第29章

光线追踪器里的高效粒子体积分割

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

NVIDIA

摘要

粒子数据的渲染是游戏、电影、科学可视化等很多领域的一个常见的问题。这项技术传统上可以通过基于栅格化的分割方法来实现。这种方法可以根据问题的体积进行线性的缩放。假设给定了足够低消耗的射线以对数复杂度运动，光线追踪框架中的分割就可以更好地扩大到大型的集合体。在这一章节，我们提供了一个能够高效率地渲染大型粒子数据的方法。此方法用于研究光线的相关度以及在持有RT Cores科技的NVIDIA RTX 2080 Ti (Turing) GPU架构上最大化硬件加速。

29.1 动机

当大多数图元有次像素空间(subpixel footprints)[@note1]以及需要进行深度排序时，基于格栅化的GPU分割方法基本就不可行了。这个问题的是由于对片元进行深度排序的线性消耗不相关的帧缓冲的堵塞，以及实际应用中对于超过2千万的粒子数交互性能会受到阻碍（取决于GPU）所导致的。有大量的解决办法可以达到更快的格栅化，包括依赖于视角的空间细分，LOD细节层次，关闭深度测试和透明度混合，或者重新在代理上采样，比如纹理切片(resampling onto a proxy such as texture slices)[@note2]。然而针对所有上述方法，在渲染相当大数量的粒子时性能都会受到影响。

光线追踪架构可以有效地纵览和渲染完整的粒子数据。纵览一个加速结构通常有一个对数时间复杂度。此外，它可以通过很多小的，本地的图元排序来完成，而不是一个单一的大型排序。通过这个手段，我们希望性能反映的是图元与每束射线相交的的数量，而不是总体场景的复杂程度。在光线追踪架构内渲染粒子数据还有其他的原因，比如允许高效地渲染带反射的粒子效果。对大型透明几何物体使用射线投射算法也带来了新的挑战，这个章节提供了一个解决方案。它尤其适合来自N-body和类似模拟的大型空间粒子数据的可视化，比如图29-1所示的免费可获取的DarkSky太空数据集。它同样可以用于分子、材料、流体动力仿真和游戏电影里潜在的更大型的粒子效果。



图29-1。DarkSky N-body引力宇宙学的一亿个粒子的子集模拟，在持有RT Cores技术的英伟达RTX 2080Ti显卡上渲染，全部以35帧/秒（1080P）或14帧/秒 (4K)渲染，无细节层次。

29.2 算法

我们的目标是创建一个可扩展，类似于基于格栅化的使用光线追踪遍历的的广告牌分割(billboard splatting)@note3（参见 Westover的工作[7]）。 它的核心理念是沿着视角射线对靠近其中心点的每个粒子采样，然后沿着射线深度排序的采样集合进行积分。

我们的图元是一个径向基函数(RBF)，含有一个半径r，粒子中心P和边界。边界是通过2r宽度围绕粒子中心的包围盒来定义的。采样点(交叉碰撞)X由沿着原点为O和方向为**d**的射线到粒子P中心的距离计算得出。

X =O + ||P -O|| **d**. (1)

之后，我们估算一个在这一采样点的高斯径向基函数，

C:\Users\wenyu\AppData\Local\Temp\1569051457(1).png (2)

这个图元测试发生在物体空间，在三维包围盒里对RBF采样，它不同于栅格化分割里的二维的广告牌@note3。这在放大到粒子中心时，将生成更连续的结果。并且不需要把加速结构重新调整为与摄影机对齐的广告牌几何体。

然后，沿着每束光线的深度排序采样的集合{*ϕ*(*Xi*)}都通过以下的表达式[3]进行合并，

C:\Users\wenyu\AppData\Local\Temp\1569051542(1).png (3)

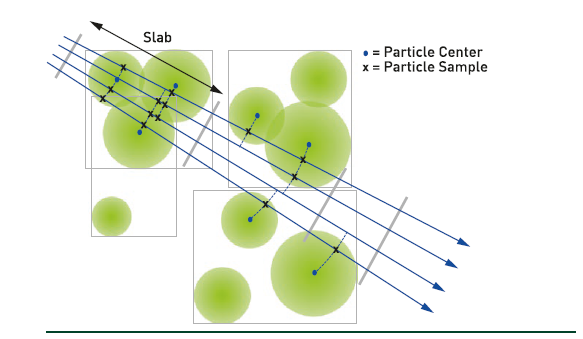
在此，一个采样*α* = *ϕ*(*Xi*)的透明度和颜色**c** = **c**(*ϕ*(*Xi*))对应于通过传递函数映射的当前采样点，f和b分别表示混合操作前后值。

29.3 实现

我们目前的挑战是高效地在光线追踪架构里遍历和排序尽可能多的粒子。我们选择了NVIDIA OptiX SDK [4]，它适合在Linux下运行科学可视化和高性能计算的应用。尽管使用内存效率更高的方法会有益处，但对于这个案例我们采用通用的16字节[float4]图元，配以光线追踪API支持的默认的加速结构和遍历机制。

