metaball densities along rays, and finds intersections with the isosurface using those. This new approach scales well for a high number of particles, it can handle local extremities of depth complexity robustly, and it does not suffer from the inaccuracies and limitations of screen-space filtering approximation methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Metaballs [NHK*85, Bli82] are widely used to visualize the results of particle simulations, from hydrodynamics to molecular modeling. Such simulations, even with hundreds of thousands of particles, can now be performed in real-time, and therefore an efficient visualization method is needed.

Each metaball has a density function, and a set of metaballs represents a smooth surface as an isosurface of the density field. The isosurface is mainly visualized by polygonization (the marching cubes algorithm [LC87]), ray casting [NN94, Bol10], or approximated by screen-space filtering of a solid estimating surface [vdLGS09]. The marching cubes algorithm can suffer from voxelization artifacts for low resolution grids, or be extremely computation and memory intensive when using high resolution. Ray casting produces high quality smooth surfaces, but the ray–isosurface intersection test is also very expensive to perform. Finally, screen-space filtering can produce visually acceptable results in real-time under controlled conditions, but it is not useful when close-ups or other exact details are required — as in molecule visualization.

We propose a fast ray casting method to render metaballs on the GPU (Figure 1). The approach is similar to that of Kanamori et al. [KSN08], gathering metaballs along rays, and maintaining relevant density information as we process them. However, we replace depth peeling (which is multi-pass, and allows limited depth complexity) with gathering per-pixel fragment lists in a single pass. Also, we use a piecewise cubic approximation of the density function, which allows for faster intersection computation, replacing Bezier clipping. These points also make the main contributions of the paper. Otherwise, all considerations by Kanamori et al. concerning related work apply. The resulting method scales well with the number of metaballs, allowing real-time visualization of massive particle simulations.

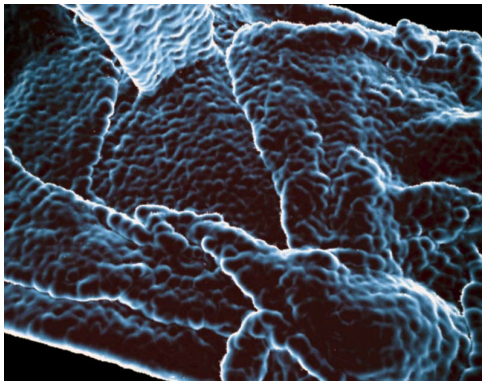

Figure 1: *Close-up of a scene with 100,000 metaballs rendered using cel shading at 13 FPS.*

2. The metaball model

The *density function* ρ_j for metaball j is a function of r, the distance from the metaball center $\mathbf{p}_j$, and monotonically decreases with r. For M metaballs, the shape of the curved

DOI: 10.2312/conf/EG2012/short/093-096

surface at density value L is defined by the points $\mathbf{x}$ satisfying

$$\rho(\mathbf{x}) = \sum_{j=0}^{M-1} \rho_j\left(\left\|\mathbf{x}-\mathbf{p}_j\right\|\right) - L = 0, \qquad (1)$$

where $\rho(\mathbf{x})$ is called the *total density*. Although Gaussian and hyperbolic kernel density functions can also be used in theory, they are impractical for computations as all terms of the summation in Equation 1 are non-zero. Instead, ρ_j is usually defined to have finite support R_j, with $\rho_j(r) = 0$ where $r \geq R_j$. Within $r < R_j$, $\rho_j(r)$ is typically given in polynomial form, which is not only easy to evaluate, but it can be chosen to meet continuity criteria. One of the most widely used density functions is the elegant fifth-degree version by Perlin [Per02] which ensures tangential continuity. The six-degree polynomial proposed by Wyvill et al. [WMW86] can be written as a function of r^2, meaning that taking a square root for the vector norm computation in Equation 1 is not necessary, and $\rho(\mathbf{x})$ itself is piecewise polynomial.

