CHAPTER 29

Efficient Particle Volume Splatting in a Ray Tracer

Aaron Knoll, R. Keith Morley, Ingo Wald, Nick Leaf, and Peter Messmer
NVIDIA

ABSTRACT

Rendering of particle data sets is a common problem in many domains including games, film, and scientific visualization. Conventionally, this has been accomplished using rasterization-based splatting methods, which scale linearly with respect to problem size. Given sufficiently low-cost ray traversal with logarithmic complexity, splatting within a ray tracing framework could scale better to larger geometry. In this chapter, we provide a method for efficiently rendering larger particle data, exploiting ray coherence and leveraging hardware-accelerated traversal on architectures such as the NVIDIA RTX 2080 Ti (Turing) GPUs with RT Cores technology.

29.1 MOTIVATION

Rasterization-based GPU splatting approaches generally break down when most primitives have subpixel footprints and when depth sorting is desired. This occurs due to the linear cost of depth-sorting fragments, as well as incoherent framebuffer traffic, and in practice hampers interactive performance for particle counts beyond 20 million depending on the GPU. There are numerous workarounds for faster raster performance including view-dependent spatial subdivision, level of detail, disabling the depth test and alpha blending, or resampling onto a proxy such as texture slices. However, for all of these, performance suffers when one actually renders a sufficiently high number of particles.

One could equally use ray tracing architectures to efficiently traverse and render full particle data. Traversing an acceleration structure generally has logarithmic time complexity; moreover, it can be done in a way that fosters many small, localized primitive sorts instead of a single large sort. In this manner, we wish performance to mirror the number of primitives actually intersected by each ray, not the total complexity of the whole scene. There are other reasons for rendering particle data within a ray tracing framework, for example allowing particle

© NVIDIA 2019

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_29

effects to be efficiently rendered within reflections. Ray casting large quantities of transparent geometry poses its own challenges; this chapter provides one solution to this problem. It is particularly geared toward visualization of large sparse particle data from N-body and similar simulations, such as the freely available DarkSky cosmology data sets [6] shown in Figure 29-1. It could also be of use in molecular, materials, and hydrodynamics simulations and potentially larger particle effects in games and film.

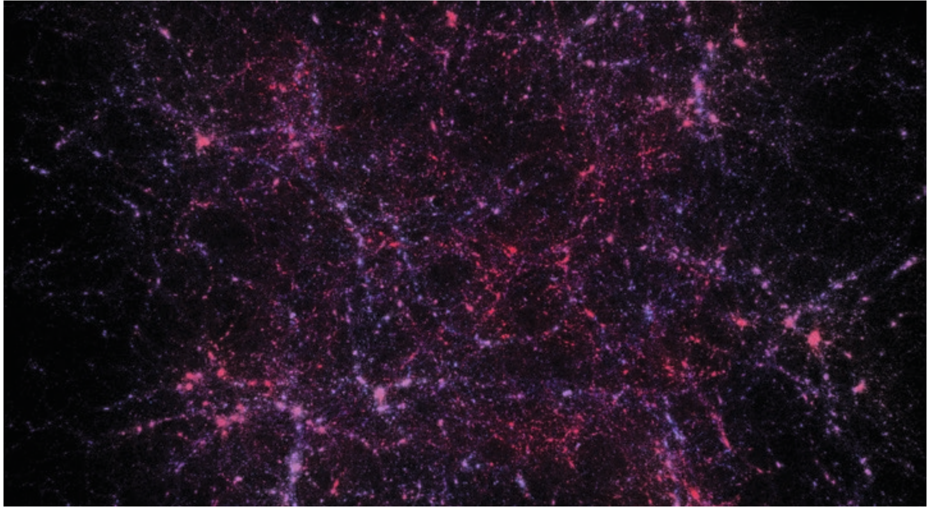

Figure 29-1. *One hundred million particle subset of the DarkSky N-body gravitational cosmology simulation, rendered in its entirety at 35 FPS (1080p) or 14 FPS (4k) without level of detail, on an NVIDIA RTX 2080 Ti with RT Cores technology.*

29.2 ALGORITHM

Our aim is to create a scalable analog to rasterization-based billboard splatting (see, e.g., Westover's work [7]) using ray tracing traversal. The core idea is to sample each particle close to its center point along the viewing ray, then integrate over the set of depth-sorted samples along that ray.

