Volkit: A Performance-Portable Computer Vision Library for 3D Volumetric Data

Stefan Zellmann¹[0000–0003–2880–9090], Giovanni Aguirre²[0000–0001–7999–8528],
and Jürgen P. Schulze²[0000–0003–4903–4837]

¹ University of Cologne, Weyertal 121, 50931 Cologne, Germany
zellmann@uni-koeln.de
² UC San Diego, 9500 Gilman Drive, La Jolla, CA 92039, USA
{g3aguirre,jschulze}@ucsd.edu

Abstract. We present volkit, an open source library with high performance implementations of image manipulation and computer vision algorithms that focus on 3D volumetric representations. Volkit implements a cross-platform, performance-portable API targeting both CPUs and GPUs that defers data and resource movement and hides them from the application developer using a managed API. We use volkit to process medical and simulation data that is rendered in VR and consequently integrated the library into the C++ virtual reality software CalVR. The paper presents case studies and performance results and by that demonstrates the library’s effectiveness and the efficiency of this approach.

Keywords: Volume data · Computer vision algorithms · Virtual reality.

1 Introduction

Volumetric data representations are prevalent in the simulating sciences as well as in the clinical daily routine. While design studios or 3D artists often use dedicated graphics workstations, typical clinical IT systems have restricted hardware capabilities, and simulation codes are run on supercomputers that use compute
GPUs, or CPUs with instruction set architectures like PowerPC or ARM that are best targeted with vendor-specific APIs.

Computer vision libraries like OpenCV [5] or the Insight Toolkit [10] focus on 2D images, are optimized for single platforms, and don’t support hierarchical or unstructured volume data. As such there is still a lack of a general, cross-platform, performance-portable library for 3D image processing and computer vision tasks (cf. Fig. [1]) using modern concepts such as the ones described below.

The simulating sciences often employ a post processing pipeline at the end of the exploration process that does or does not involve 2D or 3D visualization, but that usually requires data wrangling to transition the data from one pipeline—the data source—to the other one that is concerned with post processing. It is this interface between data acquisition and post processing where we position our cross-platform volume manipulation library volkit.

Volkit was designed to accommodate the typical data wrangling, data filtering and post processing phases that most pipelines involve prior to data presentation and as such implements tasks that most typical interactive 3D or virtual reality rendering systems would implement. Volkit aims at both high performance, high bandwidth data processing and at versatility and ease of use.

Volkit is not a 3D volume rendering library in particular—although it comes with a high quality path tracing renderer that might be suitable for rendering in many cases—yet we acknowledge that volumetric image processing eventually may culminate in the user visually exploring their data sets. Visual exploration, and in particular interactive 3D visualization is a part of a typical scientific post processing pipeline, and we therefore also integrate and evaluate volkit in the context of the virtual reality and scientific visualization software CalVR [25].

2 Related Work

The most common volumetric data representations nowadays are structured and unstructured [24] as well as hierarchical grids—e.g., Octrees [6] and other, often regular trees with wider or variable branching factors [7]. While structured grids are most frequently generated by microscopes and CT or MRI scanners, unstructured and hierarchical grids—the latter are also referred to as adaptive mesh refinement (AMR) [4,3]—are often the result of simulation codes. Typical codes that output AMR data are for example FLASH [7] or p4est [6].

The image processing library OpenCV [5] provides many filtering and contrast enhancement algorithms, but doesn’t support volumes. ITK [10] does support images of higher dimension than 2D, but treats those as if they were 2D images and hence has no notion of the topologies described above that are arguably more common in 3D than they are in 2D. Furthermore, ITK is targeted at CPUs only. The volume manipulation tool 3D Slicer [11] has a Python API, but is not performance-portable and only supports structured volumes.

While performance-portable volume manipulation libraries for arbitrary topologies are scarce, there are other GPU libraries hat volkit draws inspiration from. The deferred API for example was inspired by the OptiX ray tracing
Volkit Image Manipulation Library for Volumetric Data

Fig. 2. Path tracing an astrophysics data set using the Render algorithm. Volkit can render both structured and AMR volumes using various integrators. Cell-centric AMR volumes are rendered with high-quality linear interpolation even at level boundaries.

library [20]. OptiX’s node graph 3D scene abstraction uses group nodes with accelerators (bounding volume hierarchies) and optional data buffers. The user specifies which data is copied to those buffers from the CPU, but not when that happens; similarly, when OptiX will (re-)build the accelerator isn’t controlled by the user. OptiX however recently switched to an explicit API where the user has more control at the cost of increased complexity. The OptiX 7 Wrappers Library (OWL) [27] fills that gap by providing a node graph API on top of OptiX 7.

