EUROGRAPHICS 2010 / T. Akenine-Möller and M. Zwicker
(Guest Editors)

Volume 29 (2010), Number 2

Fitted BVH for Fast Raytracing of Metaballs

Olivier Gourmel[1], Anthony Pajot[1], Mathias Paulin[1], Loïc Barthe[1] and Pierre Poulin[1,2]

[1]IRIT-VORTEX, Université Paul Sabatier, Toulouse, France
[2]LIGUM, Dept. I.R.O., Université de Montréal

Abstract

Raytracing metaballs is a problem that has numerous applications in the rendering of dynamic soft objects such as fluids. However, current techniques are either limited in the visual effects that they can render or their performance drops as the number of metaballs and their density increase. We present a new acceleration structure based on BVH and kd-tree for efficient raytracing of a large number of metaballs. This structure is built from an adapted SAH using a fast greedy algorithm and allows the visualization of several hundreds of thousands metaballs at interactive-to-real-time framerates. Our method can handle arbitrary rays to simulate any complex secondary effects such as reflections or soft shadows, and is robust with respect to the density of metaballs. We achieve this performance thanks to a balanced CPU-GPU (using CUDA) implementation of the animation, structure creation, and rendering.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible line/surface algorithms

1. Introduction

Modeling and visualizing fluids and other dynamic objects of varying topology is a difficult problem. One common representation uses a large number of implicit surfaces known as soft objects or metaballs [Blo97]. However visualizing a large number of metaballs in real time remains a difficult problem in itself due to their implicit nature.

A common method to display metaballs is ray casting, which has the benefit of rendering the surface very precisely. Since no analytical solution is known in the general case, the ray-isosurface intersection is usually computed iteratively via for instance, ray marching [Har93], Bezier clipping [NN94] or interval arithmetic [Flo08, KHK*09]. Unfortunately, computing such intersections can become problematic since the cost of evaluating the potential function of the surface increases as the number of metaballs grows. Most papers that tackle this issue try to reduce the cost and the number of evaluations of this potential function, for instance by reconstructing the potential function along the ray [WT90, NN94].

Another solution tessellates the isosurface, and then rasterizes the corresponding polygons [Ura06]. The main problem of this technique is that the resulting mesh is very dependent of the tessellation sampling grid, and thus some geometry can be missed if the grid resolution is too low. Moreover, the cost of meshing the surface can become another issue when dealing with large numbers of metaballs.

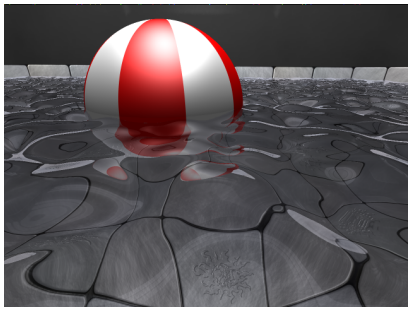

Figure 1: *Screenshot of a fluid simulation composed of 10,000 metaballs rendered with reflection and refraction using our technique. Our acceleration structure makes their visualization possible at an interactive framerate.*

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.
Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

In this paper, we present a new acceleration structure, combining aspects of a kd-tree and a BVH. This structure allows interactive raytracing of large numbers of metaballs while still capturing advanced secondary lighting effects such as true mirror reflections, translucencies and soft shadows (cf. Figure 1).

Our method aims at reducing the number of evaluations of the potential function. Our ray-isosurface intersection test works efficiently by evaluating the metaballs potential field at only a few points along the ray. The use of an acceleration structure, such as a BVH, can make our intersection tests even faster by reducing the number of metaballs accounted for. However, a BVH needs specific attention for it to work with metaball rendering, which we detail in Section 4. As successfully done with triangles [EG07, SFD09], we show that a specific Surface Area Heuristic (SAH) and the fitting of the bounding boxes of each node can make the BVH more robust according to the density of the metaballs. Our GPU implementation shows that scenes composed of a large number of metaballs can be rendered at an interactive-to-real-time framerate.

