CHAPTER 35

RAY TRACING OF BLOBBIES

Manuele Sabbadin and Marc Droske
Weta Digital

ABSTRACT

Particles are widely used in movie production rendering for various different effects. Blobbies (aka metaballs) are a very useful primitive to bridge the intermediate regime between the bulk of a fluid and fast-moving spray particles as well as providing geometric variation to droplets. The use of anisotropy allows one to represent thin line structures better than classic isotropic shapes. Tessellation of such fine geometric structures is prone to geometric artifacts, especially under strong motion blur, which may heavily distort the surface during the shutter or because topology changes can't be represented well. Intersecting rays with the isosurface analytically has robustness and precision advantages. Operating on the original representation provides highly accurate spatial and temporal derivatives that are useful for filtering specular highlights. In this chapter we describe some algorithmic tools to robustly and efficiently intersect blobby surfaces supporting anisotropy and higher-order motion blur.

35.1 MOTIVATION

High-quality rendering of special effects elements (see Figure 35-1) requires robust and accurate representation of geometric details at various different scales. Elements such as fine spray can be represented by volumes and particles, whereas for the bulk of a fluid, morphological surfacing techniques can successfully be applied [8]. High-frequency details of implicit surfaces, especially under motion, pose challenges for tessellation-based techniques due to distortions and topology changes, which might require a large amount of motion steps to mask and special care to avoid artifacts due to potential self-intersections. Furthermore, high curvature and temporal variation of the normal cause specular highlights to be challenging to resolve. Here, spatial [6, 11] and temporal [10] antialiasing techniques yield very good results. These rely on surface derivatives to estimate the normal variation to translate into Beckmann roughness and to compute ray differentials. In

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_35

Figure 35-1. *Blobbies are largely used in visual effects to represent splashes of water. Anisotropy of the particles allows one to preserve the correct shape of the thin walls and lines of water. In this image 1,221,370 blobbies represent an exploding bowl of water.*

particular, we rely on computing up to second higher-order and mixed derivatives to be computed reliably, which can be a challenge on its own.

Blobby surfaces, as first introduced by Blinn [1], offer an analytic definition of an isosurface based on the combination of kernel functions around given particles (see Figure 35-2). The resulting surfaces are smooth and are nicely and compactly represented by points with some parameters. Motion can be expressed in a very natural way in a Lagrangian formulation. However, in their basic form the resulting surfaces are more suitable for molecular visualization than for fluids. Their wobbly appearance makes it difficult to represent thin structures. The definition of blobbies extends, however, easily to anisotropic surfacing [12], which overcomes these issues and makes them a compelling modeling representation for various forms of splashes and fluid droplets.

Because the surface is implicitly defined by particles that interact with each other, finding the intersections both robustly and efficiently requires some extra care. We describe some ingredients for using ray traced blobbies in practice:

- Revisit anisotropic blobby particles.
- Bounding volume hierarchy (BVH) traversal tailored to interval refinement methods [9].
- Computing tight bounds of individual blobby functions.

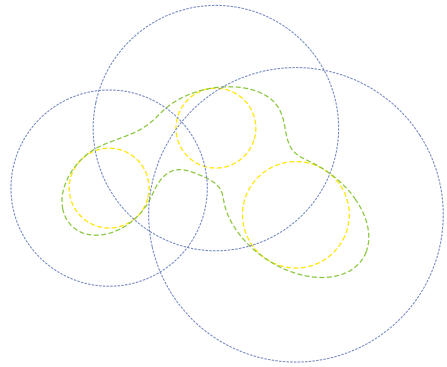

Figure 35-2. *Blobby field from three anisotropic particles with corresponding isosurface (green). With each particle we associate an inner sphere (orange) and a bounding sphere (blue), the smallest sphere that contains the support region.*

35.2 ANISOTROPIC BLOBBIES

A blobby particle B_i is represented by an implicit field $\psi_i : [t_0, t_1] \times \mathbb{R}^d \to \mathbb{R}$ defined on time in $[t_0, t_1]$ and space. A set of blobbies defines an implicit field $\phi(t, \mathbf{x})$ in the following way:

$$\phi(t, \mathbf{x}) = \sum_i \psi_i(t, \mathbf{x}) - T, \tag{35.1}$$

where T is a threshold parameter that influences the blending of the different blobbies (see Figure 35-2). To visualize $\phi(t, \mathbf{x})$, we are interested in the isosurface defined by $\mathcal{M}(t) = \{\phi(t, \cdot) = 0\}$.

We focus on the classic blobby variant

$$\psi_i(t, \mathbf{x}) = \begin{cases} \left(1 - R_i^{-2} \|\mathbf{x} - \mathbf{x}_i(t)\|^2\right)^3 & \|\mathbf{x} - \mathbf{x}_i\| < R_i, \\ 0 & \text{otherwise}, \end{cases} \tag{35.2}$$

where $\mathbf{x}_i(t)$ defines the center of the particle at time t and R_i is the radius of the influence region. We denote with R_i the *bounding radius* of the blobby B_i.