Interactive volume rendering is nowadays implemented using ray casting [15] with absorption and emission [17]. Optimizations such as early ray termination [9] or empty space skipping [8,33,32] using real-time ray tracing hardware [29] render this effective even for large data sets. The community is however gradually transitioning to volumetric path tracing [12,1]. Volkit’s interactive path tracer was inspired by the one proposed by Raab [22]. Rendering unstructured grids requires acceleration data structures [18,28] or connectivity information [19]. Cell-centric AMR volumes require complex interpolants for high-quality reconstruction [31]. It has still been shown that in both cases, with carefully designed data structures, high-quality interactive rendering is possible [23,30].

3 System Overview

With data-intensive GPU and coprocessor libraries, developers have to make certain design decisions that are centered around data management paradigms (e.g., object-oriented or data flow-centric), around when data is copied to and from the device (immediate vs. deferred APIs), and about how much control is given to the user regarding data handling and control flow. With volkit we decided to adopt an implicit, retained mode API using a data management layer for volumes and for auxiliary data such as lookup tables. Volkit uses a data flow-centric paradigm and provides efficient routines to handle very large data sets. On the I/O side this is accomplished using streams, which allow for partial data loading and storing, the specifics of which are described below.
Fig. 3. Interactive ray tracer set up with RGBA lookup table (LUT). The user-supplied LUT is used as a transfer function to classify samples. LUTs are managed and—as with volumes—are identified via their resource handle. The viewer allows for rudimentary mouse navigation. The volumetric path tracing renderer shown here will interactively render convergence frames.

3.1 Deferred and Managed Resource Handling

Volkit uses a deferred memory managed layer for implicit GPU data transfers. For that we use a thread-local ExecutionPolicy object that is accessed via the functions SetThreadExecutionPolicy() and GetThreadExecutionPolicy(). By that we acknowledge that users might themselves run volkit in a multithreading environment. The object stores the volkit state associated with the current thread, such as for example the device type (CPU or GPU). The ExecutionPolicy is also used to set other options, e.g., to enable debug message, or to print out algorithm execution times to the console. When the user changes the device settings of the current ExecutionPolicy, all subsequent algorithms that are executed in the current thread will run on the device that is currently being set.

Algorithms like Render will implicitly call the member function migrate on the managed objects. migrate compares the current ExecutionPolicy object with the one that was active when the managed object was last accesses. For that, the last ExecutionPolicy is carried along by the managed types. If the device has changed, before the data is accessed, a data transfer is initiated and the last ExecutionPolicy is updated. That way, the data is transferred right before it is needed. The user can however also call migrate themselves and by that control when the actual transfer happens. If the current and last ExecutionPolicy are the same, migrate is a no-op. The base class for managed types is called ManagedBuffer<T> and is templated on the storage type (e.g., char for bytes). The volume types and other auxiliary buffers inherit from that template class. Volkit uses resource handles (cf. Fig. 3), which are integral types and allow to identify the managed objects transparently on both CPU and GPU.
3.2 Algorithms and Data Structures

In the following, we give an overview of the most important data types and assorted algorithms. Most of the algorithms that are described below come in two versions: a default version that operates on the whole volume, and a range version where the identifier `Range` is appended to the function name. The range versions operate on a region of interest (ROI) specified via an axis-aligned bounding box. Furthermore, volkit currently has two backends for execution on the CPU or on NVIDIA GPUs using CUDA. The thread-local `ExecutionPolicy` object from above is used to seamlessly switch between CPU and GPU execution.

**Volume Types** Volkit currently supports structured and AMR volumes. Integrating unstructured volumes would however be straightforward. The classes `StructuredVolume` and `HierarchicalVolume` encapsulate the two volume types. Volkit uses the *basis interpolation method* [30] for high-quality reconstruction of cell-centric AMR data. With range algorithms, if the volume is structured, the ROI is specified in cell coordinates. With AMR volumes, ROIs are relative to what we call the *logical grid*—a hypothetical structured grid whose size would allow us to resample the AMR volume without loss of detail.