2. Related work

2.1. Density function

A metaball i is a potential field centered at a point p_i called the center of the metaball. This potential field is often described by a density function f_i, which decreases as we move away from p_i. N metaballs define a single potential field f as the sum of the density function of each metaball. Visualizing metaballs actually looks for the isosurface defined by the equation:

$$f(x) = \sum_{i=0}^{N} f_i(x) - T = 0$$

for a chosen threshold value T. The original potential field is Blinn's blob [Bli82], for which the density function is defined as a Gaussian function, and thus has an infinite support. Other common density functions are piecewise polynomials of the form:

$$f_i(x) = \begin{cases} \left(1 - \frac{\|x-p_i\|^2}{R_i^2}\right)^{\alpha} & \text{if } \| x - p_i \| \leq R_i; \\ 0 & \text{otherwise.} \end{cases}$$

The larger the value for α, the smoother the surface. These density functions have a finite support, which allows skipping metaballs that are out of the range of any point x when computing $f(x)$. The sphere centered at p_i and of radius R_i is called the *bounding sphere* of metaball i. Our method is presented using this last potential field, but it handles any finite support density function.

2.2. Raytracing metaballs on the GPU

A number of papers have addressed the issue of rendering metaballs using the GPU. Iwasaki et al. [IDYN06] perform surface reconstruction of a particle simulation on GPU. However, since their method discretizes the surface, it might fail to render high frequency details such as thin splashes in fluid simulations. Van Kooten et al. [vKvdBT07] render metaballs on the GPU using point-based visualization of particles spread uniformly along the isosurface, but their method can miss small objects as well.

Loop and Blinn [LB06] ray cast piecewise algebraic surfaces defined by Bezier tetrahedra on the GPU using Bernstein polynomials. However their method is not really suited for a large number of metaballs as the cost of computing the coefficients of the polynomials at each vertex of the tetrahedra depends on the number of metaballs. More recently, Kanamori et al. [KSN08] efficiently ray cast a large number of metaballs on GPU using depth peeling and Bezier clipping [NN94], but since they rely on rasterization from the image, their method is limited to primary rays, thus restricting their rendering effects to the ones achievable in screen space (for instance, shadow maps or screen space ambient occlusion), contrary to our method which performs true ray tracing.

In order to render point clouds, Wald and Seidel [WS05] first reconstruct a potential field of finite support using a partition of unity, and then perform a kd-tree accelerated ray tracing of an isosurface on the CPU. Our method for rendering metaballs has similarities with their technique: we both use an acceleration structure built greedily over the bounding boxes of the primitives with the use of a SAH, and then try to reduce the size of its nodes in order to reduce the number of intersection tests. Their method for reducing the size of a node consists in slicing that node in thin layers, then checking which layers do not intersect the surface in order to remove them from the node. This last step is done in a Monte-Carlo way, by evaluating the potential field at several samples at the base of each layer. This technique increases drastically the construction time of the kd-tree as hundreds of samples are needed to safely remove a layer. This is not a problem in their context, since they consider static geometries only. The optimizations of our acceleration structure are efficient and enough to achieve interactive rendering of dynamic metaballs. In our context, our intersection test is also more accurate than the ray marching they use.

3. Intersection test

Numerous methods for ray-isosurface intersection have been studied. The Bezier clipping method, developed by Nishita and Nakamae. [NN94], is robust but requires to sort the intersections with the metaballs' bounding spheres along the ray. Sorting these intersections can be efficiently realized for primary rays, thanks to their coherency, by rasterizing the bounding spheres as did Kanamori et al. [KSN08].

Our method for intersecting the isosurface consists in two steps. First it looks for an interval $[a;b]$ along the ray containing only the first intersection. Then it uses the secant method [MKF*04, NMHW02] to narrow this interval until

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

sufficient precision is reached. The secant method always converges if there is only one intersection in $[a;b]$.

To find such a starting interval, we make the assumption that wherever the ray intersects the isosurface, the potential will be positive at at least one of the points p_{m_i}, corresponding to the projection of the center p_i of each metaball i on the ray. While this assumption is true for most rays, it can be false for very few rays that are nearly tangent to the surface. Fortunately, missing these tangential rays hardly causes any visual artifacts, and denser sets of metaballs create less silhouettes. We then set b as the first point p_{m_i} met by the ray at which the potential is positive and a as the origin of the ray. If $f(p_{m_i}) < 0$ for every i, we consider that our ray does not intersect anything, due to our above assumption.

Once the intersection point has been computed, the normal is found in a standard manner by computing the gradient of the potential field at that point.

4. BVH for rendering metaballs

In order to make the intersection computation more efficient and keep interactive the whole rendering process, we must build an acceleration structure that minimizes the number of metaballs for ray-metaball intersection tests.

A *Metaball Connected Set* (MCS) is defined as a set of metaballs creating a continuous surface (cf. Figure 2). As the metaballs move in the scene, the number and configuration of the MCSs are likely to vary at each frame.