This can easily be generalized to anisotropic particles by writing it in the form

$$\psi_i(t, \mathbf{x}) = k\left(g\left(\mathbf{x} - \mathbf{x}_i(t), \mathbf{x} - \mathbf{x}_i(t)\right)\right), \quad \text{where} \quad k(y) = (1 - y)^3 \tag{35.3}$$

and g is a scalar product $g_A(\mathbf{u}, \mathbf{v}) = \langle A\mathbf{u}, A\mathbf{v} \rangle$ that encodes the anisotropy and size. In particular, for a unit basis $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2$ and radii R_0, R_1, R_2, we can set

$A_i = \text{diag}(\frac{1}{R_0}, \frac{1}{R_1}, \frac{1}{R_2}) \cdot [\mathbf{b}_0\ \mathbf{b}_1\ \mathbf{b}_2]^T$. The values R_i can be easily chosen depending on T such that the isosurface of an isolated blobby describes an ellipsoid with the prescribed lengths r_1, r_2, r_3 of the axes. In the case of anisotropic particles, the *bounding radius* is defined as the maximum of $\{R_0, R_1, R_2\}$. We also define r_i, the *inner radius* of the blobby B_i, as the radius of the largest sphere that is contained in the isosurface, if B_i is not influenced by any other blobby. It is equal to the minimum value of $\{r_1, r_2, r_3\}$.

Expressions like the shape operator $\mathcal{S}$, the temporal derivative of the normal, or the derivative of the intersection distance (see [10]) are easily obtained through basic derivatives of the level set function such as $\partial_t \phi$, $\nabla_x \phi$, $\text{Hess}\, \phi$, and $\partial_t \nabla_x \phi$. In particular, the shape operator corresponds to the matrix representation of the *Weingarten map*:

$$\mathcal{S} = \frac{1}{\|\nabla_x \phi\|} P[\nabla_x \phi]\, \text{Hess}\phi\, P[\nabla_x \phi], \quad \text{where} \quad P[v] = (\text{id} - v \otimes v), \tag{35.4}$$

which is useful to compute normal derivative in direction v as $D_v N = \mathcal{S}\, v$.

Setting $g_i(t, x) = \langle A_i(x - x_i(t)), A_i(x - x_i(t)) \rangle$, we have, for example,

$$\partial_t \nabla_x \psi_i(t, x) = k''(g_i(t, x))\, \partial_t g_i(t, x)\, \nabla_x g_i(t, x) + k'(g_i(t, x))\, \partial_t \nabla_x g_i(t, x), \tag{35.5}$$

where

$$\begin{aligned} \partial_t g_i(t, x) &= -2 \left\langle A(x - x_i(t)), A\, \partial x_i(t) \right\rangle, \\ \nabla_x g_i(t, x) &= 2A^T A(x - x_i(t)), \\ \partial_t \nabla_x g_i(t, x) &= -2A^T A\, \partial_t x_i(t). \end{aligned} \tag{35.6}$$

Motion blur is expressed simply as a parametric form of the center $x_i(t)$ depending on t. The equations can easily be extended to support a time-dependent metric $A(t)$ to represent, for example, oscillations and spin.

35.3 BVH AND HIGHER-ORDER MOTION BLUR

We use a classic BVH to store blobbies and to identify particles whose supports overlap with a ray segment. Each blobby is represented as a sphere inside the BVH (even for the anisotropic case, as we will discuss in Section 35.4). For a generic blobby B_i, we store its bounding radius R_i and the inner radius r_i. To tackle the motion of each blobby, we also store its velocity v_i and acceleration a_i at time t_0. This will allow us to represent higher-order motion blur, instead of just a linear motion. During the construction of a BVH, it is important to create bounding boxes as tight as possible to the actual

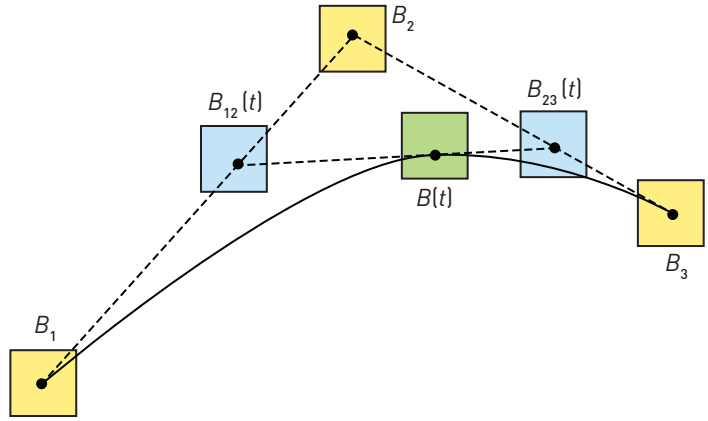

Figure 35-3. *Higher-order motion bounds: Given the bounds B_1, B_2, and B_3 that contain the control points of the input curve, $B_{12}(t) = (1 - t)B_1 + tB_2$ and $B_{23}(t) = (1 - t)B_2 + tB_3$ are bounds of the two control points of the first iteration of the de Castlejau interpolation. Eventually, $B(t) = (1 - t)B_{12}(t) + tB_{23}(t)$ contains the curve evaluated at time t.*

geometry. When objects are moving, this task becomes trickier. The simplest solution, using a bounding box that contains the entire trajectory of the object, will become inefficient under fast motion as bounds for a specific ray time become loose. A better option is to exploit velocity and acceleration to interpolate bounds in time on leaf nodes and to propagate this motion higher up the tree.