As proposed by Wald et al. [30], when loading an AMR volume, we drop the AMR hierarchy, keep only the subgrids, and build a bounding volume hierarchy (BVH) over the *active brick regions* (the regions where the interpolation domains of the subgrids overlap). When accessing the volume, we use the BVH to perform point sampling using basis interpolation. We currently support AMR volumes generated by the FLASH simulation code [7], but integrating other file formats would be straightforward. Path tracing of an AMR data set is shown in Fig. 2.

**File Handling** I/O is realized using `InputStream` and `OutputStream`, which are passed a `DataSource` object. Data sources contain headers with meta information such as volume dimensions, cell sizes, cell types, or, in the case of AMR volumes the subgrid meta data. Data sources (which for simplicity also act as sinks) provide low-level I/O facilities via the functions `read`, `write`, `seek`, and `flush` that operate on byte arrays. In contrast to that, stream objects are aware of the actual object that is being read or written—their interface is comprised of functions `{read|write}` whose parameters are `volumes`; the functions `{read|write}Range` allow us to support large volumes via partial loads.

**Volume Manipulation** Volume manipulation algorithms are categorized into *core algorithms* such as, e.g., Fill, Crop, Resample, or Transform. Resample can be used to change the size, the data type, or the topology of the volume (e.g., to resample an AMR volume on a structured grid). Transform iterates over all cells, invokes a user-supplied callback function, and provides that current cell index and pointer to the cell itself, allowing for arbitrary volume manipulation.

---

3 For a complete documentation see volkit’s website under [https://docs.volkit.org](https://docs.volkit.org)
/* --- ANSI C99 --------------------- */
vktStructuredVolume volC;
vktStructuredVolumeCreate(& volC, 64, 64, 64, /* dims */
                          vktDataFormatUint8, /* format */
                          1.f, 1.f, 1.f, /* cell size */
                          0.f, 1.f); /* range */
vktFillRangeSV(volC, 1, 1, 1, 63, 63, 63, 1.f); /* SV: structured */
vktStructuredVolumeDestroy(volC);

/* --- C++03 ------------------------ */
vkt::StructuredVolume volCPP(64, 64, 64, vkt::DataFormat::UInt8);
vkt::FillRange(volCPP, {1, 1, 1}, {63, 63, 63}, 1.f);

---

Fig. 4. ANSI C99 and C++03 APIs, the code snippets demonstrate how to use the two APIs, and the major differences when performing equivalent computations.

Derived algorithms build off of the core algorithms; examples are ApplyFilter to apply user-specified convolution kernels (e.g., edge detection or Gaussian blur), which can be implemented using Transform, or Delete, which is derived from Crop. The significance here is that the algorithms can be implemented using core algorithms—that doesn’t necessarily mean that they actually are: volkit might resort to a more efficient implementation if the behavior stays unaffected.

Transform algorithms (not to be confused with the algorithm Transform) are used to Flip, Scale, or Rotate the volume with subcell accuracy. Domain decomposition algorithms decompose the volume into smaller parts. The BrickDecompose algorithm for example decomposes a given volume into a number of subgrids and optionally computes ghost cells for interpolation. Volkit also includes statistical algorithms such as ComputeHistogram or ComputeAggregates—the latter are, e.g., the min and max cell values and their respective argmin and argmax, the average and standard deviation of the cell values, etc. Arithmetic algorithms are used to compute sums, products, etc. on the whole volume or on subranges.

Rendering Volume rendering is supported through the Render algorithm that will invoke an interactive, windowed ray tracer. The algorithm can be configured to perform either volume ray marching with absorption and emission, implicit iso-surface ray casting, or volumetric path tracing with an isotropic phase function. We used the C++ library Visionaray [34] for the implementation to target x86-64 and ARM CPUs as well as NVIDIA GPUs. Passing an RGBA lookup table to the algorithm will invoke an interactive transfer function editor in the same window as the ray tracer. The RenderState object parameter is used to apply general settings (cf. Fig. 3). In general, the algorithm is meant to be used for prototyping to obtain high-quality renderings with little effort.
3.3 Language Bindings

Volkit is primarily a C/C++ library, but comes with bindings for other languages. Internally, volkit is implemented in C++14 and optionally uses language extensions such as NVIDIA CUDA. We however ship header files with separate interfaces for C99 and for C++03. The two interfaces are designed to expose the same functionality, but the C++03 interface adds a couple of convenience functions such as RAII for managed types, function overloads for short vectors, etc. Exemplary use of the two interfaces is shown in Fig. 4.