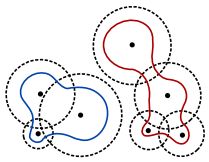

Figure 2: *A set of metaballs defining two MCSs in red and blue.*

A BVH is an efficient acceleration structure for many primitives, which can be built quickly either on CPU [Wal07] or GPU [LGS*09]. However, in our context, a classical BVH built to enclose the MCSs would not work well. Indeed, the metaballs usually define only a few MCSs, unless the metaballs are sparsely distributed, which is not the case for most fluid simulations. Therefore, such a BVH would result in a tree of low depth, whose nodes would each encompass a large number of metaballs.

Instead, we propose to compute the BVH on the metaballs' bounding spheres. However, such a structure does not contain all the necessary data to be able to perform ray-isosurface intersection tests within the nodes themselves, since the surface generated by the metaballs in a node might get modified by metaballs outside that node. We thus modified the BVH to add *split metaballs* to the nodes at the leaves.

The *split metaballs* of a node are the metaballs that do not belong to the node, but whose bounding spheres intersect the bounding sphere of at least one metaball from that node. One can see the split metaballs of a node as the minimal set of metaballs required to compute any intersection in that node. Thus, storing the indices of those split metaballs in each leaf makes the BVH fully functional and autonomous.

The BVH accelerates intersection tests by reducing the number of metaballs accounted for: when looking for an intersection in a leaf, we explore only the projections p_{m_i} of each metaball i from that leaf and use only the metaballs and split metaballs of the leaf to compute the value of the potential field at any point. Note that it is unnecessary to compute and look for an intersection at the projection p_{m_j} of all split metaballs j, as they belong to other leaves and will be explored when looking for an intersection in those leaves anyway.

4.1. Construction

The modified BVH is built as a classical BVH, from top to bottom. As in [Wal07], at each step of the construction, several ways of splitting the latest node built are browsed using binning. The best split position is then evaluated using a greedy SAH and its cost is compared to the cost of turning that node into a leaf. However, at each step of the algorithm, we keep track of the split metaballs of each node. Each time a node is split into two children, the split metaballs of each child are computed by looking both at the split metaballs of their parent and the metaballs of the other child (cf. Algorithm 1).

Algorithm 1 Construction of a BVH for metaballs

```
1:  BUILDNODE(nodeMballs, splitMballs, nodeBbox) {
2:  find best splitting plane s for nodeBbox using binning
3:  leftChildMballs, rightChildMballs ← ∅
4:  leftChildBbox, rightChildBbox ← emptyBbox
5:  for each i in nodeMballs do
6:      if p_i is at the left of s then
7:          leftChildMballs ← leftChildMballs ∪ {i}
8:          leftChildBbox ← leftChildBbox ∪ bbox(i)
9:      else
10:         rightChildMballs ← rightChildMballs ∪ {i}
11:         rightChildBbox ← rightChildBbox ∪ bbox(i)
12:     end if
13: end for
14: leftSplitMballs ← ∅
15: for each i in splitMballs ∪ rightChildMballs do
16:     if i overlaps the bounding sphere of any j ∈ leftChildMballs then
17:         leftSplitMballs ← leftSplitMballs ∪ {i}
18:     end if
19: end for
20: BUILDNODE(leftChildMballs, leftSplitMballs, leftChildBbox)
21: execute the instructions 14 – 20, but for the right node }
```

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

5. Fitted-BVH

As the leaves of the modified BVH can overlap each other, the same intersection can be computed several times during ray traversal. This becomes an issue when dealing with dense sets of metaballs as the leaves grow bigger, therefore increasing the cost of computing a ray-isosurface intersection.

The Fitted-BVH (FBVH) reduces the number of redundant intersection tests during the traversal of the structure. When two nodes A and B of the modified BVH overlap each other, most of the metaballs from A whose bounding spheres intersect the overlapping volume $A \cap B$ are duplicated as split metaballs in B. Therefore, during ray traversal, the surface generated by those metaballs can be intersected several times as stated earlier.

Like a BVH, an FBVH is a tree in which the bounding boxes of two child nodes define a partition of the surface encompassed by their parent node. However, contrary to a BVH, the two child nodes cannot intersect each other (cf. Figure 3). Therefore, if we explore the leaves in order of traversal, we can stop as soon as an intersection is found, thus minimizing any redundant test.