Our primitive is a radial basis function (RBF) with a radius r, particle center P, and bounds defined by a bounding box centered around the particle with width $2r$. The sample (intersection hit) point X is given by the distance to the center of the particle P evaluated along the ray with origin O and direction $\mathbf{d}$,

$$X = O + \left\| P - O \right\| \mathbf{d}. \tag{1}$$

We then evaluate a Gaussian radial basis function at this sample point,

$$\phi(X) = e^{-(X-P)^2/r^2}. \tag{2}$$

This primitive test occurs in object space, sampling the RBF within a three-dimensional bounding box as opposed to a two-dimensional billboard in a rasterized splatter. This yields more continuous results when zoomed into particle centers and does not require refitting the acceleration structure to camera-aligned billboard geometry.

Then, the set of depth-sorted samples $\{\phi(X_i)\}$ along each ray is composited using the over operator [3],

$$\mathbf{c}_f = (1-\alpha)\mathbf{c}_b + \alpha\mathbf{c}, \tag{3}$$

where the opacity of a sample $\alpha = \phi(X_i)$ and color $\mathbf{c} = \mathbf{c}(\phi(X_i))$ correspond to the current sample mapped via a transfer function, and f and b denote front and back values in the blending operation, respectively.

29.3 IMPLEMENTATION

Our challenge is now to efficiently traverse and sort as many particles as possible within a ray tracing framework. We chose to use the NVIDIA OptiX SDK [4], which is suited for scientific visualization and high-performance computing applications running under Linux. Though more memory-efficient approaches would be beneficial, for this sample we use a generic 16-byte (`float4`) primitive paired with the default acceleration structure and traversal mechanism supplied by the ray tracing API.

This method could be implemented naively with an OptiX [4] closest-hit program, casting first a primary ray and then a secondary transmission ray for each particle hit until termination. However, this would entail large numbers of incoherent rays, resulting in poor performance.

We therefore use an approach that coherently traverses and intersects subregions of the volume in as few traversals as possible, as shown in Figure 29-2. This bears similarities to the RBF volume methods [2], as well as game particle effects that resample onto regularly spaced two-dimensional texture slices [1]. However, it is simpler and more brute-force in the sense that, given a sufficiently large buffer to prevent overflow, it faithfully reproduces every intersected particle. We implement this using an any-hit program in OptiX as described in Section 29.3.2.

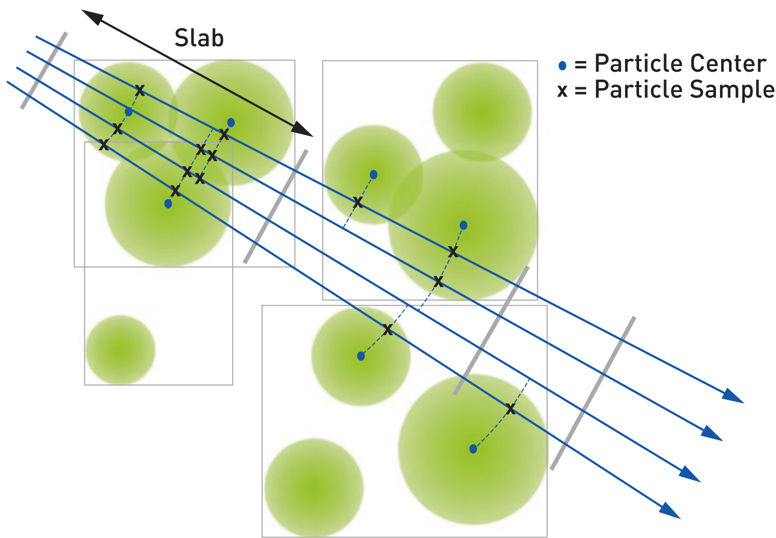

Figure 29-2. *Overview of our algorithm. The geometric primitive is a spherical radial basis function centered at a point. The hit position is the distance to the particle center evaluated along the view ray. To ensure coherent behavior during traversal of this geometry, we divide the volume into segments along rays, resulting in slabs. We then traverse and sort the set of intersected particles within each slab.*