Linear motion blur can easily be handled by linearly interpolating the bounding boxes during traversal according to the ray time [3], which yields much tighter bounds than growing the bounding boxes to contain the entire motion path. Because de Casteljau's algorithm is an iterated version of linear interpolation, this can easily be extended to higher-order Bézier interpolation: the control points can be used to define control bounding boxes that are Bézier-interpolated during the traversal (see Figure 35-3).

Therefore, we only need to convert the incoming parabola given by velocity v and acceleration a defined on [0, 1] as

$$\mathbf{x}_i(t) = \mathbf{p}_i(0) + t\mathbf{v}_i + \frac{1}{2}t^2\mathbf{a}_i \tag{35.7}$$

into Bézier form:

$$\mathbf{B}_i(t) = (1 - t)^2\mathbf{P}_{i,0} + 2(1 - t)t\mathbf{P}_{i,1} + t^2\mathbf{P}_{i,2}. \tag{35.8}$$

The change of basis is given by $\mathbf{P}_{i,0} = \mathbf{x}_i(0) = \mathbf{p}_i$, $\mathbf{P}_{i,1} = \frac{1}{2}\mathbf{v}_i + \mathbf{P}_{i,0}$ and $\mathbf{P}_{i,2} = \frac{1}{2}\mathbf{a}_i + \mathbf{v}_i + \mathbf{p}_i = \mathbf{x}_i(1)$. From these the bounds of the control points for each

leaf node are obtained, which defines three Bézier control bounding boxes. Again analogously to the linear interpolation case, the control bounding boxes are aggregated up toward the root.

35.4 INTERSECTION METHODS

We define $r = (\mathbf{o}, \mathbf{d})$ as the ray with origin $\mathbf{o}$, direction $\mathbf{d}$, and limits $[s_{\text{min}}, s_{\text{max}}]$, parameterized as $r(s) = \mathbf{o} + s\mathbf{d}$.[1] We want to determine an intersection distance $s_{\text{int}} \geq 0$ that is the minimum $s \in [s_{\text{min}}, s_{\text{max}}]$ such that $\phi(t, r(s_{\text{int}})) = 0$. We indicate the hit point with $\mathbf{h}$. For an interval I (along the ray) in which $\phi(t, r(s))$ assumes both positive and negative values and is monotone, we can apply any root-finding method to find $\mathbf{h}$ within that interval. We can consider the bounding box enclosing the entire set of blobbies and intersect it with the ray r, finding a first guess for such an interval. Then, bisection can be applied iteratively until it satisfies the above-mentioned conditions. This approach works in theory, but it is not feasible in practice, as the amount of blobbies that define the implicit field ϕ can be very large. This means that every time we want to evaluate $\phi(t, \mathbf{x})$, we need to sum up the contribution from all the blobbies. Any root-finding algorithm would require $\phi(t, \mathbf{x})$ to be computed more than once, at different points along the ray. Given the finite support of the kernel function, only a few of them will contribute to the value $\phi(t, \mathbf{x})$. Naturally, we can restrict the number of blobbies we use to those whose bounding boxes intersect r. Even so, we are potentially considering a large amount of blobbies that are too far away from $\mathbf{h}$ to contribute to its computation. Our aim is to find the set of blobbies $\mathcal{A}$ required by the root-finding algorithm (i.e., it contains all the blobbies B_i for which $\psi_i(t, \mathbf{h}) \neq 0$) while discarding as many as possible.

The main steps of our algorithm are the following:

1. Determine $\mathcal{A}$ and $I_0 \subseteq [s_{\text{min}}, s_{\text{max}}]$, which is our first guess for the interval I.
2. Refine the interval I_0 iteratively, proceeding in front-to-back order, until we obtain an interval I_n that contains exactly one root.
3. Find the root inside the interval I_n.

We will focus on the first two steps, as the third consists of using a standard iterative root-finding algorithm.

[1] We use s to parameterize the ray to avoid conflict with the time denoted by t.

35.4.1 DETERMINE THE ACTIVE BLOBBIES

As we have previously seen, given a blobby B_i, we can define two different radii on it: The first one is the bounding radius R_i, which refers to the region of influence of B_i (i.e., $\psi_i(\mathbf{x}) = 0$ when $\|\mathbf{x} - \mathbf{x}_i\| > R_i$). The second radius is the inner radius r_i, which represents the radius of the largest sphere that is contained in the isosurface of B_i if not influenced by any other blobby. We associate a sphere to each of them: respectively, the bounding sphere $\mathcal{S}_{\text{bound},i}$ and the inner sphere $\mathcal{S}_{\text{inner},i}$. Each time we intersect the ray r with B_i, we obtain four values: $S_i[\text{min}]$, $S_i[\text{max}]$, $s_i[\text{min}]$, and $s_i[\text{max}]$. The first two quantities are the values of s that lead to the two intersections on the bounding sphere $\mathcal{S}_{\text{bound},i}$, while the latter two refer to the intersections on the inner sphere $\mathcal{S}_{\text{inner},i}$. It is worth noting that we check for intersections against spheres even for the anisotropic case because the intersection test is computationally faster compared to a ray-ellipse test. As we need to be conservative in our criteria to discard a node, we set R_i to be the major radius of the bounding ellipse and r_i to be the minor radius of the inner ellipse.

