Challenges and Opportunities using Software-defined Visualization in MegaMol

Tobias Rau, Michael Krone, Guido Reina, and Thomas Ertl
Visualization Research Center, University of Stuttgart (VISUS)
Allmandring 19, 70569 Stuttgart, Germany
Email: {firstname}.{lastname}@visus.uni-stuttgart.de

***Abstract*—In this paper we describe how we integrated the OSPRay ray tracing engine into the traditionally GPU-centric MegaMol visualization framework. Since OSPRay is purely CPU-based, this adds software-defined visualization to MegaMol. This enables us to use MegaMol for in-situ rendering on HPC systems that lack GPUs. Furthermore, we designed the integration so that the new OSPRay rendering can be used alongside the classical OpenGL-based rendering. We describe how the ray tracing paradigm, where the whole scene has to be available during rendering, changes the module graph of MegaMol. The performance of the OSPRay ray tracing is shown to be at least competitive with classical GPU-accelerated rendering methods for particle rendering available in MegaMol.**

I. Introduction

Visualization is an established means for analyzing and exploring scientific data. Although the rendering capabilities of modern desktop PC are constantly improving, the disconnect between performance and data set sizes is increasing. One obvious remedy is to use High-Performance Compute clusters for data visualization. As pointed out by Moreland et al. [1] visualization in the HPC world is currently experiencing significant changes due to the constantly rising number of nodes and cores, leading to an increasing degree of parallelism. Future supercomputers are expected to exceed the exaflop scale. Such HPC systems are able to run simulations that generate data sets that are already problematic from the storage perspective. One prominent example are cosmological simulations like the DarkSky dark matter simulations where each time step can exceed 30 TB [2]. Such data sets make traditional postmortem visualization approaches unfeasible and call for a paradigm shift to in-situ visualization.

MegaMol is a framework for scientific visualization development that has been successfully applied to many problems including large particle data sets. It offers a thin abstraction layer on top of OpenGL and is primarily tailored to running on a normal desktop PC equipped with a high-performance graphics card. MegaMol is designed to be used for postmortem visualization of data sets that fit into the main memory of a desktop PC. Due to it's GPU-centric nature, it outperforms several well-known applications in its intended scenario [3]. However, MegaMol offers only rudimentary support for in-situ visualization, since it requires a GPU. Looking at the Top 100 list of HPC systems shows that less than 25 % of these systems are equipped with GPUs (and only two systems in the Top 10 have GPUs). Therefore, adding *software-defined visualization* that does not rely on high-performance GPUs for rendering to MegaMol would be a feasible solution to enable HPC rendering and in-situ visualization. Software-defined visualization (http://www.sdvis.org/) is an open source initiative started by Intel to improve performance and efficiency of prominent visualization solutions by using CPU-based ray tracing.

Implementing a CPU ray tracing engine into an OpenGL framework is not straightforward. The MegaMol rendering paradigm implements separate renderer for each type of geometry (e.g., spheres, streamlines, volumes) and data sets. Using the OpenGL depth buffer, combining the rendering results requires no explicit handling, as long as all geometry is opaque. For ray tracing, the complete scene has to be passed to the renderer to be able to compute a global spatial acceleration structure, which is necessary for fast ray-object intersection tests. Conversely, such a global view of all data allows the visualization to use global illumination methods like ambient occlusion [4] or path tracing that highlight the shape and spatial arrangement of the displayed object(s). This cannot be obtained straightforwardly using OpenGL, due to the inherently local paradigm of the rasterization. Specialized approaches for specific problems, for example ambient occlusion for molecular data, have been presented (e.g., [5]–[7]).

Using the Open source, Scalable, and Portable Ray tracing engine (OSPRay) [8] as renderer brings two major differences to classical GPU-based rendering: The rasterizer is replaced by a ray tracer, which is executed on a multi-core CPU. Computing the visualization on the CPU has the additional benefit that access to the full memory of the machine is granted, which allows for loading much larger data sets that would not fit into the more limited GPU memory anymore. Software-defined visualization is also beneficial for *in-situ visualization*, where the simulation and the visualization run in parallel on an HPC system.

OSPRay offers rendering routines for typical input data types, for example volumes, polygon meshes, and non-polygonal geometry like implicitly defined spheres or cylinders. It builds upon the fast *Embree* ray tracing kernels [9], which are highly optimized for *Single Instruction, Multiple Data* (SIMD) execution. Here, the same instructions are executed by multiple threads on individual data simultaneously. Implicit geometry is also used for particle visualization in the MegaMol framework [3] using OpenGL GLSL shaders [10], [11]. Consequently, the most commonly used visualization

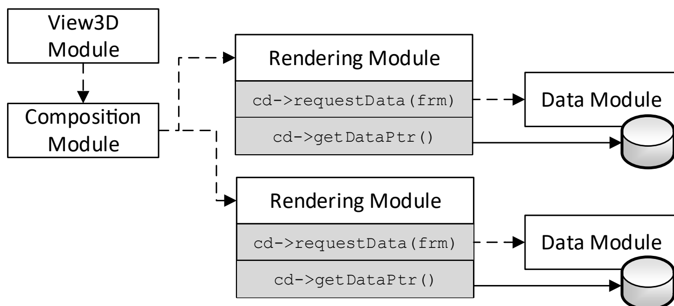

Fig. 1. Classical MegaMol module graph with OpenGL rendering modules. In this picture, the data call is annotated as *cd*. The rendering module requests a frame (*frm*) and the call offers a pointer to the requested frame data.