29.3.1 RAY GENERATION PROGRAM

Our approach proceeds as follows: we intersect the volume bounding box and divide the resulting interval into slabs, spaced by `slab_spacing`. For each slab, we set the `ray.tmin` and `ray.tmax` to appropriately prune acceleration structure traversal. We then traverse with `rtTrace()`, which fills the buffer in `PerRayData` with the intersected samples within that slab. We subsequently sort and integrate that list of samples in the buffer. The following pseudocode omits some details (evaluating the radial basis function and applying a transfer function); for complete code refer to the accompanying source (Section 29.5).

```
 1 struct ParticleSample {
 2   float t;
 3   uint id;
 4 };
 5
 6 const int PARTICLE_BUFFER_SIZE = 31;    // 31 for Turing, 255 for Volta
 7
 8 struct PerRayData {
 9   int             tail;                  // End index of the array
10   int             pad;
11   ParticleSample     particles[PARTICLE_BUFFER_SIZE]; // Array
12 };
13
```

```
14 rtDeclareVariable(rtObject,        top_object, , );
15 rtDeclareVariable(float,           radius, , );
16 rtDeclareVariable(float3,          volume_bbox_min, , );
17 rtDeclareVariable(float3,          volume_bbox_max, , );
18 rtBuffer<uchar4, 2>                output_buffer;
19
20 RT_PROGRAM raygen_program()
21 {
22   optix::Ray ray;
23   PerRayData prd;
24
25   generate_ray(launch_index, camera); // Pinhole camera or similar
26   optix::Aabb aabb(volume_bbox_min, volume_bbox_max);
27
28   float tenter, texit;
29   intersect_Aabb(ray, aabb, tenter, texit);
30
31   float3 result_color = make_float3(0.f);
32   float result_alpha = 0.f;
33
34   if (tenter < texit)
35   {
36     const float slab_spacing =
37           PARTICLE_BUFFER_SIZE * particlesPerSlab * radius;
38     float tslab = 0.f;
39
40     while (tslab < texit && result_alpha < 0.97f)
41     {
42       prd.tail = 0;
43       ray.tmin = fmaxf(tenter, tslab);
44       ray.tmax = fminf(texit, tslab + slabWidth);
45
46       if (ray. tmax > tenter)
47       {
48         rtTrace(top_object, ray, prd);
49
50         sort(prd.particles, prd.tail);
51
52         // Integrate depth-sorted list of particles.
53         for (int i=0; i< prd.tail; i++) {
54           float drbf = evaluate_rbf(prd.particles[i]);
55           float4 color_sample = transfer_function(drbf); // return RGBA
56           float alpha_1msa = color_sample.w * (1.0 - result_alpha);
57           result_color += alpha_1msa * make_float3(
58                 color_sample.x, color_sample.y, color_sample.z);
59           result_alpha += alpha_1msa;
60         }
61       }
```

```
62        tslab += slab_spacing;
63      }
64    }
65
66    output_buffer[launch_index] = make_color( result_color ));
67 }
```

29.3.2 INTERSECTION AND ANY-HIT PROGRAMS

The intersection program is simple even when compared to ray/sphere intersection: We use the distance to the particle center along the viewing ray as the hit point `sample_pos`. We then check whether the sample is within the RBF radius; if so, we report an intersection. Our any-hit program then appends the intersected particle to the buffer, which is sorted by the ray generation program when traversal of the slab completes.

```
 1 rtDeclareVariable(ParticleSample,hit_particle,attribute hit_particle,);
 2
 3 RT_PROGRAM void particle_intersect( int primIdx )
 4 {
 5   const float3 center = make_float3(particles_buffer[primIdx]);
 6   const float t = length(center - ray.origin);
 7   const float3 sample_pos = ray.origin + ray.direction * t;
 8   const float3 offset = center - sample_pos;
 9   if ( dot(offset, offset) < radius * radius &&
10        rtPotentialIntersection(t) )
11   {
12     hit_particle.t = t;
13     hit_particle.id = primIdx;
14     rtReportIntersection( 0 );
15   }
16 }
17
18 RT_PROGRAM void any_hit()
19 {
20   if (prd.tail < PARTICLE_BUFFER_SIZE) {
21     prd.particles[prd.tail++] = hit_particle;
22     rtIgnoreIntersection();
23   }
24 }
```

29.3.3 SORTING AND OPTIMIZATIONS

The choice of `PARTICLE_BUFFER_SIZE` and consequently the ideal sorting algorithm depends on the expected performance of `rtTrace()`. On the NVIDIA Turing architecture with dedicated traversal hardware, we achieved the best performance with an array size of 31 and bubble sort. This is not surprising given the small size of the array, and that the elements are already partially sorted from

bounding volume hierarchy traversal. On architectures with software traversal such as Volta, we experienced best results with a larger array of 255, relatively fewer slabs (thus traversals), and bitonic sort. Both are implemented in our reference code.

