Interactive and Scalable Ray-Casting of Metaballs on the GPU

Jon McCaffrey

Advisor: Dr. Norman Badler

University of Pennsylvania

ABSTRACT

Metaballs are a useful technique to model blobby surfaces. Because metaballs define an implicit surface, rendering is difficult. We seek to render the surface directly via ray-casting, to precisely locate the isosurfaces and preserve the concise representation of the surface. To perform this rendering at interactive rates on dynamic data, we will use acceleration structures for fast surface calculation, and will implement it in the CUDA framework using Nvidia GPU's.

Project Blog: http://csgrendering.blogspot.com/

1. INTRODUCTION

Metaballs are a modeling technique appropriate for blobby surfaces such as liquid droplets, covalent bonds. A surface is created by a summation of density functions centered at points. Each function is roughly Gaussian in shape, and for efficiency usually has finite support, meaning that it is nonzero only in some limited area. Overlapping density functions combine additively. The surface is defined as the isosurface of the density function at some given threshold. Thus, as two metaballs move together, their density functions overlap and sum together until a bridge is formed.

While metaballs are an intuitive modeling method for these types of surfaces, rendering is difficult because the metaballs define an implicit surface. Various approaches to rendering include tessellation of the surface into polygons and voxelization.

However, these methods introduce an additional level of sampling error, and also destroy the original concise representation of the surface. Each metaball might only require a position, amplitude, and radius. However, to smoothly render a surface made of thousands of metaballs could require hundreds of thousands of polygons. For interactive rendering on modern processors, where compute power is plentiful but memory bandwidth is constrained, this is particularly a problem. We thus seek to directly render the isosurfaces via numeric root-finding methods. This process will be accelerated using Nvidia GPUs.

1.1 Design Goals

The goal will be interactive visualization of simulation data. These could be used for attractive presentations of fluid or particle data. The system will generate smooth and accurate renderings at interactive rates.

1.2 Projects Features and Functionality

Our system will take as input frames of animation consisting of “particle soup”. That is, an unstructured array of metaball positions, radii, and amplitudes. We will also take as secondary input rendering parameters such as camera position and orientation, clipping planes, resolution, and sampling, and shading parameters such as shading model and corresponding colors or maps.

The primary output will be a image buffer containing a rendered frame of animation, suitable for immediate display via OpenGL or saving to disk. Other outputs might include additional information buffers such as depth or surface normals.

The primary thing of note here would be we plan to take unstructured data per frame, and hide any acceleration structures internally. Because our target application of particles in motion has no rigid organization and may change rapidly and unpredictably, we saw little benefit in maintaining frame-to-frame coherence. Instead, we are aiming for fully general acceleration, starting from scratch each frame.

2. RELATED WORK

Interactive ray tracing has become a recent field of interest, with interactive ray tracing of surfaces, as opposed to triangles, as a subfield of that. Research has used both the CPU and GPU as platforms. The different platforms share some connections, but also differences due to the varying architectures. With GPU work, a distinction is also draw on whether the authors hijacked the rendering pipeline using shading languages or used a higher-level framework such as CUDA.

2.1 Interactive Ray Tracing

Several innovations from this field are applicable towards our work. These include research on acceleration structures and on optimizing ray tracing to exploit coherency and architectural details.

[WSBW01] discusses packet-based raytracing. This technique exploits coherence between neighboring rays and maps well to SIMD architectures. They allow optimizations, memory accesses, and culling to be amortized across packets of rays.

Rendering of metaballs can be accelerated by building an accelerating structure using bounding spheres of the metaballs as the primitives. [WIKK06] discusses the uniform grid data structure to accelerate animated scenes. While the provided acceleration is generally worse than adaptive data structures, the uniform grid is simple and efficient to construct, even interactively. It is also well-suited to dynamic particle data, which may exhibit little exploitable hierarchy or structure. [IDC09] presents an simple and fast algorithm to construct uniform grids on modern GPUs.

Other acceleration structures are possible to construct and traverse on the GPU. [ZHWG08] performs construction of Kd-tree's for ray tracing acceleration on the GPU, however with considerable complexity and a high runtime cost. [GPSS07] develop a fast BVH traversal algorithm targeted towards Nvidia GPU's. It maintains a traversal stack per-packet.

2.2 Interactive Rendering of Metaballs

Several researchers have approached rendering of metaballs on the GPU.

[KSN08] uses ray casting on the GPU to evaluate the isosurface per-pixel. They find affecting metaballs along each ray by performing depth-peeling of the bounding spheres, then intersect via root-finding with Bezier clipping.

[GPBP09] uses BVH's to ray-trace metaballs with several bounces on the GPU at interactive rates. However, they build the BVH on the CPU and then download it to the GPU, a potential bottleneck. They describe problems with efficiency due to dense groups of metaballs.