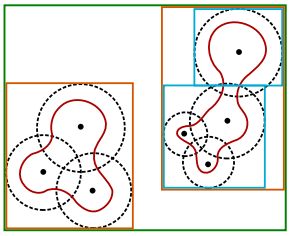

Figure 3: *An FBVH is a surface partition tree where two child nodes cannot overlap each other.*

Since the bounding boxes of two child nodes cannot intersect each other anymore, they no longer correspond to the bounding boxes of the metaballs they contain. Instead, they determine a region of space that contains a portion of the surface generated by the metaballs and split metaballs of the nodes. Therefore, there is not much difference between the metaballs and the split metaballs of a node: the union of both determines the set of metaballs whose bounding spheres intersect the node. However, we will continue to separate these two sets as it makes the construction of the FBVH more convenient.

Actually, the main difference with a kd-tree is the presence of a bounding box for each node, which enables a tight encompassing of the metaballs bounding spheres when they are sparse enough. This is of prime importance here as the cost of intersecting a leaf is high (cf. Section 5.1.1). Such an encompassing can be done with a kd-tree using techniques like empty space cutoff implemented using our fitting techniques (cf. Section 5.1.2) and our optimizations (cf. Section 5.3). However it would require additional nodes (up to 5 additional nodes to remove all empty space around the bounding box of a node) that would create additional branchings during the traversal of the structure. Those branchings are detrimental when using SIMD architectures such as a GPU, since they may reduce instruction coherency. This makes us believe that our FBVH should be a little faster than such a kd-tree in this context, even though we did not actually implement one. The difference of performance is probably small anyway as the cost of intersecting a leaf is much higher than the one of the ray traversal.

5.1. Construction

5.1.1. Surface Area Heuristic for metaballs

SAH is essential to build the acceleration structure in a fast, greedy way. However, it has to be carefully designed to ensure good performance during ray traversal. The usual cost considered when splitting a node N into two nodes A and B is

$$C_{\text{split}}(A,B) = \rho_A \times C_A + \rho_B \times C_B$$

where C_i is the cost of computing an intersection in node i, and ρ_i is the probability of having a ray intersect node i knowing that its parent node has been intersected. $C_{\text{split}}(A,B)$ is compared to the cost of leaving N as a leaf.

ρ_i is estimated as the ratio A_i/A_N of the surface area of the bounding box of node i over the surface area of the bounding box of its parent node N. Usually, the cost of intersecting a surface generated by a primitive is the same in each node, so C_i is in $\mathcal{O}(n)$ where n is the number of primitives in node i (we can even set $C_i = n$ if all primitives are of the same type). This is not the case for metaballs though, as the cost of evaluating the potential field depends on the number of metaballs in the current leaf.

When looking for an intersection in a leaf, we compute the projections on the ray of every center of the leaf's metaballs. The potential field is evaluated at those projection points, and each of these evaluations is done in $\mathcal{O}(n)$ operations where n is the number of metaballs in the leaf. Thus this base interval finding step is done in $\mathcal{O}(n^2)$ operations.

Starting from this analysis, our cost function should be $C_i = n^2 + \lambda \times n$ where n^2 corresponds to the base interval finding and the $\lambda \times n$ to the secant method. The value for constant λ is difficult to determine, as it depends on the number of iterations needed by the secant method to converge. However, it is clear that n^2 is the dominant term for leaves composed of many metaballs (which is the case for dense sets of metaballs). Thus we can as well set $\lambda = 0$ in those cases. In our experiments, neglecting $\lambda \times n$ and setting $C_i = n^2$ did not have much impact in terms of rendering speed in the case of sparse metaballs either.

5.1.2. Algorithm

To build the FBVH, we proceed as with the modified BVH. However, when splitting a node, we compute a first bounding

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

box for each child by splitting the node's bounding box at the split plane (cf. Figure 4 (a)). The metaballs from the node are distributed to either the left or the right child according to the splitting plane. Then, any metaball from the node's split metaballs or the right child's metaballs that overlaps the left child node is duplicated in its split metaballs, and the same is done for the right child node (cf. Algorithm 2).

We then try to fit the bounding box of each child node to its primitives (hence the *fitted*-BVH). This step is crucial to achieve good performance as the cost of intersecting a leaf is high ($\mathcal{O}(n^2)$, see Section 5.1.1). In order to do so, we compute the bounding box of the metaballs and split metaballs for each child (cf. Figure 4 (b)). Each resulting bounding box is then intersected with its previous bounding box (cf. Figure 4 (c)).