The value of `particlesPerSlab` should be chosen carefully based on the desired radius and degree of particle overlap; in our cosmology sample we default to 16. For larger radius values particles may overlap such that a larger `PARTICLE_BUFFER_SIZE` is required for correctness.

29.4 RESULTS

Performance of our technique on both NVIDIA RTX 2080 Ti (Turing) and Titan V (Volta) architectures is provided in Table 29-1, for a screen-filling view of the DarkSky data set at 1080p (2 megapixel) and 4k (8 megapixel) screen resolutions. The RT Cores technology in Turing enables performance at least 3× faster than on Volta, and up to nearly 6× in the case of smaller scenes.

Table 29-1. *Performance in milliseconds for screen-filling DarkSky reference scenes of varying numbers of particles.*

	1080p		4k	
#Particles	RTX 2080 Ti	Titan V	RTX 2080 Ti	Titan V
1M	2.9	17	7.4	33
10M	9.1	33	22	83
100M	28	83	71	220

We found that our slab-based approach was roughly 3× faster than the naive closest-hit approach mentioned in Section 29.2 on Turing, and 6–10× faster on Volta. We also experimented with a method based on insertion sort, which has the advantage of never over-running our fixed-size buffer; this was generally 2× and 2.5× slower than the slabs approach on Turing and Volta, respectively. Lastly, we compared performance with a rasterized splatter [5], and we found that our ray tracing method was 7× faster for the 100M particle data set for both 4k and 1080p resolution on the NVIDIA RTX 2080 Ti, with similar cameras and radii.

29.5 SUMMARY

In this chapter, we describe a method for efficiently splatting on Turing and future NVIDIA RTX architectures leveraging hardware ray traversal. Despite using custom primitives, our method is 3× faster on Turing than on Volta, roughly 3× faster than a naive closest-hit approach, and nearly an order of magnitude faster than a

comparable rasterization-based splatter with depth sorting. It enables real-time rendering of 100 million particles with full depth sorting and blending, without requiring level of detail.

Our approach is geared primarily toward sparse particle data from scientific visualization, but it could easily be adapted to other particle data. When particles significantly overlap, full RBF volume rendering, or resampling onto proxy geometry or structured volumes, may prove more advantageous.

We have released our code as open source in the *OptiX Advanced Samples* Github repository: https://github.com/nvpro-samples/optix_advanced_samples.

REFERENCES

[1] Green, S. Volumetric Particle Shadows. NVIDIA Developer Zone, https://developer.download.nvidia.com/assets/cuda/files/smokeParticles.pdf, 2008.

[2] Knoll, A., Wald, I., Navratil, P., Bowen, A., Reda, K., Papka, M. E., and Gaither, K. RBF Volume Ray Casting on Multicore and Manycore CPUs. *Computer Graphics Forum 33*, 3 (2014), 71–80.

[3] Levoy, M. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 3 (1988), 29–30.

[4] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., et al. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (2010), 66:1–66:13.

[5] Preston, A., Ghods, R., Xie, J., Sauer, F., Leaf, N., Ma, K.-L., Rangel, E., Kovacs, E., Heitmann, K., and Habib, S. An Integrated Visualization System for Interactive Analysis of Large, Heterogeneous Cosmology Data. In *Pacific Visualization Symposium* (2016), pp. 48–55.

[6] Skillman, S. W., Warren, M. S., Turk, M. J., Wechsler, R. H., Holz, D. E., and Sutter, P. M. Dark Sky Simulations: Early Data Release. arXiv, https://arxiv.org/abs/1407.2600, July 2014.

[7] Westover, L. Footprint Evaluation for Volume Rendering. *Computer Graphics (SIGGRAPH) 24*, 4 (1990), 367–376.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (http://creativecommons.org/licenses/by-nc-nd/4.0/), which permits any noncommercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.