NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation

Ken Museth
NVIDIA
Santa Clara, CA, USA
kmuseth@nvidia.com

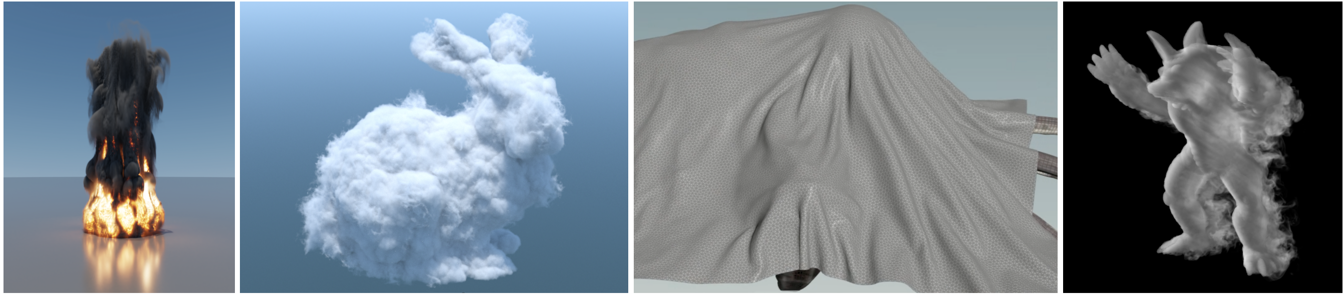

Figure 1: Rendering and simulation on GPUs with NanoVDB: The first two images from the left show Physics-Based Ray-Tracing (in PBRT v4) of NanoVDB volumes on a RTX 3090. The third image from the left demonstrates how NanoVDB can be used for real-time simulation of cloth in Houdini 18.5. Specifically, NanoVDB represents a sparse signed distance field (SDF) used for real-time collision detection on the GPU. The image on the far right shows NanoVDB applied to a real-time fluid simulation on the GPU. An armadillo model, represented as a NanoVDB, is used for sourcing of density in NVIDIA's Flow fluid solver at interactive frame-rates on a GPU. Credits are included in the accompanying video.

ABSTRACT

We introduce a sparse volumetric data structure, dubbed NanoVDB, which is portable to both C++11 and C99 as well as most graphics APIs, e.g. CUDA, OpenCL, OpenGL, WebGL, DirectX 12, OptiX, HLSL, and GLSL. As indicated by its name, NanoVDB is a mini-version of the much bigger OpenVDB library, both in terms of functionality and scope. However, NanoVDB offers one major advantage over OpenVDB, namely support for GPUs. As such it is applicable to both CPU and GPU accelerated simulation and rendering of high-resolution sparse volumes. In fact, it has already been adopted for real-time applications by several commercial renders and digital content creation tools, e.g. Autodesk's Arnold, Blender, SideFX's Houdini, and NVIDIA's Omniverse just to mention a few.

KEYWORDS

Sparse volumes, gpu data structures, rendering, simulation

ACM Reference Format:

Ken Museth. 2021. NanoVDB: A GPU-Friendly and Portable VDB Data Structure For Real-Time Rendering And Simulation. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks (SIGGRAPH '21 Talks), August 09-13, 2021.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3450623.3464653

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SIGGRAPH '21 Talks, August 09-13, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8373-8/21/08.
https://doi.org/10.1145/3450623.3464653

MOTIVATION

Efficient data structures are characterized by at least two attributes; computational efficiency and memory efficiency. While it is relatively easy to develop a fast or a compact volumetric data structure, e.g. a dense grid or an octree, it is surprisingly challenging to come up with one that is both fast and compact. This has lead to several advances in the pursuit of efficient sparse volumetric data structures, like DT-Grid [Nielsen and Museth 2006], VDB [Museth 2013], SPGrid [Setaluri et al. 2014], and TaiChi [Hu et al. 2019]. VDB was open sourced in 2012, as OpenVDB[1], and has since become a standard for sparse volumes in the movie industry. However, unlike the more recent TaiChi, which also builds on VDB, OpenVDB is limited to the CPU, and more to the point does not work on the GPU. Given the fact that GPUs have long offered superior computational acceleration over CPUs, this seems like an obvious avenue for improving the efficiency of an industry standard. This is precisely the motivation behind the development of NanoVDB presented here.

HOW IS NANOVDB SIMILAR TO OPENVDB?

NanoVDB shares several attributes with OpenVDB, most notably the memory efficiency of the underlying VDB tree structure detailed in [Museth 2013]. Similarly, it also makes use of bottom-up traversal of the hierarchical tree structure to achieve on average constant-time complexity for random access. The same is true for sequential access as well as the truly unbounded index domain, only limited by the bit-precision of signed 32bit coordinates. Finally, NanoVDB supports voxels of the same types as OpenVDB, e.g. int, float, Vec3f, and bits masks, as well as points with arbitrary attributes.

[1] http://www.openvdb.org

WHAT IS NOVEL ABOUT NANOVDB?

What sets NanoVDB apart from other sparse volumetric data structures that we are aware of, and in particular OpenVDB, is the fact that by design NanoVDB is portable to a long list of hardware, compilers and APIs. It works on CPUs (e.g. Intel and ARM) and GPUs (e.g. NVIDIA and AMD), compiles with C++11 and C99, and is compatible with most graphics APIs used for games, high-performance computing and real-time applications. More specifically, we have tested NanoVDB with CUDA, OpenCL, OpenGL, WebGL, DirectX 12, OptiX, HLSL, and GLSL. This is achieved by a complete rewrite of the foundational VDB tree structure that strips away all external dependencies found in OpenVDB, eliminating pointers, adding a vanilla-C implementation, and introducing explicit 32 byte memory alignment of all tree nodes. Consequently, NanoVDB is by design portable and nimble (a single header-file in either C++ or C99) and more importantly does not depend on OpenVDB.