We also provide Python 3 bindings that were realized using SWIG [2]; in theory, that should also allow us to support other languages but so far hasn’t been tested. Volkit has a command line interface where algorithms are invoked via separate processes. The base command is called vkt. In order for example to render a volume, a volume file is piped into the process, which is called with the argument render. Inter-process communication is implemented with file I/O. The user can—and is encouraged to—use shared memory to realize that communication to bypass the file system and costly hard drive accesses.

4 Evaluation and Case Studies

We present two case studies as well as a performance study to assess volkit’s effectiveness and efficiency. For the latter we tested an assortment of algorithms using multiple real-world data sets (cf. Table 1).

4.1 Case Study: CLAHE-3D

For this case study we implemented a 3D variant of the Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm found in OpenCV [5] using volkit’s CPU and CUDA backends. CLAHE is used for contrast enhancement.
Table 1. Structured and hierarchical volume data sets we use for the evaluation. See the papers by Lee et al. [13,14] for the DNS data set and by Seifried et al. [26] for the SILCC molecular cloud data set.

| Data Set                          | Type       | Dimensions | Cell Type | Total Size |
|----------------------------------|------------|------------|-----------|------------|
| Heptane (Gas)                    | structured | 302×302×302 | UInt8     | 26.3 MB    |
| Turbulent Channel Flow Sim (DNS) | structured | 2560×1920×384 | Float32   | 7.1 GB     |
| DNS 8                            | structured | 1280×960×192 | Float32   | 900 MB     |
| Richtmyer-Meshkov Instability (RM) | structured | 2048×2048×1920 | UInt8     | 7.5 GB     |
| SILCC Molecular Cloud (ARM)      | hierarchical | 4096×4096×81920 | Float32   | 283 MB     |
and applies histogram equalization to subregions of the volume. This is done via a brick decomposition and local histograms. The ensuing histogram equalization step uses a maximum cut-off to reduce noise in the resulting volume—noise reduction is the major difference compared to other contrast enhancement techniques. The local results are then combined using linear interpolation.

Those steps are implemented using individual CUDA kernels or, on the CPU, loops over either the whole volume, the bricks, or the local histograms. For the implementation we use the various routines provided by volkit that allow for accessing and manipulating volumes. A contrast-enhanced medical 3D image is shown in Fig. 5. The VR software CalVR [25] that we integrated volkit into comes with a GLSL implementation of CLAHE [16]. We found our CUDA version to be slightly faster than CalVR’s implementation (the difference in performance we attribute to implementation details regarding how CalVR executes compute shaders via scene graph nodes), where the performance for volumes of $256^3$ cells would be on the order of 30 ms. The CPU version we found to be about one to two orders of magnitude slower than the CUDA implementation.

4.2 Case Study: Exploring Large Simulation Data with CalVR

Our second case study is comprised of integrating volkit into the virtual reality software CalVR [25], which is targeted towards large back projection and tiled display systems, but also towards VR headsets. CalVR comes with a volume rendering plugin that we extended with a zoom interaction for AMR volume exploration. To achieve frame rates high enough for VR, we don’t render the AMR volume directly, but instead resample it on a structured grid in an interactive preprocess. Initially, we resample the AMR volume to a structured volume based on a user-defined target number of cells—for our experiments we used 50 M. The user then specifies a region of interest (ROI) to zoom in on (cf. Fig. 5). The volume is cropped accordingly using the Crop algorithm, giving us a smaller AMR volume, which we resample using Resample and the same number of target cells from before. This effectively allows us to zoom in on the volume and by that to increase the resolution inside the ROI. We experimented with the data set from [26] depicted in Figs. 2, 5 and 6, whose logical grid size is $4K \times 4K \times 80K$ and found the zoom interaction to always take on the order of below one second.

4.3 Performance

We present performance numbers for some of the algorithms implemented in volkit. Our main motivation here is not to evaluate the individual algorithms—we chose an assortment of algorithms that, due to their characteristics will scale more or less well—but rather to prove that our design decisions—deferred API, data management layer, choice of programming environment, etc., don’t severely impact overall performance and scalability.