To determine which blobbies we should consider and determine the interval I_0, we have to distinguish two different cases: if the ray is pointing inside (such as interior reflection or transmitting inward) or outside the surface. For both cases, we avoid adding a blobby B_i to the set $\mathcal{A}$ if the ray r does not intersect $\mathcal{S}_{\text{bound},i}$. Moreover, we traverse our BVH in a way that prioritizes the nodes closer to the ray origin, as shown in the following listing:

```
void VisitChildren(Stack& S, Ray r, Node n) {
    float t0 = r.GetNearHitpointBounding(n.Child[0]);
    float t1 = r.GetNearHitpointBounding(n.Child[1]);

    int furthest = 0;
    if (t1 > t0) furthest = 1;

    if (r.IntersectBounding(n.Child[0]) && r.IntersectBounding(n.Child[1]))
         {
        S.Push(n.Child[furthest]);
        S.Push(n.Child[1 - furthest]);
        return;
    }
    if (r.IntersectBounding(n.Child[0]))
        S.Push(n.Child[0]);
    if (r.IntersectBounding(n.Child[1]))
        S.Push(n.Child[1]);
}
```

TRACING TOWARD FRONTFACE

When we hit the isosurface from the outside, we can state that if we hit the inner sphere $\mathcal{S}_{\text{inner},i}$ of the blobby B_i, we don't need to add to $\mathcal{A}$ any node B_j

that is farther from the origin of r and whose bounding sphere $\mathcal{S}_{\text{bound},j}$ does not intersect $\mathcal{S}_{\text{inner},i}$. This is simply motivated by the fact that the blobby B_j cannot influence in any way the implicit field around the hit point (and cannot cover it because it is farther). This can be achieved by a simple check: we can discard B_j if $S_j[\text{min}] > s_i[\text{min}]$. The following listing shows a simple implementation of the method to find the closest hit if the ray origin lies outside the isosurface; we use a stack to keep track of the traversal's state:

```
// The BVH B stores all the blobbies.
void IntersectFromOutside(Ray r, Set A) {
    Stack S;
    S.Push(B.root);
    float tmax = FLT_MAX;

    while (!S.IsEmpty()) {
        Node n = S.Pop();
        // If the bounding box is farther than the hit point, it cannot
             influence it.
        if (r.GetNearHitpointBounding(n) > tmax)
            continue;

        if (n.IsLeaf()) {
            if (r.IntersectInner(n)) {
                // t defines the intersection along the ray.
                float t = r.GetNearHitpointInner(n);
                tmax = min(tmax, t);
            }
            if (r.IntersectBounding(n))
                A.Insert(n);
        }
        else
            VisitChildren(S, r, n);
    }
}
```

TRACING TOWARD BACKFACE

When we hit the isosurface from the inside, we cannot use the same argument we used in the previous case. In this case, when we hit the inner sphere $\mathcal{S}_{\text{inner},i}$, there is no guarantee that nodes that satisfy the condition $S_j[\text{min}] > s_i[\text{min}]$ won't contribute to determining the hit point **h**. Let's imagine, for example, a chain of intersecting blobbies. If we start the ray r from one side of the chain, we have to traverse all the blobbies to detect the exit point (see Figure 35-9). In this scenario, we can use a weaker condition that allows us to discard part of the blobbies along the ray: Let B_i be the node in $\mathcal{A}$ with the largest $S_i[\text{max}]$. If all the nodes B_j that we still have to visit satisfy $S_j[\text{min}] > S_i[\text{max}]$, we can stop collecting nodes, because we must have exited in between. There is a crucial difference to the previous case, in which we could discard the single blobby and continue the visit the other nodes on

the stack. However, in this case we can only ascertain when the entire visit can be terminated, but it can't be determined whether a specific particle can be discarded. This is due to the fact that the entries in our stack are not ordered by *S*[min]. Hence, a node that we will visit later during the traversal could make the current node active, even if it is actually too far away to contribute to the hit (see Figure 35-9). We tested this approach on the data set in Figure 35-13 and compared to a version of the algorithm where we collect all the blobbies along the ray. The proposed technique gives a speedup of 24% for the render time.

One could argue that we should use a heap data structure instead of a stack to support the visit, to allow the algorithm to consider the closest entry at each iteration and to be able to stop the visit earlier. It is indeed an option that would allow for some optimizations, but the trade-off of the added cost of updating the heap might not be worth it.

The following listing shows a simple implementation of the method to find the closest hit if the ray origin is located inside the isosurface. Note that, to check if we can terminate the traversal, each entry of the stack now saves the minimum distance from the ray origin at the moment of its insertion.

```
1  // The BVH B stores all the blobbies.
2  void IntersectFromInside(Ray r, Set A) {
3      Stack S;
4      S.Push(B.root);
5      float tmax = FLT_MAX;
6      bool hasToInitTmax = true;
7
8      while (!S.IsEmpty()) {
9          if (S.Top().MinDistanceFromOrigin > tmax)
10             return;
11
12         Node n = S.Pop();
13
14         if (n.IsLeaf()) {
15             if (r.IntersectBounding(n)) {
16                 if (hasToInitTmax) {
17                     tmax = r.GetFarHitpointBounding(n);
18                     hasToInitTmax = false;
19                 }
20                 else
21                     tmax = max(r.GetFarHitpointBounding(n), tmax);
22                 A.Insert(n);
23             }
24         }
25         else
26             VisitChildren(S, r, n);
27     }
28 }
```