Conceptually, NanoVDB is a linearized snapshot of an OpenVDB data structure, where nodes are 32B aligned in a continuous block of memory, typically with a breadth-first layout of tree nodes, i.e. nodes at the same level are packed tightly together. Since NanoVDB, unlike OpenVDB, explicitly avoids the use of memory pointers in its data structures, NanoVDB is efficient and fast to copy between devices, e.g. CPU and GPU. In fact, NanoVDB uses this novel memory layout as its independent file format, which implies that client code of NanoVDB volumes, e.g. a proprietary ray-tracer, only needs to include a few header files with no additional dependencies. Benchmark tests have even shown that on CPUs NanoVDB is faster at random access than OpenVDB, likely because of its more cache friendly memory layout that is not fragmented as in OpenVDB.

Unique to NanoVDB is also the addition of statistical metadata that is encoded into all the nodes of the hierarchical VDB tree structure. This includes minimum and maximum values as well as averages and standard deviations. This has proven useful for several applications, most notably acceleration of volume rendering and early termination during ray-tracing. Similarity all tree nodes in NanoVDB include bounding boxes of all the active values in their sub-branches, which allows for tighter ray-clipping and more accurate construction of BVH acceleration structures when making used of hardware-accelerated ray-tracing, e.g. leveraging NVIDIA's RTX cores.

Finally, NanoVDB offers both out-of-core and in-core compression, whereas OpenVDB only supports the former. In other words, NanoVDB can be configured to perform run-time decompression on random access to voxel values. Currently we support blocked floating-point bit quantizations with either a fixed bit-rate (e.g. 2, 4, 8, and 16 bits) or an adaptive per-block bit-rate. Both compression codecs utilize the per node statistics mentioned above, and the adaptive bit-rate codec also uses a custom refinement-oracle, e.g. based on a global tolerance or a distance-to-camera error metric. We have found these compression techniques to significantly reduce the memory footprints (typically by 4 to 6 times) with little to no visible artifacts. Furthermore, these relatively simple codecs are sufficiently fast that we observe small performance improvements (typically 10 to 30 %) when compared to the uncompressed data, suggesting that our benchmark tests are memory (vs compute) bound, which is not uncommon for volumetric rendering on GPUs.

TOOLS AND FUNCTIONALITIES OF NANOVDB

Whereas OpenVDB offers a large toolbox of accompanying algorithms to support dynamic simulations, NanoVDB ships with a relatively small set of tools specifically included to support real-time rendering and simulation applications with static sparse volumes. Figure 1 illustrates several such examples. As such, NanoVDB includes GPU friendly numerical schemes for 0th-3rd order accurate interpolation, 1st-5th order accurate gradients, 2nd order accurate mean, gaussian and principal curvatures, as well as Hierarchical Digital Differential Analyzers [Museth 2014] for efficient empty space skipping of volume and SDF rendering. Included are also fast multi-threaded converters to and from OpenVDB, which allows for round-trips between the two data structures.

Additionally, NanoVDB includes efficient tools to hierarchically re-compute the statistics metadata, e.g. after values have been modified, as well as fast tools to compute and validate Cyclic Redundancy Code checksums of NanoVDB files and in-memory buffers. This adds a level of safety when NanoVDB is used in complex production pipelines as an exchange format between digital creation tools and renderers. We even include tools to create NanoVDB grids from scratch, which allows the authoring of NanoVDB volumes without any dependency of OpenVDB whatsoever. Finally, NanoVDB includes a stand-alone interactive ray-tracer that demonstrates how NanoVDB can be used with most graphics APIs.

LIMITATIONS AND FUTURE WORK

By far, the biggest limitation of NanoVDB is the fact that as of today it does not include tools to modify the topology of the tree. While modification of values is trivial, the same cannot be said about adding or removing nodes to a tree that is serialized in memory. This is not to say that it is impossible, in fact the data structure is deliberately designed to allow for nodes to reside anywhere in memory, i.e the use of dynamic memory pools is possible. Nevertheless, for now we have decided to focus on static trees since it plays such an important role for especially real-time rendering and even many simulation applications.

We expect NanoVDB to be a first-class member of the OpenVDB library in the near future, and we look forward to expanding its toolset and applying it to more real-time applications.

ACKNOWLEDGMENTS

We thank NVIDIA for supporting and open sourcing this project. Major thanks goes to especially Wil Braithwaite, in particular for the interactive viewer, as well as Andrew Reidmeyer and Jeff Lait for their help with the port of NanoVDB to OpenCL and HLSL/GLSL.

REFERENCES

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. 38, 6 (2019).

Ken Museth. 2013. VDB: High-resolution Sparse Volumes with Dynamic Topology. *ACM Trans. Graph.* 32, 3, Article 27 (July 2013), 22 pages. https://doi.org/10.1145/2487228.2487235

Ken Museth. 2014. Hierarchical Digital Differential Analyzer for Efficient Ray-Marching in OpenVDB. Association for Computing Machinery.

Michael B. Nielsen and Ken Museth. 2006. Dynamic Tubular Grid: An Efficient Data Structure and Algorithms for High Resolution Level Sets. 26, 3 (2006).

Rajsekhar Setaluri, Mridul Aanjaneya, Sean Bauer, and Eftychios Sifakis. 2014. SPGrid: A Sparse Paged Grid Structure Applied to Adaptive Smoke Simulation. 33, 6 (2014).