2.3 Coherent Ray Tracing

If full ray-tracing is employed, as opposed to ray-casting, steps have to be taken to rebuild coherent packets. [MMA07] use heuristics to rebuild packets with rays generated across multiple levels of ray tracing. This requires significant global communication, and the balancing of the communication overhead vs the gained performance.

3. PROJECT PROPOSAL

We seek to directly render isosurfaces via ray casting. Given an unstructured set of metaballs each frame, we will construct an acceleration stucture, then traverse it and intersect with leaves until an intersection is found.

3.1 Acceleration Structure

We thinking that the best choice of acceleration structure for this problem is the uniform grid. Unlike adaptive structures, it is simple and efficient to construct and inexpensive to traverse. While in general it provides inferior culling, it is well-suited to unstructured and scattered data such as particle simulations. All metaballs that overlap a grid cell at all need to be included so their influence can be taken into account. This is an efficiency factor that pushes towards larger cells, to minimize duplication.

Once the acceleration structure is constructed, we will traverse it via packets of rays. The packets of rays then visit a series of cells containing metaballs in rough front to back order. [WIKK*06] traverses the grid by “slices” in the dominant direction of the packet, so while intersections may be out of order within a slice, between slices they are ordered.

3.2 Intersection

Each cell we traverse, we intersect all rays with the metaballs present in the cell to find the best intersection. We take the closest intersection for each metaball over each slice if there is more than one. Once all rays in a packet have terminated, the packet is finished. Once all packets are complete, we move to shading.

3.4 Shading and Filtering

Our initial version will use a basic shading model, with additional configuration added with time permitting. Blinn-phong shading with hard shadows and environment mapping will probably suffice for the initial implementation. One possibility for programmable shading if recursive ray tracing is not used is to use deferred shading with a shading language such as Cg.

Filtering and antialiasing, if employed, can occur in a final blend after shading. Any jittering or displacement of samples must occur during ray generation.

3.5 Implementation as Kernels

We plan to implement this renderer using the CUDA framework on Nvidia GPU's. We plan to divide the rendering into a number of separable kernels. To simplify implementation, we have separated the components of the renderer as much as possible. Each will block until all threads have completed. It may be possible to get better performance and suffer less from divergent behavior if we used uber-kernels that incorporate all parts of the rendering and dynamic load balancing techniques such as work queues with persistent threads. However, they would complicate the implementation significantly and would probably not be a good idea for the first version.

The first is a ray generation kernel, which takes in the rendering parameters and generates a grid of rays sampling the scene that are grouped into packets. A separate kernel for ray generation allows effects like random sampling to be plugged in as modular components. Antialiasing if used is a combination between ray generation and an additional filtering pass after shading.

After generation we begin a cycle of traversal and intersection. In general, we will use a CUDA thread per ray, and a block of threads per packet. The traversal kernel advances each packet through the grid slice by slice until it encounters a non-empty slice, at which point it examines each intersected cell in the slice. Frustum-culling per packet can be used in an initial pass for large granularity culling. For this culling, each packet is represented by a thread.

The intersection kernel processes each packet of rays vs. the cell that the packet is currently visiting.

The core intersection kernel is a key component for the performance of the overall system, and we are considering a number of different implementations. All implementations use one CUDA thread per ray, and a block of threads for one packet. Memory access to the metaballs for a cell is amortized across the block since all rays in a packet traverse the cell together.

The first routine we developed is probably the worst. Its is based on the methods described in [NN03] and It involves segmenting the ray along its length by the metaballs affecting each segment. A Bezier approximation for each segment is then generated, and intersected with via robust Bezier clipping. A key problem with this approach is that the searching and sorting involved are highly divergent between rays. Also, the segmentation is dependent on not only the number but the arrangement of the objects in the scene, making the runtime of this technique dependent on the depth complexity of the scene. This technique could overrun fixed size buffers and fail in the case of complex arrangements. It has the advantage that is processes intersections in front-to-back order, allowing for early termination, and that it uses a numerically robust root-finding technique.

The second routine developed is probably the most novel. It involves choosing a limited set of basis functions to represent the density along each ray. For each metaball, its density function along the ray is found and projected onto the basis via a linear transformation in function space. The limited set of basis functions means that many coefficients have to be dropped. The coefficients for each metaball are summed to form a vector of coefficients that represents approximately the density function along the ray. Intersection is then performed against this function.

There are a number of mathematical unknowns behind this routine. The key choices are the choice of basis and the choice of density function. The basis must represent the range of possible density functions well with a small number of coefficients. The density functions and the basis must be chosen carefully so that projection can be performed elegantly and efficiently (general projection onto a function involves the integral of the product along the length, to be avoided at all cost). Finally, the final basis function must be efficient to intersect against (ie not Fourier series).