We use Resample to scale down the structured volumes shown in Table 1 by a factor of two in each dimension; Resample to convert an AMR volume (also Table 1) to structured volumes of different sizes; FillRange to fill the
Fig. 6. Single-threaded, multi-threaded (16x, with OpenMP) and GPU results for an assortment of algorithms implemented in volkit. Note the logarithmical scale to make the scalability of the CPU and GPU results comparable.

whole volume with a uniform value; ApplyFilter to apply Gaussian blur to the structured volumes; Crop with with a sliding window and ROIs of 20%, 40%, 60%, and 80% of the original AMR volume’s size; and Flip to mirror the structured volumes along the axis where their extents are longest. We ran single-threaded and multi-threaded tests on a 16 core Intel Xeon CPU with 2.2 GHz that is equipped with 128 GB DDR memory and an NVIDIA Quadro RTX 8000 GPU with 12 GB of GDDR memory. Results are presented in Fig. 6. We observe that the GPU outperforms the CPU for structured volumes due to superior memory bandwidth. For AMR volumes we see mixed results—this is likely due to the fact that most of our algorithms parallelize over subgrids, so that the problem is less balanced and exposes less parallelism.

5 Conclusions

We presented the performance-portable 3D volume manipulation library volkit targeted at large volumes that are for example generated by simulation codes. In contrast to other libraries, volkit focuses on particular volume representations such as for example hierarchical / AMR data. We presented volkit’s overall system design, as well as case studies and evaluations. Finally, volkit is open source and can be downloaded from [https://github.com/volkit/volkit](https://github.com/volkit/volkit).

Acknowledgments

This research was funded in part by a grant from The Leona M. and Harry B. Helmsley Charitable Trust. The SILCC AMR data set is courtesy Daniel Seifried. The structured volume data sets are hosted online by Pavol Klacansky under [https://klacansky.com/open-scivis-datasets/](https://klacansky.com/open-scivis-datasets/).
References