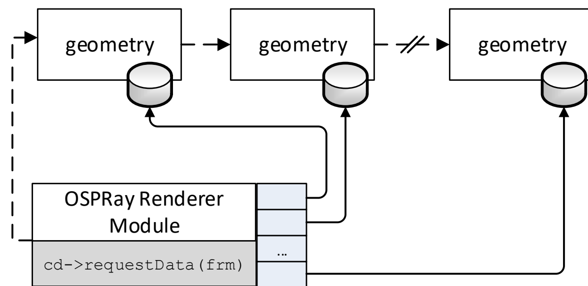

Fig. 2. Module connection and communication model of the OSPRay implementation. The rendering module requests a frame and geometry modules write the pointer to the requested data via the call into a container inside the rendering module. This method allows for infinite stacking of modules.

data in MegaMol can be used straightforwardly in OSPRay. The OSPRay engine is already used by other scientific visualization software like ParaView [12] or VMD [13], [14].

An alternative to using OSPRay would have been the *VTK-m* [1] visualization toolkit, which is tailored to HPC execution and provides a set of data parallel primitives. Here, a set of simple algorithms act as an abstraction layer between the hardware architecture and the visualization code. These algorithms are optimized for extreme parallelism by taking advantage of SIMD execution and node-level parallelism. However, this would require to rewrite all our rendering code from scratch in contrast to using OSPRay as mentioned above.

II. MegaMol 101

MegaMol is designed as a modular, rapid prototyping framework tailored to scientific visualization written in C++. Functionality is encapsulated in *modules* that communicate via strongly-typed channels termed *calls*. A running MegaMol instance consists of an arbitrary number of interconnected modules that form a graph. This concept allows to reuse components and recombine them to solve new challenges. A minimal MegaMol module graph consists of a data source module, a rendering module, a view module that represents the window and the corresponding calls to connect these modules. Module slots define compatible calls that are able to connect modules at runtime. MegaMol control flow follows the pull paradigm in the sense that the window starts polling all connected modules for updates. However, a call also optionally transports parameters or state variables to the called module, which allows for bidirectional communication (push-pull). Figure 1 shows an exemplary module graph with two renderers that have their individual data providers. Each module called by the view can in turn issue an arbitrary number of calls to its downstream connected modules. For example, a renderer module called for an updated image will in turn call the data provider for the data that should be rendered, for example one particular frame of a particle simulation. If the requested frame was not yet loaded, the data module loads this frame from disk. The renderer can then access the data via a pointer provided by the data call. As a result, there is no need for duplicating any data in memory during the whole visualization pipeline. For more details, please refer to Grottel et al. [3].

III. Integration of OSPRay into MegaMol

The major difference between OSPRay and MegaMol is that OSPRay gathers the data for entire scene to be rendered, as mentioned in the Introduction. MegaMol rendering modules, on the other hand, are self-contained in a way that each module renders its content independently of other modules at the same module graph depth. That is, multiple different opaque renderable objects can be drawn by different renderers that have no need of communicating with each other, since OpenGL will take care of depth compositing by design and most MegaMol renderers only use local lighting. Since OSPRay implements ray tracing, it always operates on the whole scene. That is, OSPRay conceptually doesn't fit well to the MegaMols architecture. Consequently, the integration of OSPRay into MegaMol necessitates a mechanism to collect all data prior to passing it to OSPRay and triggering the rendering.

The new way of communicating data to the rendering module is to pass all data pointers directly to the OSPRay rendering module, which appends it to a temporary list per rendered frame. This offers a significant advantage over the original MegaMol paradigm, since there are no limitations of different data sets and data types for a single rendering module. We added *OSPRay geometry modules* instead of classical renderers. These geometry modules only collect the data, transform it into a renderable representation, and pass it to the OSPRay rendering module, which exists only once. Upon being called, the first geometry module will get the interface for adding data from the upstream OSPRay renderer and use it to add its own data. It will then call its downstream geometry module and pass the interface for adding more data. This results in a pattern that allows to daisy-chain an arbitrary number of geometry modules. In Figure 2, a schematic of this new data handling paradigm is shown. Note that light modules use the same mechanism as geometry modules. We decided to encapsulate the concrete geometry list and light list of OSPRay to ensure that no module can modify or delete data that has been added by other modules.

The information put into the geometry list of the rendering module is also used to calculate the union of individual bounding boxes for all included objects, resulting in a new bounding box that encloses the entire scene. This bounding

box is used to compute the initial camera setup so that the whole scene is visible.