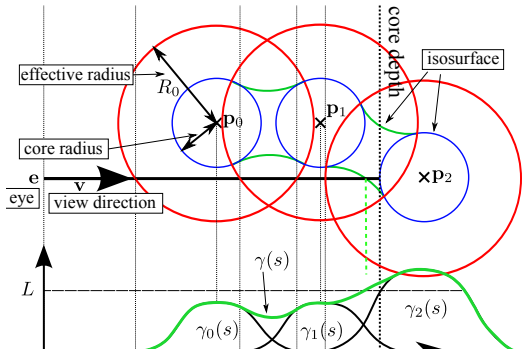

Figure 2: *Nomenclature for metaballs and their density functions along a ray.*

Figure 2 explains the nomenclature. We call R_j the *effective radius* of metaball j, and the sphere of centre $\mathbf{p}_j$ and radius R_j is the *effective sphere*. The radius for which the density function equals L is the *core radius*. The *core sphere* of this radius is always contained within the isosurface. If we wish to visualize this implicit surface using ray casting, we need to be able to find the intersection of the isosurface and a ray. Let the ray equation be

$$\mathbf{y}(s) = \mathbf{e} + s\mathbf{v},$$

where $\mathbf{e}$ is the eye position and $\mathbf{v}$ is the viewing direction. Substituting this into Equation 1 gives:

$$\rho\left(\mathbf{y}(s)\right) = \sum_{j=0}^{M-1} \rho_j\left(\left\|\mathbf{e}+s\mathbf{v}-\mathbf{p}_j\right\|\right) - L.$$

Let us introduce $\gamma(s) = \rho\left(\mathbf{y}(s)\right)$ for the total density as a function of the ray parameter. Let $\gamma_j(s)$ be the jth term in the sum, which is an individual kernel density, also as the function of the ray parameter.

3. Polynomial approximation

If ρ_j can be written as a function of r^2, $\gamma(s)$ will be piecewise polynomial. The fact that such a choice of a density function is possible is an important motivation for using a piecewise polynomial approximation $\tau(s)$ for $\gamma(s)$. We construct $\tau(s)$ as a sum of $\tau_j(s)$ individual piecewise polynomial approximations of $\gamma_j(s)$ metaball density functions. Further on in this discussion, we are going to assume piecewise cubic $\tau_j(s)$, and we provide a method for their construction from density kernel functions $\rho_j(r)$. However, it is easy to extend our method to higher degrees, and it is also possible to use exact polynomials for appropriately chosen density functions. The use of cubics introduces subtle, temporally coherent, view-dependent distortion, but it allows for simpler implementation and faster evaluation.

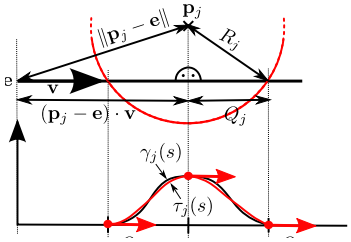

Figure 3: *Effective interval computation and third order spline approximation of the density function.*

The original $\gamma_j(s)$ is non-zero wherever the ray intersects its effective sphere (Figure 3). For the *midpoint* of the intersected interval, the ray parameter m_j can be computed as

$$m_j = (\mathbf{p}_j - \mathbf{e}) \cdot \mathbf{v},$$

and the half-length of the intersected segment (using the Pythagorean theorem) is

$$Q_j = \sqrt{R_j^2 - \left(\left\|\mathbf{p}_j - \mathbf{e}\right\|^2 - m_j^2\right)}.$$