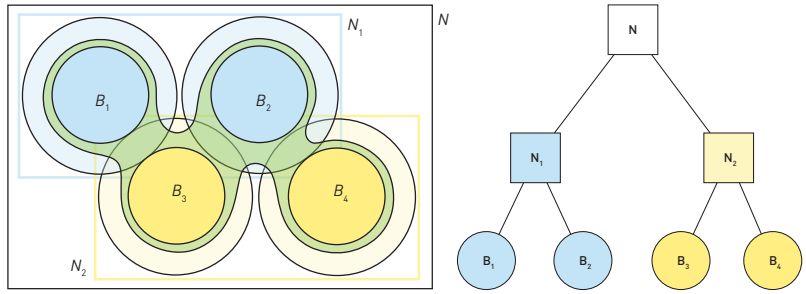

Figure 35-4. *Left: to better explain the different cases that we have to consider when creating the set $\mathcal{A}$, we will use a data set made by four blobbies, whose bounding spheres $\mathcal{S}_{\text{bound},i}$ intersect in pairs. In green we represent the isosurface. For each blobby, we used an opaque color to represent the inner sphere and a semitransparent one to represent the bounding sphere. Right: The BVH stores the nodes in a way that B_1 and B_2 share the same parent node N_1, while B_3 and B_4 share the same parent node N_2.*

We have to update the method `VisitChildren()` to push the value `MinDistanceFromOrigin` onto the stack, with each new entry. The value to push is the minimum between the old minimum distance (`MinDistanceFromOrigin` of the current top of the stack) and the distance to the node that we are going to push. For a generic node *n*, it works in the following way:

```
1 float minValue = S.Top().MinDistanceFromOrigin;
2 S.Push(n, min(minValue, r.GetNearHitpointBounding(n)));
```

EXAMPLES

In the following we provide a graphical representation to illustrate how the algorithm works. We will use the blobbies configuration in Figure 35-4 and change the ray origin and direction to present the most common situations. Figures 35-5 to 35-7 consider a ray hitting from outside the isosurface, whereas Figures 35-8 to 35-10 show a ray whose origin is inside it. In all the figures we will represent with a dotted contour the blobbies that are not part of the set $\mathcal{A}$ at the end of the traversal (for example, node B_4 in Figure 35-6).

35.4.2 INTERVAL REFINEMENT

From the set $\mathcal{A}$ we can easily find the interval $I_0 = [\min_0, \max_0]$: we simply have to intersect *r* against all the blobbies in the set and compute the boundaries of the interval in the following way:

$$\begin{aligned} \min_0 &= \textbf{Min}(S_i[\min]) \; \forall B_i \in \mathcal{A}, \\ \max_0 &= \textbf{Max}(S_i[\max]) \; \forall B_i \in \mathcal{A}. \end{aligned} \tag{35.9}$$

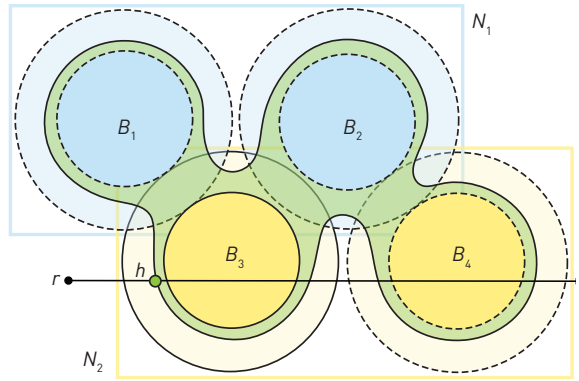

Figure 35-5. *The simplest case happens when the ray intersects an inner sphere $\mathcal{S}_{\text{inner},i}$ and no other sphere $\mathcal{S}_{\text{bound},j}$ intersects both the ray and $\mathcal{S}_{\text{inner},i}$. In the example, r does not intersect N_1, which is discarded immediately, with all its children. It first intersects $\mathcal{S}_{\text{inner},3}$ and adds the blobby to the set $\mathcal{A}$. When r intersects $\mathcal{S}_{\text{bound},4}$, the test $S_4[\min] > s_3[\min]$ succeeds, so we can discard it. The only active node is B_3.*

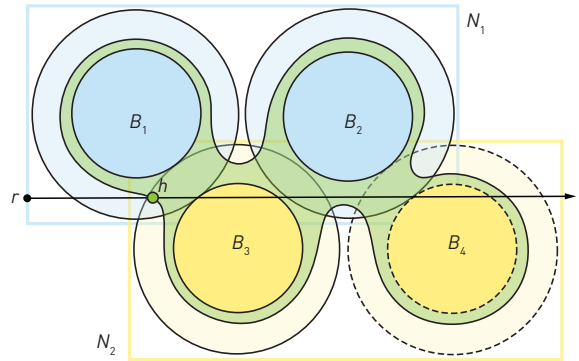

Figure 35-6. *When we intersect B_1, we don't intersect its inner sphere. In this case we cannot set an upper bound for the future intersections, as we don't know if B_1 will contribute to determining the hit point or not. Because we are visiting N_1, the next blobby we process is B_2, for which the same argument holds. When the algorithm processes N_2 and hits the inner sphere of B_3, it will set $s_3[\min]$ as an upper bound and discard B_4 at the next iteration, as $S_4[\min] > s_3[\min]$. The set of active nodes contains B_1, B_2, and B_3, even if B_2 won't contribute to computing the hit point.*