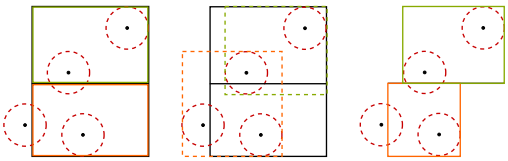

Figure 4: *When splitting a node, the bounding boxes of the metaballs and split metaballs of each child are computed (b). Intersecting them with the former bounding boxes of each child (a) gives their new bounding boxes (c).*

Algorithm 2 Construction of an FBVH for metaballs

```
1:  BUILDNODE(nodeMballs, splitMballs, nodeBbox) {
2:  find best splitting plane s for nodeBbox using binning
3:  leftChildMballs ← ∅, rightChildMballs ← ∅
4:  leftChildBbox ← nodeBbox.splitLeftAlong(s)
5:  rightChildBbox ← nodeBbox.splitRightAlong(s)
6:  leftMballBbox ← emptyBbox
7:  rightMballBbox ← emptyBbox
8:  for each i in nodeMballs do
9:      distribute i to either the left or right child node and
        update left and right metaball bounding boxes accord-
        ingly as in Algorithm 1
10: end for
11: leftSplitMballs ← ∅
12: for each i in splitMballs ∪ rightChildMballs do
13:     if i overlaps leftChildBbox then
14:         leftSplitMballs ← leftSplitMballs ∪ {i}
15:         leftMballBbox ← leftMballBbox ∪ bbox(i)
16:     end if
17: end for
18: leftChildBbox ← leftChildBbox ∩ leftMballBbox
19: BUILDNODE(leftChildMballs, leftSplitMballs, leftChildBbox)
20: execute the instructions 11 – 19, but for the right node }
```

5.2. Ray traversal

The ray traversal is not much different from the one of a BVH. When a ray intersects a node, we load the bounding boxes of both child nodes and compute which one is intersected first (if any). The first one to be intersected is then traversed first. As soon as an intersection is found, we can stop the traversal since other leaves can only result in an intersection farther away.

The intersection computation in a leaf is a little different than when using a BVH. Since the split metaballs of the current leaf are not fully encompassed within another leaf, their projection along the ray has to be computed and checked as well when looking for an interval containing the first root of the potential field.

5.3. Optimizations

The bounding boxes of the nodes, built from the bounding spheres of their metaballs, might not fit as tightly to the surface they encompass after the fitting step described in Section 5.1.2. We propose a modification to the computation of the bounding box of a set of metaballs that leads to better fitted nodes. We also observed that some of the split metaballs of the nodes do not contribute to the surface encompassed by that node and propose a technique for discarding them.

5.3.1. Bounding box computation

When computing the bounding box of some metaballs, we usually compute the smallest bounding box which encompasses every metaballs' bounding sphere. However, the surface generated by the metaballs might not be so close from the bounding spheres (cf. Figure 5). Thus it is possible to compute a tighter bounding box, which would ensure better performance by reducing the number of intersection tests in each leaf.

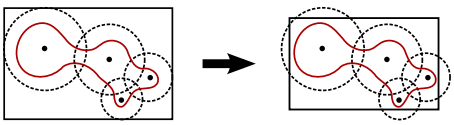

Figure 5: *The bounding box encompassing the bounding spheres of the metaballs might not be the tightest one around the surface itself.*

A way to do so is to consider that the largest sphere generated by the metaballs is obtained when every metaball of the leaf shares the same center, creating a sphere of radius r such that

$$\sum_{i=1}^{n} f_i(r) = T \quad \text{where} \quad f_i(r) = \left(1 - \frac{r^2}{R_i^2}\right)^{\alpha}$$

in our case. This sum is bounded by $n \times f_K(r)$ where K is the metaball with the largest bounding sphere (of radius R_K)

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

in the node. Thanks to this, we can easily compute an upper bound for r:

$$r\max = R_K \times \sqrt{1 - \left(\frac{T}{n}\right)^{\frac{1}{\alpha}}}.$$

Therefore, $r_i = \min(R_i, r\max)$ can be used as the radius of metaball i's bounding sphere when computing the bounding box of the metaballs.

5.3.2. Discarding non contributing split metaballs

The split metaballs of a node have bounding spheres overlapping their node's bounding box. However, some of the split metaballs might belong to an MCS which does not overlap the node, i.e., they do not contribute to any surface in that node (cf. Figure 6). Therefore, storing them as split metaballs is useless and harmful, as they increase computation time for the intersection tests in the node.