1. Intel Open Volume Kernel Library. [https://www.openvkl.org](https://www.openvkl.org) accessed: Aug 3, 2021
2. Beazley, D.M.: SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4. p. 15. TCTK’96, USENIX Association, USA (1996)
3. Berger, M.J., Colella, P.: Local adaptive mesh refinement for shock hydrodynamics. Journal of Computational Physics 82(1) (1989)
4. Berger, M.J., Oliger, J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. Journal of Computational Physics (1984)
5. Bradski, G., Kaehler, A.: Learning OpenCV: Computer Vision in C++ with the OpenCV Library. O’Reilly Media, Inc., 2nd edn. (2013)
6. Burstedde, C., Wilcox, L.C., Ghattas, O.: p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. SIAM Journal on Scientific Computing 33(3) (2011). [https://doi.org/10.1137/100791634](https://doi.org/10.1137/100791634)
7. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Truran, J.W., Tufo, H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. The Astrophysical Journal Supplement Series (2000)
8. Hadwiger, M., Al-Awami, A.K., Beyer, J., Agos, M., Pfister, H.: SparseLeap: Efficient empty space skipping for large-scale volume rendering. IEEE Transactions on Visualization and Computer Graphics (2018)
9. Hadwiger, M., Kniss, J.M., Rezk-Salama, C., Weiskopf, D., Engel, K.: Real-time Volume Graphics. A. K. Peters, Ltd., Natick, MA, USA (2006)
10. Johnson, H.J., McCormick, M.M., Ibanez, L.: The ITK Software Guide Book 1: Introduction and Development Guidelines - Volume 1. Kitware, Inc., Clifton Park, NY, USA (2015)
11. Kikinis, R., Pieper, S.D., Vosburgh, K.G.: 3D Slicer: A Platform for Subject-Specific Image Analysis, Visualization, and Clinical Support, pp. 277-289. Springer New York, New York, NY (2014). [https://doi.org/10.1007/978-1-4614-7657-3_19](https://doi.org/10.1007/978-1-4614-7657-3_19)
12. Kroes, T., Post, F.H., Botha, C.P.: Exposure render: an interactive photorealistic volume rendering framework. PLoS ONE 7(7) (07 2012), 10.1371/journal.pone.0038586
13. Lee, M., Malaya, N., Moser, R.D.: Petascale direct numerical simulation of turbulent channel flow on up to 786k cores. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC ’13, Association for Computing Machinery, New York, NY, USA (2013)
14. Lee, M., Moser, R.D.: Direct numerical simulation of turbulent channel flow up to \( Re \approx 5200 \). Journal of Fluid Mechanics 774, 395–415 (2015)
15. Levoy, M.: Display of Surfaces from Volume Data. IEEE Computer Graphics and Applications (1988)
16. Lucknavalai, K., Schulze, J.P.: Real-time contrast enhancement for 3d medical images using histogram equalization. Advances in Visual Computing (2020)
17. Max, N.: Optical models for direct volume rendering. IEEE Transactions on Visualization and Computer Graphics 1(2), 99–108 (Jun 1995)
18. Morrical, N., Wald, I., Usher, W., Pascucci, V.: RTX Beyond Ray Tracing: Extending the Use of Hardware Ray Tracing Cores for Unstructured Mesh Point Location. Transactions on Visualization and Computer Graphics (2020)
19. Muigg, P., Hadwiger, M., Doleisch, H., Gröller, E.: Interactive Volume Visualization of General Polyhedral Grids. IEEE Transactions on Visualization and Computer Graphics (2011)
20. Parker, S.G., Bigler, J., Dietrich, A., Friedrich, H., Hobrock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., Stich, M.: OptiX: A general purpose ray tracing engine. ACM Trans. Graph. 29(4), 66:1–66:13 (Jul 2010)
21. Pizer, S.M., Amburn, E.P., Austin, J.D., Cromartie, R., Geselowitz, A., Greer, T., ter Haar Romeny, B., Zimmerman, J.B., Zuiderveld, K.: Adaptive histogram equalization and its variations. Computer Vision, Graphics, and Image Processing 39(3), 355–368 (1987)
22. Raab, M.: Ray Tracing Inhomogeneous Volumes, pp. 521–531. Apress, Berkeley, CA (2019). https://doi.org/10.1007/978-1-4842-4427-2_28
23. Sahistan, A., Demirci, S., Morrical, N., Zellmann, S., Aman, A., Wald, I., Güdükbay, U.: Ray-traced Shell Traversal of Tetrahedral Meshes for Direct Volume Visualization. In: 2021 IEEE Visualization Conference (VIS) (to appear). pp. 1–5 (Oct 2021)
24. Schroeder, W.J., Martin, K.M.: 1 - overview of visualization. In: Hansen, C.D., Johnson, C.R. (eds.) Visualization Handbook, pp. 3–35. Butterworth-Heinemann, Burlington (2005)
25. Schulze, J.P., Prudhomme, A., Weber, P., DeFanti, T.A.: CaVR: an advanced open source virtual reality software framework. In: Dolinsky, M., McDowall, I.E. (eds.) The Engineering Reality of Virtual Reality 2013. vol. 8649, pp. 1 – 8. International Society for Optics and Photonics, SPIE (2013)
26. Seifried, D., Walch, S., Girichidis, P., Naab, T., Wünsch, R., Klessen, R.S., Glover, S.C.O., Peters, T., Clark, P.: SILCC-Zoom: the dynamic and chemical evolution of molecular clouds. mnras 472(4), 4797–4818 (Dec 2017)
27. Wald, I., Morrical, N., Haines, E.: OWL – The Optix 7 Wrapper Library (2020), https://github.com/owl-project/owl
28. Wald, I., Zellmann, S., Morrical, N.: A Memory Efficient Encoding for Ray Tracing Large Unstructured Data. IEEE Transactions on Visualization and Computer Graphics (to appear) pp. 1–10 (2022)
29. Wald, I., Zellmann, S., Morrical, N.: Faster RTX-Accelerated Empty Space Skipping using Triangulated Active Region Boundary Geometry. In: Larsen, M., Sadlo, F. (eds.) Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association (2021). https://doi.org/10.2312/pgv.20211042
30. Wald, I., Zellmann, S., Usher, W., Morrical, N., Lang, U., Pascucci, V.: Ray Tracing Structured AMR Data Using ExaBricks. IEEE Transactions on Visualization and Computer Graphics 27(2), 625–634 (2021)
31. Wang, F., Wald, I., Wu, Q., Usher, W., Johnson, C.R.: CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data. IEEE Transactions on Visualization and Computer Graphics (2019)
32. Zellmann, S., Schulze, J.P., Lang, U.: Binned k-d tree construction for sparse volume data on multi-core and GPU systems. IEEE Transactions on Visualization and Computer Graphics pp. 1–1 (2019). https://doi.org/10.1109/TVCG.2019.2938957
33. Zellmann, S., Schulze, J.P., Lang, U.: Rapid k-d tree construction for sparse volume data. In: Childs, H., Cucchietti, F. (eds.) Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association (2018). https://doi.org/10.2312/pgv.20181097
34. Zellmann, S., Wickeroth, D., Lang, U.: Visionaray: A cross-platform ray tracing template library. In: Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS 2017). IEEE (2017)