In terms of precise implementations, this approach has several advantages in that it requires constant memory per ray, to store the vector of coefficients, and time complexity in the intersection routine that depends on the complexity of the density function, not directly on the number of intersected particles. If a basis with finite local support is used, such as Bsplines, it also allows the ray to be processed in front-to-back order allowing for early termination. A downside is that per-ray constant memory cost is significant, to store the necessary coefficient vector. Half-precision could probably be used to save space.

The third approach is the simplest and most memory efficient. It also requires the most computation. An iterative root-finding technique such as Newton-Raphson or the secant method is used, and to evaluate the function/its derivative when needed, each density function is evaluated and accumulated. This approach needs only a small constant working set of memory, just enough to hold the current and previous iteration of values, and significantly less than the method above.

A disadvantage to this approach is that all metaballs for a cell must remain in shared memory for the entire procedure, unless they are loaded from global memory each time. The second method can stream the metaballs through.

All of these approaches can be optimized for sparse cells by using bounding spheres. Any rays that miss the bounding spheres of all metaballs can be immediately discarded. If any rays hit a single metaball alone, an intersection test can be performed against the simple smaller sphere than a metaball forms alone.

Once all rays in all kernels have intersected a metaball, we move to a shading kernel. Information necessary for shading such as surface normals is passed on from the intersection kernel. Per-particle attributes such as temperature or velocity can also be interpolated using percentage contributation towards the density function as a blending weight.

For our initial implementation, shading will probably be performed in a hard-coded CUDA kernel. A promising option for the final implementation is deferred shading using Cg.

The final result of this rendering will be an OpenGl buffer. This can be copied back to main memory for storage, displayed directly, or composited with a rendered polygonal scene using depth maps and alpha blending.

3.6 Foreseen Challenges

Divergent rays and packets are a dangerous problem since they could delay the entire rendering.

At the moment we plan to perform ray-casting, rather than full ray tracing, with secondary bounces only for shadowing. A main reason for this is that secondary rays in general are less coherent that primary rays, and may form much more divergent packets.

The behavior of all three given intersection kernels near very dense regions of space, such as those that could occur in fluid simulations, is worrisome. A very tight adaptive structure, such as an aggressively fit Kd-tree, could alleviate this problem. Even that approach doesn't solve the problem of many many overlapping metaballs.

3.7 Target Platforms

We will implement this using the CUDA framework on Nvidia GPU's, with OpenGL for display and C++ for client-side wrappers.

3.8 Evaluation Criteria

This work can be evaluated on a number of dimensions compared to other work. These include quality, performance, and scalability of rendering, as compared to other implementations on both the GPU and CPU.

Quality of rendering is mostly relevant as compared to completely different approaches to rendering metaballs, including image-space methods, tessellation, or point-based methods. Ideally we will more precisely capture curvature and fine detail.

Performance and scalability can be best measured as frame rate vs. number of particles at several different resolutions, and with different shading complexities on sample animation sequences. Multiple resolutions/shading models are important because packets cause runtime cost to scale sublinearly with the number of rays cast due to supersampling or increased resolution. This is because increased ray density leads to more coherent packets and more efficient traversals.

These performance curves should be compared vs the results in the best known CPU implementations as well as the comparable results in [GPBP09] or [KSN08].

An benefit of our design is that the traversal and intersection kernels can be separated, tested, and evaluated quite independently. The traversal kernel can be plugged into a “dummy” intersection kernel which merely intersects with bounding spheres to make sure that the acceleration structure is being correctly traversed. The intersection kernel meanwhile can be tested by casting all rays into and rendering the results merely from one cell with several metaballs.

4. RESEARCH TIMELINE

We would like to submit the research for publication by April 1st. This means that, if we give a minimum of two weeks to write a paper and collect results. This means that the implementation must absolutely be finished by March 15th. We want to have all functionality in place and working by the Alpha Review on March 1st. This gives us two weeks to optimize and bug-fix before the drop-dead completion date. If the project should be mostly complete by March 1st and has an estimated time of 130 hours, then in the intervening 6 weeks we'll need to put in roughly 15-20 hrs/week.

Please see the attached Gantt chart for a more exact decomposition of tasks.

Project Milestone Report (Alpha Version)

- All projected functionality included (no new features beyond here)
- Renders of particles in animation
- Rough C++ wrapper

Project Final Deliverables

- All projected functionality battle-tested
- Performance bottlenecks isolated and fixed
- Performance results under various loads
- Renders with varying shading models of varying particle data
- Improved wrappers for use as a visualization library

Project Future Tasks

- Add secondary bounces for true reflection and refraction.
- Explore different acceleration structures and real-time construction algorithms
- Add full-featured material management
- Use persistent threads with a work queue for dynamic load balancing
- Use stream filtering techniques to rebuild packets out of multiple levels of the ray hierarchy
- Better interface with traditional rasterization pipeline, allowing ray-cast metaballs to be embedded in traditionally rendered polygonal scenes

---

5. Method

6. RESULTS

7. CONCLUSIONS and FUTURE WORK

APPENDIX

A. Questions

Very dense overlapping regions remain a problem for all of our intersection kernels. A special way to handle such regions could make our performance more robust.