We call the interval $\left[m_j - Q_j, m_j + Q_j\right]$ the *effective interval*. For all practical, smooth density functions, the values and derivatives of $\gamma_j(s)$ are zero at the endpoints of the effective interval (also shown in Figure 3). The derivative is also zero at the midpoint. We construct $\tau_j(s)$ by fitting a third-order spline on these characteristic points and tangents. This results in two cubic segments in $\left[m_j - Q_j, m_j\right]$ and $\left[m_j, m_j + Q_j\right]$. Thus, with $\mathbf{s}$ as $\left(s^3, s^2, s, 1\right)$,

$$\tau_j(s) = \begin{cases} 0 & \text{if } s \notin \left[m_j - Q_j, m_j + Q_j\right], \\ \mathbf{s} \cdot \mathbf{u}_j^{\uparrow} & \text{if } s \in \left[m_j - Q_j, m_j\right), \\ \mathbf{s} \cdot \mathbf{u}_j^{\downarrow} & \text{if } s \in \left[m_j, m_j + Q_j\right], \end{cases}$$

where the formulae for the coefficients can be found by algebraically solving the system of equations written for the values and the derivatives. This can also be seen as a linear reparametrization of the *smoothstep* function.

4. Ray decomposition

As $\tau(s)$ is the sum of piecewise cubics, it is piecewise cubic itself, with all end- and midpoints of effective intervals as *subdomain separators*. Let us denote these points in increasing order by s_i, with $i = 1 \ldots n$, where n is the number of such points, and prepend $s_0 = 0$ to this ordered list.

$$\gamma(s) \approx \tau(s) = \mathbf{s} \cdot \mathbf{w}_i \quad \text{if } s_i \leq s < s_{i+1} \;\Big|\; i \in \{0, \ldots, n-1\},$$

where $\mathbf{w}_i$ are the polynomial coefficient vectors. In every subdomain, the density function can be derived from a given set of metaballs. Our goal is to identify all these subdomains, compute the approximating polynomials, and solve it for intersection with the isosurface.

Along the ray, the coefficients of $\tau(s)$ change at every subdomain separator. The change of coefficients $\Delta\mathbf{w}_i$ at s_i can be deduced from the cubic coefficients as

$$\Delta\mathbf{w}_i = \frac{\gamma_j(s_i)}{Q_j^3} \begin{cases} \alpha_i + \beta_i & \text{if } s_i = m_j - Q_j, \\ -2\alpha_i & \text{if } s_i = m_j, \\ \alpha_i - \beta_i & \text{if } s_i = m_j + Q_j, \end{cases}$$

$$\text{with } \alpha_i = (-2, 6s_i, -6s_i^2, 2s_i^3)^{\mathrm{T}}, \beta_i = Q_j(0, 3, -6s_i, 3s_i^2)^{\mathrm{T}}.$$

If the eye is not within the effective sphere of any metaball, $\mathbf{w}_0 = \mathbf{0}$. Otherwise, the cubics of the intervals that contain the eye have to be summed. Coefficients for subsequent subdomains can simply be found by applying the changes

$$\mathbf{w}_i = \mathbf{w}_{i-1} + \Delta\mathbf{w}_i.$$

5. The proposed method

Gathering a list of fragments for image pixels is possible with Shader Model 5 hardware, as it has been demonstrated in the Order-Independent Transparency (OIT) method [Eve01]. In the OIT method the fragments are gathered so that they can be sorted by depth to evaluate alpha-blending-like transparency. We use similar fragment lists to process intersected metaballs.

The algorithm consists of the following phases:

Sorting: We sort metaballs according to their *effective distance* $\left(\left\|\mathbf{p}_j - \mathbf{e}\right\| - R_j\right)$ from the camera, ascending.

Lay down depth: We render solid scene and metaball core depth into a *core depth* texture, which will later be used to cull more distant, non-contributing metaballs. The geometry rendered for core spheres is a covering billboard or some more fitting polygonal enclosing object (we did not find a significant performance difference). The pixel shader discards pixels in which the sphere is not intersected. Z-buffering is used, cores are rendered front-to-back for performance.