To guarantee that an interval I_i contains a single root, it is sufficient for ϕ to be monotone and the two extremes of the range B_I to have different sign. Therefore, to isolate the roots, one can use the well-known interval refinement approach as described, for example, in [7, 4]: we successively refine the ray segment until an interval is reached in which $s \mapsto \phi(t, r(s))$ is not

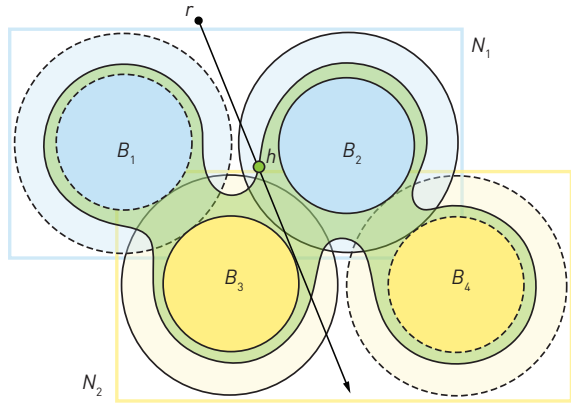

Figure 35-7. *The most unfortunate case, if we hit from outside, is when the ray r intersects many bounding spheres, but no inner spheres. In this example, we add to $\mathcal{A}$ both of the nodes B_2 and B_3 (which are required to compute the hit point). Because we are not able to set an upper bound, if r intersects any other blobby along is trajectory, it will be added to $\mathcal{A}$, no matter its distance from the origin. This cannot be avoided: there are cases where two bounding spheres intersects, but not enough to define the isosurface between them. In this case, the ray can pass in between the two blobbies and intersect something that is farther away.*

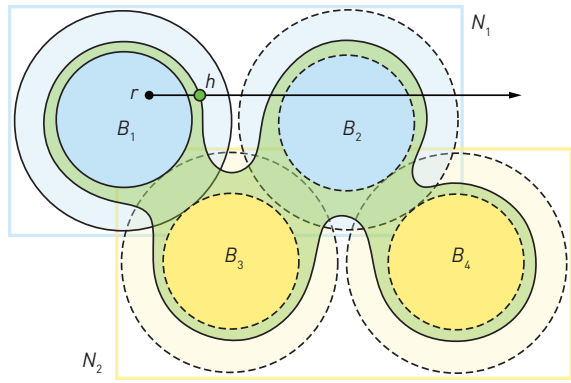

Figure 35-8. *When hitting the isosurface from inside, we can stop the process only if all the entries on the visiting stack are farther than the current upper bound. In this case, because r doesn't intersect N_2, this node doesn't appear on the stack. When we hit B_1, the only entry to compare with will be B_2 whose bounding sphere is not touching B_1. The traversal can stop immediately.*

bounded away from zero and is monotone (can't contain multiple roots), i.e., the derivative with respect to *s* is guaranteed not to be zero. The interval refinement can be done by bisection or an *Interval Newton method* [2], which uses the bounds of the derivatives in the refinement process.

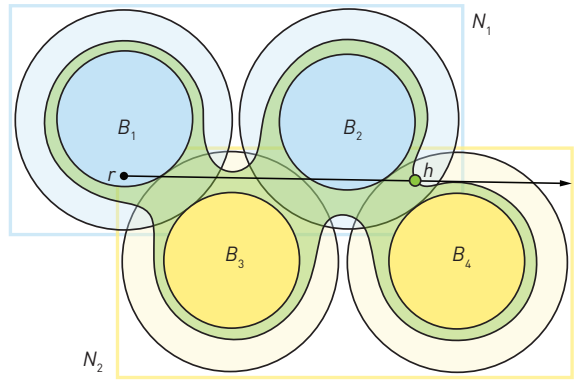

Figure 35-9. *In this example, we can see why the criteria used for an outside ray would not work for an inside ray. When we hit the inner sphere of B_1, we set S_1[max] as the upper bound. As B_2 does not intersect it, if hitting from outside, we would discard the node and keep going with the blobbies in N_2, ignoring the fact that the hit point is on its area of influence. This happens because we cannot know, beforehand, that there will be a node in N_2 acting as a bridge between B_1 and B_2 (B_3). Hence, the test for an inside ray checks all the entries on the stack.*

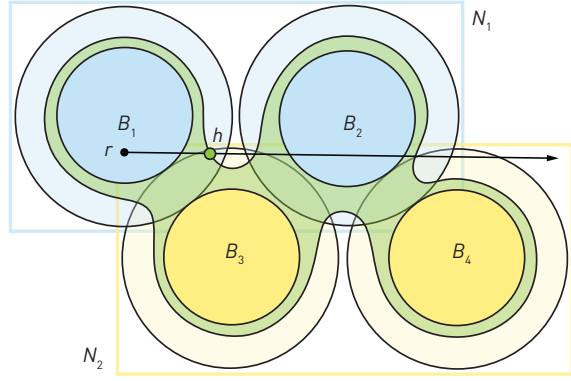

Figure 35-10. *An unfortunate case for the inside ray scenario is when we hit the isosurface soon, but we have a long chain of connected blobbies. The algorithm cannot know, a priori, that B_2 and B_4 are not needed to detect the hit point, so it will add them to the set $\mathcal{A}$, in the same fashion as Figure 35-9.*

Therefore, for a given ray $r(s) = \mathbf{o} + s\mathbf{d}$, the intersection distance interval relies on computing the bounds of $f(s) = \phi(t, r(s))$ and its derivative $f'(s)$ in an arbitrary subrange $[s_{\text{min}}, s_{\text{max}}]$. Of course, the efficiency of the refinement process depends on how tight the bounds are.