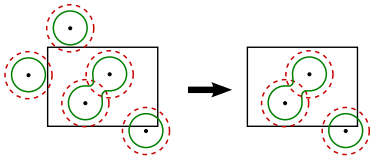

Figure 6: *Some of the split metaballs whose bounding sphere (in red) intersects a node do not generate a surface (in green) intersecting that node. Storing them in that node is useless. Note that any metaball participating to the surface in a node has to be stored in that node.*

Identifying such split metaballs that can thus be removed is a difficult problem. However, some conservative tests can be applied. Given split metaball i, we test:

- that the sphere centered at p_i and of radius $r\max$ (see above) does not overlap the node's bounding box. This ensures the MCS containing metaball i does not intersect the node.
- that for any other metaball j of the node,

$$\| p_i - p_j \| \geq R_i + r\max.$$

  This ensures metaball i does not interfere with the surface generated by any other metaball j of the node.

If both those tests succeed, the metaball can be safely discarded from the split metaballs.

6. Results

We wrote our implementation in C++ and CUDA. We perform the FBVH construction on the CPU, while ray traversals and intersection computations are done on the GPU. This allows us to compute the FBVH of the next frame on the CPU while the current frame is being rendered on the GPU, thus hiding its computation cost. In order to benefit from cache for data access, metaballs and node data are stored in the GPU texture memory. Traversal on the GPU is done using a stack stored in local memory. Animation of metaballs is done on the GPU, then transferred to the host memory. We use streaming so that memory transfers between CPU and GPU can be hidden with computations (cf. Figure 7).

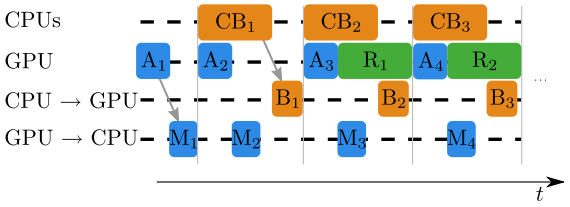

Figure 7: *Runtime scheme. Animation of the metaballs (A_i, in blue) is done on the GPU, then transferred to the host (M_i). The host computes the BVH or FBVH (CB_i, in orange), then transfers it to the device memory (B_i). Rendering (R_i, in green) is done on the GPU while the CPU computes the acceleration structure for the next frame. Most of the memory transfers are hidden thanks to streaming.*

Our configuration test is a PC with an intel Core 2 E6600 (2.66 GHz) and an Nvidia GTX 280. We compared the FBVH to the BVH in various scenes (cf. Table 1 and Figure 8). In most scenes, we observed a significant gain of performance over the BVH, especially with those scenes containing dense sets of metaballs. In these cases, such as the Pool scene, the FBVH can lead to more than 3 times faster renderings. The difference of performance with the BVH decreases with the density of metaballs, but as long as the metaballs are not too sparse, the FBVH still outperforms the BVH. It is also worth considering that while the times of construction of both structures are of the same magnitude, the FBVH is usually a little faster to build. This can seem surprising, as it could have been expected that fitting the bounding boxes of each node would result in a larger amount of computations. Actually, even if the bounding boxes of the nodes have a few more steps in their construction, their reduced sizes lead to fewer split metaballs to track during the construction of the structure, hence the speedup.

Considering equivalent parameter values (such as metaball radius, implicit surface polynomial degree, etc.) and hardware as those used in [KSN08], we estimate that our raytracer (acting as a ray caster in this context) would be about 30% slower than their rasterization-based method. This is a good performance for a pure raytracer that, by nature, is not limited to primary ray effects. Moreover, in our implementation, the cost of additional non-primary rays can be quite low (cf. Table 2).

We measured the average number of node traversals, intersection tests, and the average number of metaballs considered in these intersection tests per ray (cf. Table 3). As one can see, the FBVH does a clearly better job at culling unnecessary nodes and metaballs.