这个方法可以在OptiX [4] closest-hit程序里直接执行，对于每个命中的粒子,首先投射一次射线然后投射二次透射射线,直到终止。然而，这将生成大量不相关的光线，从而导致性能的低下。

我们因此用一种方法能尽可能减少相关的遍历和相交体积的子区域@note4，如图29-2所示。此方法与RBF体积方法[2]有些类似，并且游戏粒子效果可以重新采样到正常空间化的二维纹理切片上.但是它确实更简单也更暴力(brute-force)，假设有一个足够大的缓冲来防止溢出，它会认真地重建每个相交的粒子。我们将在29.3.2讨论在OptiX里用任意碰撞(any-hit)程序实现这个功能。



**图29.2。** 我们的算法概览。几何图元是一个以点为中心的球形径向基函数。命中位置是沿着视线到粒子中心的距离。为了确保在遍历此集合体时行为保持一致，我们把体积沿着射线分割成多个分段，从而变成条状。然后我们在每个长条里对相交的粒子集合进行遍历和排序。

29.3.1 光线生成程序

我们的方法如下：我们与体积包围盒相交,并将得到的结果间隔分成多个长条(slab)，通过slab\_spacing来隔开。对于每一个长条，我们定义ray.tmin和ray.tmax来恰当的裁切加速结构的遍历。然后,我们使用rtTrace() 来遍历，它通过长条里碰撞的采样点填充PerRayData中的缓冲区。随后,我们将样本列表排序并整合到缓冲区中下面的伪代码省略了一些细节{估算径向基函数和应用转换函数}；完整的代码请参考附带的资源 [小节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 相交和任意触碰程序

即使和光线/球体相交比较，相交程序依然很简单：我们把沿着视线到粒子中心的距离作为命中点sample\_pos。然后我们检查采样点是否在RBF半径内；如果是，我们报告有一次交叉。之后,我们的任意触碰程序会把交叉的粒子附到缓冲里，当长条遍历完成时,射线生成程序会对缓冲区进行排序。

**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 排序和优化

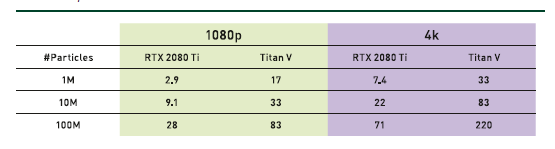
PARTICLE\_BUFFER\_SIZE的选择和后续的理想排序算法取决于rtTrace()的预期性能。在具有专用遍历硬件的NVIDIA Turing架构上，我们用大小为31的阵列和冒泡排序达到了最佳性能。对于小规模的阵列这并不令人惊奇。而且那些元素已经根据边界体积层次结构遍历进行了部分排序了。在例如Volta这种带有软件遍历的架构上，我们在一个255大小的大型阵列相对较少的长条（以及遍历）和双调排序上获得了最佳结果。它们都在我们的参考代码中实现了。

particlesPerSlab的值应该基于所需的半径和粒子重叠程度仔细地选择；在我们的宇宙采样样本中我们默认为16。对于更大的半径值，粒子可能重叠，因此更大的PARTICLE\_BUFFER\_SIZE才能确保正确性。

29.4 结果

我们的技术在NVIDIA RTX 2080 Ti (Turing)和Titan V(Volta)架构上的性能可见表29-1，可以通过填充屏幕的视角看到1080p(200万像素)和4k(8百万像素)下的DarkSky数据集。Turing架构下的RT Cores技术使性能至少比Volta快三倍，在较小场景下近乎快6倍。

**表29-1.** \屏幕填充不同数量粒子的DarkSky参考场景的性能(以毫秒为单位)



我们发现在Turing上，我们基于长条的方法大致比29.2里提到的cloest-hit方法快了三倍，在Volta上则可以快6至10倍。我们对基于插入排序的方法也进行了实验，该方法的优点永远不会超出固定大小的缓冲区；这个方法通常比Turing和Volta上的长条方法慢2倍和2.5倍。最后，我们将性能与光栅化分割@note5进行了比较。我们发现,在具有相似相机和半径的NVIDIA RTX 2080TI上,对于4k和1080p分辨率的100M粒子数据集,我们的光线跟踪方法快了7倍.

29.5 总结

在这一章节，我们阐述了一个利用硬件光线遍历来来高效地在Turing和未来NVIDIA RTX架构上使用抛雪球算法。尽管用了自定义的图元，我们的方法在Turing上比在Volta上快了三倍，比cloest-hit方法快了约三倍，比带有深度排序的基于格栅化的抛雪球方法快了一个数量级。它不需要任何细节级别的处理就可以实时渲染一亿个粒子,并且进行完整的深度排序和混合.我们的方法主要面向来自科学可视化的空间粒子数据，但是可以很容易地将其应用到其他粒子数据。当粒子明显地重叠时，完整的RBF体积渲染或者重新采样到代理几何体上或结构体积上，可能会更能凸显优势。

我们把代码开源放在了OptiX Advanced Samples Github库中: https://github.com/nvpro-samples/optix\_advanced\_samples.

参考文献

[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.