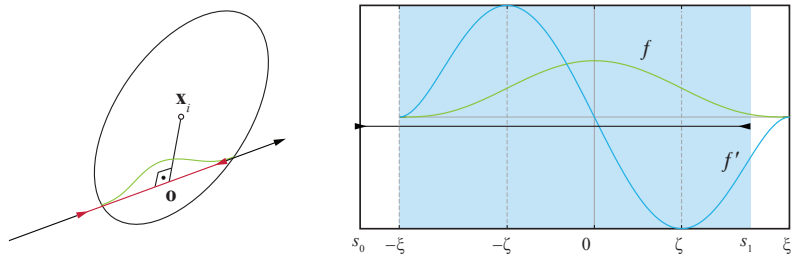

Figure 35-11. *Calculating bounds of ψ along the ray by identifying monotone intervals.*

Although interval arithmetic [4] could be applied for the computation of these bounds, this approach tends to produce too loose bounds. Instead, we proceed by computing tight bounds for the individual ψ_i (see Figure 35-11) and aggregate them through additive composition. Furthermore, we exploit the fact that $\psi_i \geq 0$ for early termination of the check for whether the bounds contain a root at all.

To compute tight bounds for ψ_i, one can simply exploit the well-known fact that the bounds of a monotone function are given by the values at the endpoints. We assume for simplicity that **o** is at the projection (with respect to g_A) of the center $x_i(t)$ to the ray

$$g_A(\mathbf{x}_i(t) - \mathbf{o}, \mathbf{d}) = 0, \tag{35.10}$$

by computing the projection and shifting $[s_{\text{min}}, s_{\text{max}}]$ accordingly.

Defining $\alpha = g_A(\mathbf{o} - \mathbf{x}_i, \mathbf{o} - \mathbf{x}_i)$ and $\beta = g_A(\mathbf{d}, \mathbf{d})$, we would then like to find the bounds of $f(s) = k(\alpha + s^2\beta) = (1 - \alpha - s^2\beta)^3$.

It can easily be seen that due to Equation 35.10 the support of f is $[-\xi, \xi]$ with $\xi = \sqrt{(1-\alpha)/\beta}$. It is monotonely increasing in $[-\xi, 0]$ and decreasing in $[0, \xi]$ and furthermore has inflection points at $-\zeta$, 0, and ζ with $\zeta = \sqrt{\frac{1}{5}}\xi$ (see Figure 35-11, right).

Therefore, computing the bounds on f in $I = [s_{\text{min}}, s_{\text{max}}]$ amounts to evaluating at the endpoints of the subintervals $[-\xi, 0] \cap I$ and $[0, \xi] \cap I$. Similarly, the bounds of f' are obtained by evaluating f' at the values $-\xi, -\zeta, 0, \zeta, \xi$ clipped to I.

NOTES

It is worth mentioning that some optimization could be done here, storing the active blobbies in an interval tree to accelerate the query of all candidates in every iteration of the interval refinement. However, due to the strategies described previously to limit the growth of $\mathcal{A}$, we have not confirmed in the implementation that the interval tree amortizes in practice.

Furthermore, when aggregating the bounds of ψ_i to determine whether $\phi(t, r(\cdot))$ may contain a root, we can discard the interval based on the fact that all ψ_i are nonnegative: if the lower bound of the partial sum is larger than $-T$, it will not recover from being bounded away from zero and therefore will not contain a root.

For a GPU implementation, the described approach of collecting the active set $\mathcal{A}$ per ray is not feasible because the number of elements is unbounded. However, the interval refinement approach can also be implemented by accumulating interval bounds on the fly in an `anyhit` program (see also [9]), adjusting $s_{\max}$ as described in Section 35.4.1. The strategy for rays inside the surface as described in Section 35.4.1 is possible in principle but requires some modifications to the stack to keep track of the minimum distance of all its elements.

35.5 RESULTS

We apply the ray tracing algorithm on two different data sets. The first scene is composed of 500 blobbies. They have been generated by randomizing their position within a unit radius sphere. Each particle comes with an initial velocity and acceleration. In Figure 35-12 we show how the BVH can be used to render higher-order motion blur in an efficient way and the benefit of having continuous derivatives all along the surface. We use the derivatives to apply the temporal antialiasing technique described by Tessari et al. [10]. The second asset is composed of 1,221,370 particles. It has been produced by simulating the explosion of a water bowl and surfacing the final result with an approach similar to that of Yu and Turk [12]. In Figure 35-13 we show how the anisotropy improves the shape of the thin lines of water produced by the explosion. We used Manuka [5] to run all our tests on a machine with 24 CPUs at a resolution of 1920×1080. In the first scene we had an average of 1.3 million rays per second, while for the second asset, where particles tend to overlap more to each other, we averaged 236,000 rays per second.