The effect of using the adapted SAH n^2 instead of the

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

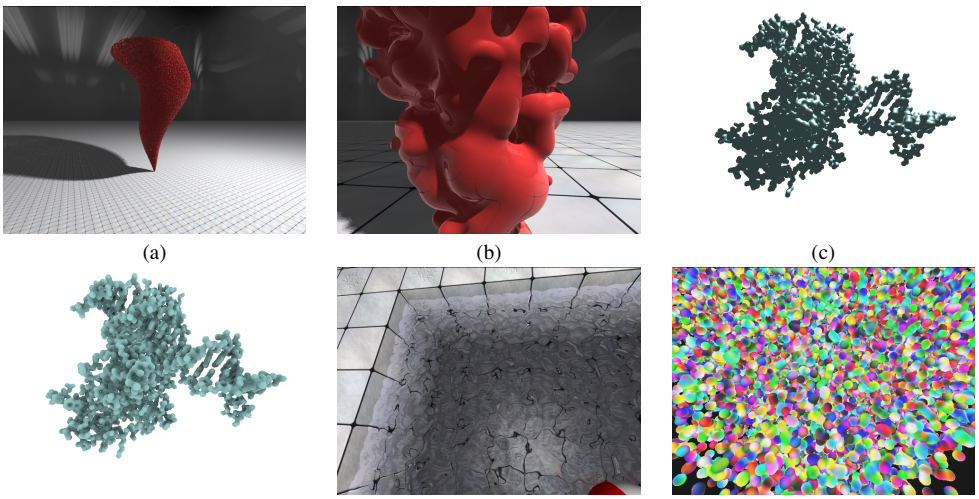

Figure 8: *Screenshots of the different test scenes: (a,b) the Tornado scene, rendered using three levels of reflection; (c) the Molecule 1 scene, rendered with Phong shading; (d) the Molecule 2 scene, rendered with ambient occlusion using 64 shadow rays per pixel; (e) the Pool scene, rendered using Fresnel reflectance; (f) the Scattered metaballs scene (pure ray casting). All these scenes are rendered at a* 640×480 *resolution in our tests.*

Scene	number of metaballs	modified BVH				FBVH			
		avg time of construction	fps			avg time of construction	fps		
			min	max	avg		min	max	avg
Scattered metaballs	30,000	101 ms	4.6	12.4	7.4	79 ms	4.9	12.5	9.4
	100,000	160 ms	1.8	3.9	3.3	150 ms	2.3	9.1	4.3
Tornado	4,000	8.0 ms	3.6	5.4	4.6	5.5 ms	4	14.5	11
	128,000	500 ms	0.6	1.1	1.0	290 ms	1.9	2.6	2.4
Molecule 1	5,440	8.9 ms	3.2	8.6	5.8	9.9 ms	7.5	24.1	17.7
Molecule 2	5,440	8.9 ms	0.5	0.7	0.6	9.9 ms	1.2	1.4	1.3
Pool	10,000	41 ms	1.5	2.3	1.9	40 ms	4.1	11.0	7.1

Table 1: *Rendering time in fps of various scenes containing metaballs using an acceleration structure (cf. Figure 8). The construction of the acceleration structure is completely hidden by the rendering in most cases.*

Scene	Effects (avg fps)			
	Ray cast	Hard shadows	Hard shadows + 1 level of reflection	Hard Shadows + 3 levels of reflection
Molecule	22.3	17.7	12.7	8.2
Tornado	5.7	4.9	3.4	2.5

Table 2: *Performance of our algorithm with secondary effects.*

	Number of traversals	Number of intersection tests	No. metaballs considered
BVH	9.3	1.2	83.5
FBVH	6.2	0.5	35.1

Table 3: *Average number of traversals, intersection tests, and metaballs considered per ray on the Tornado scene.*

standard one n provides much better performance overall. In our tests, the Pool scene gains 40% more fps using the adapted SAH (5.1 fps with the standard SAH, 7.1 fps with ours). Our SAH leads to a tree that is better suited for our intersection test than the standard one, hence the better performance.

We also measured the effects of fitting the bounding boxes of the nodes to their metaballs, as explained in Section 5.1.2. To this effect, we omitted the fitting step during the construction of the FBVH so that each node's bounding box was just the result of splitting the bounding box of its parent. As one can see in Table 4, the number of intersection tests avoided by the fitting step clearly leads to better performance. This gain of performance increases with the density of metaballs, since the cost of computing an intersection in a leaf increases with the number of metaballs in that leaf. Therefore, it is a

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.

worthy improvement since it does not impact the FBVH construction cost too much.