Gathering: We render the effective spheres of metaballs, gathering their IDs into per pixel linked lists, maintained in a read/write GPU buffer. Effective spheres are rendered with the same technique as core spheres in the previous phase. The intersected effective interval is clipped against the core depth. Finally, the metaball ID is prepended to the linked list that belongs to the pixel. Metaballs are rendered back-to-front, thus the created lists are ordered by increasing effective distance.

Ray casting: We render a full-viewport quad. For every pixel, the lists are expanded to end- and midpoint records, which are sorted and used to evaluate ray–isosurface intersection. This process is detailed in Section 6.

Shading: The normal of the isosurface at the intersection point $\mathbf{x}$ is computed as $-\nabla\rho(\mathbf{x})$, iterating over the metaballs gathered for the pixel. Then, the shading formula is evaluated to get the pixel color.

6. Ray casting

In every pixel, we need to assemble an ordered list of end- and midpoints of effective intervals of all intersected metaballs, storing distances s_i and coefficient changes $\Delta\mathbf{w}_i$. The fact that metaballs are already sorted by effective depth helps minimize local shader memory usage and sorting overhead, because it allows us to process partial subdomain lists. For every pixel, the shader performs the following steps:

1. The polynomial coefficient vector $\mathbf{w}$, which is a running variable, is initialized to zero.
2. For every metaball in the list:
   a. The effective sphere is intersected with the ray through the pixel. Separator records $(s_i, \Delta\mathbf{w}_i)$ for end- and midpoints are inserted into a list ordered by depth. This list is short and it is stored in a fixed-sized local-memory array.
   b. *Safe* separator records are those separators in the list, which are within the effective distance of the next, yet unprocessed metaball (in the *safe zone*). As metaballs are ordered, separators of further effective spheres cannot precede the safe ones. For these safe separators, the polynomial coefficients are updated, and the intersection within subdomains is evaluated. The polynomial coefficients are maintained in the running variable $\mathbf{w}$, adding $\Delta\mathbf{w}_i$ as the separators are processed. The intersection computation only requires the solution of a cubic, where the practically exclusive monotonous case can easily be identified and very effectively handled. The processed separators are discarded from the ordered list. If no metaballs are left, we can evaluate all remaining subdomains.
3. When an intersection was found, we compute the normal and shading.

We store the elements of the ordered list in a local-memory array in reverse order. Elements are removed simply by decreasing the element count. For every intersected sphere we need to insert three records in known order, meaning records of a lesser distance have to be moved only once.

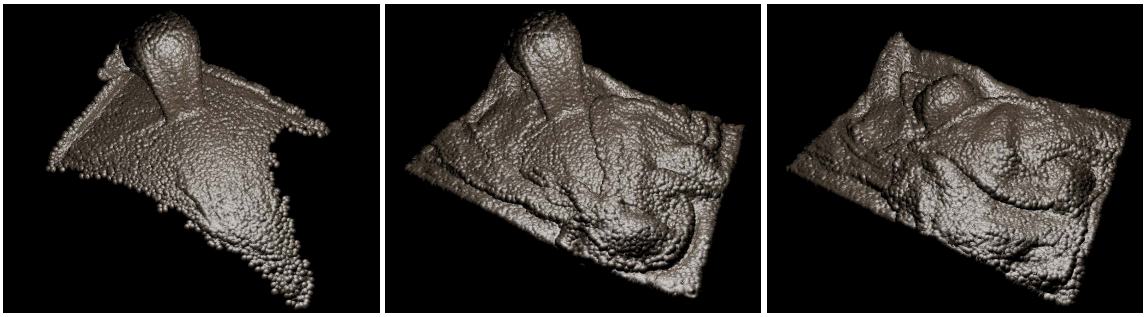

Figure 4: *Frames of a mud simulation reaching 200,000 metaballs. Frame rates are 25 FPS, 13 FPS, and 6 FPS.*

The fixed-size local memory array may be insufficient to hold separators from too many overlapping spheres, forcing us to process non-safe separators. This creates a trade-off between performance and the amount of overlapping we can handle robustly. Note that there is no limit on screen-space depth complexity, merely on the number of effective spheres a world-space point can belong to. As intersection records can be kept small, bounds in typical particle simulation scenarios are easily met.