One possibility would be detecting such preprocessing. Highly dense regions could then be represented with a more concise description; for example, very close metaballs could be merged together and their densities added. However, this could introduce popping between frames, as this alternative description flickers on and off.

Should we target other applications besides particle/fluid/molecular visualization, for example surface modeling?

What shading features do you place priority on for high-quality pleasing imagery? (Does not necessarily need to be “realistic”)

Do you have any different ideas for intersection kernels or ideas to make option 2 feasible?

References

[GPBP09] Gourmel O., Pajot A., Barthe L. Paulin M. : BVH for Efficient Raytracing of Dynamic Metaballs. SIGGRAPH 2009

[GPSS07] Gunther J., Popov S., Seidel H., Slusallek P.,: Realtime Ray Tracing on GPU with BVH-based Packet Traversal

[IDC09] Ivson P., Duarte L. Celes W.: GPU-Accelerated Uniform Grid Construction for Ray Tracing Dynamic Scenes.

[KSN08] Kanamori Y., Szego Z., Nishita T.: GPU-Based Fast Ray Casting for a Large Number of Metaballs. EUROGRAPHICS 2008 Volume 27, Number 2

[MMA07] Mansson E., Munkberg J., Akenine-Moller T.: Deep Coherent Ray Tracing. Interactive Ray Tracing 2007

[NN03] Nishita T., Nakamae E.: A Method for Displaying Metaballs by Using Bezier Clipping. Computer Graphics Forum, Volume 13 Issue 3, Pages 271-280

[WIKK*06] Wald I., Ize T., Kensler A., Knoll A.: Ray Tracing Animated Scenes Using Coherent Grid Traversal. ACM TOG Vol 25 issue 3 (2006) pages 485-493

[WSBW01] Wald I., Slusallek P., Benthin C., Wagner M.: Interactive Rendering with Coherent Ray Tracing. EUROGRAPHICS 2001 Volume 20, Number 3

[ZHWG08] Zhou K., Gong M., Huang X., Guo B.,: Real-Time KD-Tree Construction on Graphics Hardware. SIGGRAPH Asia 2008.

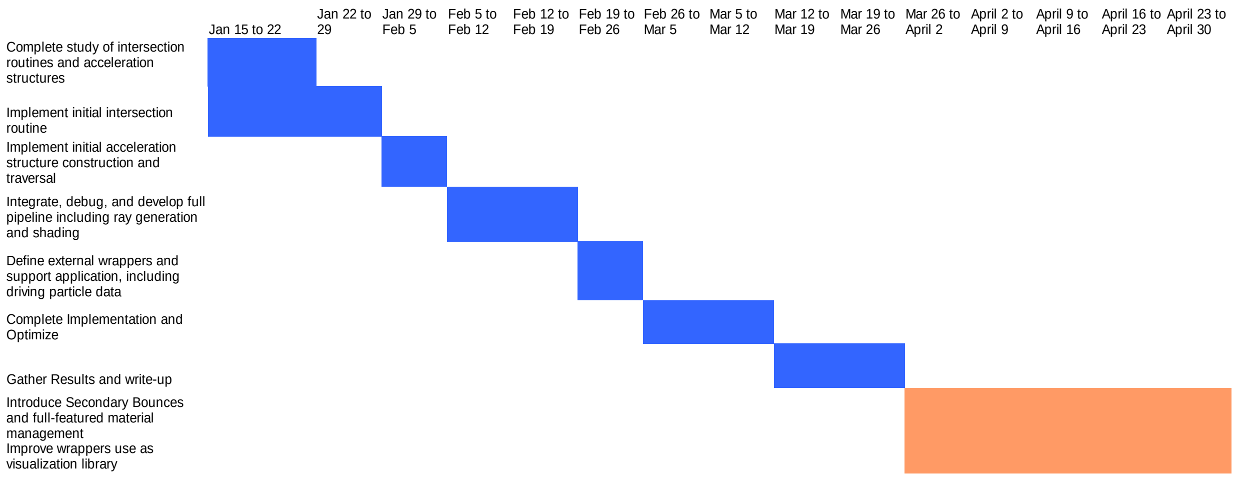

Figure 1: *Gantt Chart. April 1st is our deadline for publication, with time after that being spent on refining our application.*