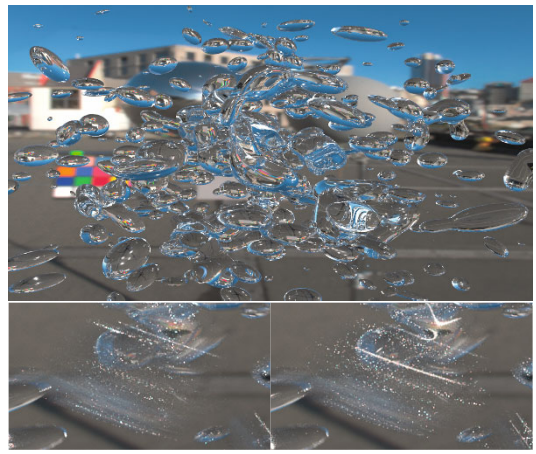

Figure 35-12. *Top: a static frame of our data set, composed of 500 anisotropic particles. Bottom: the details of one particle, after applying higher-order motion blur to it. On the right side, we exploited the derivatives to apply temporal antialiasing [10]. We limited the number of samples per pixel to 64, to show how the temporal antialiasing helps the convergence of the rendering.*

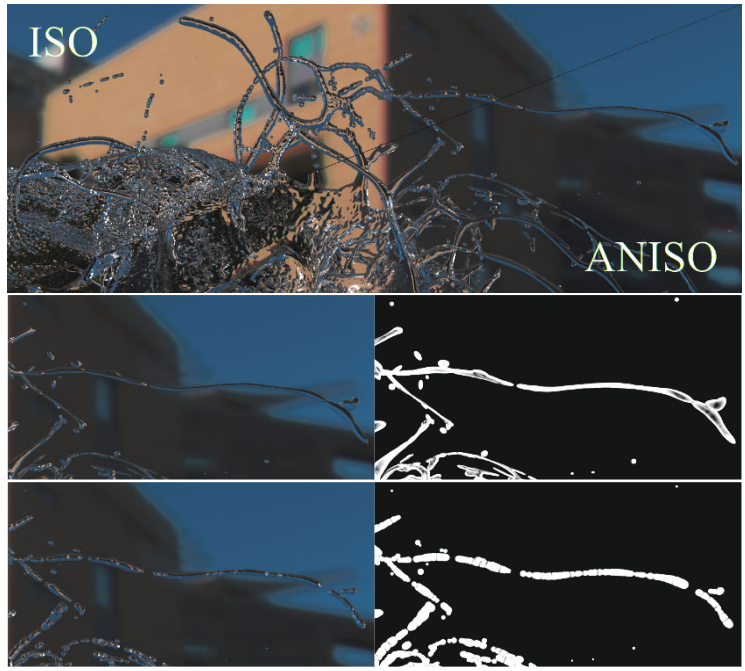

Figure 35-13. *In the case of a water bowl explosion, we have patterns of water representing thin walls and lines. Anisotropy helps in preserving the correct shape of these structures. In the top image, we can see how anisotropy compares to the isotropic case. The other images show the details of a thin line of water, with its derivatives on the right side.*

ACKNOWLEDGMENTS

We'd like to thank Louis-Daniel Poulin for his enormously helpful input in various discussions on special effects rendering.

REFERENCES

[1] Blinn, J. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982. DOI: 10.1145/357306.357310.

[2] Capriani, O., Hvidegaard, L., Mortensen, M., and Schneider, T. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 6:9–21, 2000. DOI: 10.1023/A:1009921806032.

[3] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. Ray tracing for the movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006. DOI: 10.1109/RT.2006.280208.

[4] Díaz, J. E. F. *Improvements in the Ray Tracing of Implicit Surfaces Based on Interval Arithmetic*. PhD thesis, Universitat de Girona, 2008.

[5] Fascione, L., Hanika, J., Leone, M., Droske, M., Schwarzhaupt, J., Davidovič, T., Weidlich, A., and Meng, J. Manuka: A batch-shading architecture for spectral path tracing in movie production. *ACM Transactions on Graphics*, 37(3):31:1–31:18, Aug. 2018. DOI: 10.1145/3182161.

[6] Kaplanyan, A. S., Hill, S., Patney, A., and Lefohn, A. Filtering distributions of normals for shading antialiasing. In *Proceedings of High Performance Graphics*, pages 151–162, 2016.

[7] Knoll, A. *Ray Tracing Implicit Surfaces for Interactive Visualization*. PhD thesis, School of Computing, Utah University, 2009.

[8] Museth, K. A flexible image processing approach to the surfacing of particle-based fluid animation (invited talk). In K. Anjyo, editor, *Mathematical Progress in Expressive Image Synthesis I: Extended and Selected Results from the Symposium MEIS2013*, pages 81–84. Springer Japan, 2014. DOI: 10.1007/978-4-431-55007-5_11.

[9] Singh, J. M. and Narayanan, P. J. Real-time ray-tracing of implicit surfaces on the GPU. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):261–272, 2010. DOI: 10.1109/TVCG.2009.41.

[10] Tessari, L., Hanika, J., Dachsbacher, C., and Droske, M. Temporal normal distribution functions. In *Eurographics Symposium on Rendering—DL-only Track*, pages 1–12, 2020. DOI: 10.2312/sr.20201132.

[11] Tokuyoshi, Y. and Kaplanyan, A. S. Improved geometric specular antialiasing. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 8:1–8:8, 2019. DOI: 10.1145/3306131.3317026.

[12] Yu, J. and Turk, G. Reconstructing surfaces of particle-based fluids using anisotropic kernels. *ACM Transactions on Graphics*, 32(1):5:1–5:12, Feb. 2013. DOI: 10.1145/2421636.2421641.