7. Results and conclusion

Our approach was implemented in C++, using Direct3D11 and HLSL shaders. We ran our tests on a PC with an Intel Quad 2.40 GHz CPU and an NVIDIA GeForce GTX 560 Ti graphics card. According to our measurements the method runs real-time even for tens of thousands of metaballs (Table 7, Figure 4), and it can achieve interactive framerates for 1,600,000 particles. Our test scenes were simulated in *Next Limit's RealFlow*.

Metaballs	FPS	Sort	Depth	Gather	Cast
50,000	25	8	2	6	24
100,000	13	15	5	15	42
200,000	6	32	12	36	80
800,000	2	165	65	191	84
1,600,000	1	350	131	402	108

Table 1: *Overall frame rate and phase execution times in msecs at* 1024×768*, full viewport coverage.* Sort *includes CPU-GPU data transfer,* Cast *includes shading.*

The cost of the ray casting phase, which used to be the bottleneck in earlier algorithms, becomes fairly constant after a number of metaballs. The $\mathcal{O}(M)$ complexity of rendering all particles and the $\mathcal{O}(M \log M)$ cost of sorting take over at large numbers. We can conclude that our approach scales well for massive amounts of particles, with parallelized sorting and *Potentially Visible Set* solutions being the broadest avenues for further improvement. We also wish to investigate applicability to computing multiple intersections along rays, allowing for translucent shading.

Frame rates are similar to those of approximate image space techniques [vdLGS09] (22 vs 20-50 FPS, 64K metaballs), but with increasing metaball counts the image processing cost would likely be amortized. Compared to Kanamori et al. [KSN08], there is a speedup factor of 2 on equivalent hardware (6 vs 3 FPS, 200K metaballs).

This work was supported by OTKA K-719922 and 101527, and TÁMOP-4.2.1/B-09/1/KMR-2010-0002.

References

[Bli82] BLINN J.: A generalization of algebraic surface drawing. *ACM Transactions on Graphics (TOG) 1*, 3 (1982), 235–256. 1

[Bol10] BOLLA N.: *High Quality Rendering of Large Point-based Surfaces.* Master's thesis, International Institute of Information Technology, Hyderabad-500 032, INDIA, 2010. 1

[Eve01] EVERITT C.: Interactive order-independent transparency. *White paper, nVIDIA 2*, 6 (2001), 7. 3

[KSN08] KANAMORI Y., SZEGO Z., NISHITA T.: GPU-based fast ray casting for a large number of metaballs. In *Computer Graphics Forum* (2008), vol. 27, pp. 351–360. 1, 4

[LC87] LORENSEN W., CLINE H.: Marching cubes: A high resolution 3D surface construction algorithm. In *ACM Siggraph Computer Graphics* (1987), vol. 21, ACM, pp. 163–169. 1

[NHK*85] NISHIMURA H., HIRAI M., KAWAI T., KAWATA T., SHIRAKAWA I., OMURA K.: Object modeling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan 68*, Part 4 (1985), 718–725. 1

[NN94] NISHITA T., NAKAMAE E.: A method for displaying metaballs by using bézier clipping. In *Computer Graphics Forum* (1994), vol. 13, Wiley Online Library, pp. 271–280. 1

[Per02] PERLIN K.: Improving noise. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 681–682. 2

[vdLGS09] VAN DER LAAN W., GREEN S., SAINZ M.: Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), ACM, pp. 91–98. 1, 4

[WMW86] WYVILL G., MCPHEETERS C., WYVILL B.: Data structure for soft objects. *The visual computer 2*, 4 (1986), 227–234. 2