Scene	Without the fitting step		With the fitting step	
	build time	avg fps	build time	avg fps
Molecule 1	8.7 ms	12.7	9.2 ms	14.6
Pool	35 ms	3.2	37 ms	6.5

Table 4: *Benefits of fitting the nodes' bounding boxes after a split during the construction of the FBVH.*

Scene	Optimizations (avg fps)			
	None	Tighter Bbox	Discarding metaballs	Both
Molecule 1	14.6	15.3	16.7	17.7
Pool	6.5	6.7	6.8	7.1

Table 5: *Benefits of the optimizations.*

The optimizations presented in Section 5.3 bring a noticeable performance improvement (cf. Table 5), but their impact decreases as the density of the metaballs increases. This was to be expected, as the value of $r_{\max}$ increases with the number of metaballs overlapping each node. Therefore, it might be preferable to skip these optimizations in the leaves containing more than a dozen metaballs, as their cost increases with the number of metaballs (especially the discarding of the split metaballs), and since they can drastically increase the time of construction of the FBVH.

7. Conclusion and future work

We have presented a new acceleration structure for the rendering of metaballs, that supports advanced secondary effects. The resulting speedup enables the visualization of a large number of metaballs at interactive-to-real-time framerates on the GPU. The FBVH provides a consequent speedup compared to the BVH in the case of dense sets of metaballs and can be built quickly on the CPU using our SAH.

However, for extremely dense sets of a large number of metaballs, the construction time of the FBVH might become an issue, as there is a lot of split metaballs to track. In those cases, for the first levels of the tree, we do not use binning to find the best splitting plane for each node but simply split the nodes along their longest axis. This does not affect the quality of the tree too much and provides a good speedup.

Another issue is the time of computing a ray-isosurface intersection in leaves that contain many metaballs. In these cases, the leaves usually contain a single MCS whose surface is very close to the leaves' bounding boxes. We will work on approximations to accelerate intersection with the isosurface in these cases.

References

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

[Blo97] BLOOMENTHAL J.: *Introduction to Implicit Surfaces*. Morgan Kaufmann, August 1997.

[EG07] ERNST M., GREINER G.: Early split clipping for bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (2007), pp. 73–78.

[Flo08] FLOREZ J.: *Improvements In The Ray Tracing Of Implicit Surfaces Based On Interval Arithmetic*. Ph.d. thesis, Department of Electronics, Computer Science and Control University of Girona, November 2008.

[Har93] HART J. C.: Ray tracing implicit surfaces. In *SIGGRAPH 93 Course Notes: Design, Visualization and Animation of Implicit Surfaces* (1993), pp. 1–16.

[IDYN06] IWASAKI K., DOBASHI Y., YOSHIMOTO F., NISHITA T.: Real-time rendering of point based water surfaces. In *Computer Graphics International 2006* (June 2006), pp. 102–114.

[KHK*09] KNOLL A., HIJAZI Y., KENSLER A., SCHOTT M., HANSEN C., HAGEN H.: Fast Ray Tracing of Arbitrary Implicit Surfaces with Interval and Affine Arithmetic. In *Computer Graphics Forum 28* (2009), no. 1, pp. 26–40.

[KSN08] KANAMORI Y., SZEGO Z., NISHITA T.: GPU-based fast ray casting for a large number of metaballs. *Computer Graphics Forum (Proc. of Eurographics 2008) 27*, 3 (2008), 351–360.

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

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on gpus. *Comput. Graph. Forum 28*, 2 (2009), 375–384.

[MKF*04] MARMITT G., KLEER A., FRIEDRICH H., WALD I., SLUSALLEK P.: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Vision, modeling, and visualization 2004 (VMV-04)* (2004), pp. 429–435.

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. In *VISSYM '02: Proceedings of the symposium on Data Visualisation* (2002), Eurographics Association, pp. 77–86.

[NN94] NISHITA T., NAKAMAE E.: A method for displaying metaballs by using bézier clipping. *Comput. Graph. Forum (Proc. of Eurographics '94) 13*, 3 (1994), 271–280.

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), ACM, pp. 7–13.

[Ura06] URALSKY Y.: Dx10: Practical metaballs and implicit surfaces. In *Game Developers Conference* (2006).

[vKvdBT07] VAN KOOTEN K., VAN DEN BERGEN G., TELEA A.: Point-based visualization of metaballs on a gpu. In *GPU GEMS 3*, Nguyen H., (Ed.). Addison-Wesley, 2007, ch. 7.

[Wal07] WALD I.: On Fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (2007).

[WS05] WALD I., SEIDEL H.-P.: Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics* (2005).

[WT90] WYVILL G., TROTMAN A.: Ray-tracing soft objects. In *CG International '90* (1990), pp. 469–476.

© 2010 The Author(s)
Journal compilation © 2010 The Eurographics Association and Blackwell Publishing Ltd.