IV. Results and Discussion

A. MegaMol Configuration

Figure 3 shows a module graph within the MegaMol configurator. The module graph includes OSPRay modules and shows the connection scheme as explained in section III. The View3D module requests an image via the CallRender3D that triggers the OSPRayRenderer to request data via the `CallOSPRayStructure` and `CallOSPRayLight` calls. Each OSPRay geometry module (e.g. `OSPRaySphereGeometry`, `OSPRayTriangleMesh`) has a connection to a data source. In the case of a `OSPRaySphereGeometry`, a `MultiparticleDataCall` is connected that offers a universal interface for particle data. Other modifications of the displayed data are available via additional modules like the `OSPRayOBJMaterial` or the `LinearTransferFunction`. The OSPRay API is loaded only into the `OSPRayRenderer` module for two reasons: the first one is the ability to retain existing data management modules; the second one is the ability to handle the data management inside the geometry modules independent from the OSPRay API. This makes it easy to write additional geometry modules even in a completely separate plugin of MegaMol.

B. Image Quality and Performance

Ray tracing typically offers higher image quality than OpenGL rendering due to the global illumination, which is usually only partially approximated when using OpenGL. Figure 4 shows a direct comparison of the new OPSRay rendering in MegaMol and a traditional MegaMol OpenGL renderer that uses ambient occlusion to achieve similar image quality. The data set is a laser ablation simulation containing 199,940,704 atoms, which has a size of 2.97 GB for one frame. Both methods render each atom as a sphere while the image resolution was set to 1920×1200. The OpenGL module uses fast GPU-based sphere ray casting [10] with local lighting. The OSPRay module uses the *scivis* rendering method provided by the OSPRay library, and was configured to simple ray casting using a distant light (local lighting). MegaMol achieves 2.9 frames per second (fps) on an Nvidia Geforce GTX 1060 and OSPRay reaches 14.3 fps on an Intel i7-6700 (4×3.4 GHz). The higher rendering speed of OSPRay is expected since the OpenGL renderer has to rasterize each sphere and generate one ray per fragment while the ray tracing only generates one primary ray per pixel and can terminate the rays on the first hit.

We also measured the performance using ambient occlusion, since this technique gives valuable depth cues for large particle data sets. In case of the OpenGL-based rendering, we used the grid-based ambient occlusion approximation by Staib et al. [6]. OSPRay's *scivis* was configured to compute secondary rays for ambient occlusion. The OpenGL rendering runs at approximately 1.6 fps. The image quality and rendering speed of OSPRay depends on the number of rays cast per pixel. The performance of a single ray per pixel is approximately 7.6 fps. However, a comparable image quality can only be reached at a minimum of 10 rays per pixel, which results in a frame rate drop to about 2 fps.

C. Adaptive Rendering

For better performance during interaction, the user can configure the rendering module to use the adaptive rendering provided by OSPRay. As long as the camera parameters or other parameters that trigger a recalculation of the image are not changed, more and more rays are cast into the already rendered scene. This improves the image quality over time. While the user is interacting with the visualization—e.g., by moving the camera—the OSPRay rendering module is only casting one ray per pixel, resulting in smooth interaction. For rendering a still image or a movie, however, the adaptive sampling can be deactivated to get a consistent image quality during the tracking shot, which is then computed offline.

Activating the adaptive rendering for the 199.9 M atom data set used for performance measurements retains the 7.6 fps and the full image quality shown in Figure 4 is reached after 10 interations ($\sim$1.4 seconds).

D. Ray Tracing Performance and Scaling

OSPRay makes heavy use of Intel's single program multiple data (SPMD) compiler ISPC [15] and the Embree ray tracing kernels [9]. ISPC optimizes code for the CPU vector unit. That way, a SPMD code can efficiently run on the SIMD units available on modern multi-core CPUs. While ISPC takes care of the rendering and shading, Embree is used to build and traverse the spatial acceleration structures, for example hierarchical scene graphs [8]. Besides, Embree also takes advantage of SIMD vectorization using ISPC.

The most efficient vector extensions available on the current hardware are automatically chosen during runtime, which makes OSPRay a highly portable library. We compared the performance of OSPRay on an Intel Xeon Phi 7210 2nd generation (Knights Landing, KNL) and an Intel i7-6700. The Xeon Phi represents a highly parallel vector instruction machine, while the i7 represents a typical desktop workstation CPU the average MegaMol user is using. For the test, we rendered a 16k image of a medium occupied scene on both systems with an increasing number of rays per pixel. The result of this performance test is shown in Figure 5. For a low number of rays per pixel, the i7 performs as good as the KNL, however, with an increasing number on rays, the KNL scales significantly better than the i7.

V. Conclusion and Future Work

We presented the implementation of the CPU ray tracing engine OSPRay into the visualization framework MegaMol. Because of the software-defined aspect provided by OSPRay, MegaMol can now be used for scientific visualization on HPC systems that lack GPUs. The integration of OSPRay required a rethinking of MegaMol's data distribution paradigm to be able to take advantage of all features OSPRay offers. The new

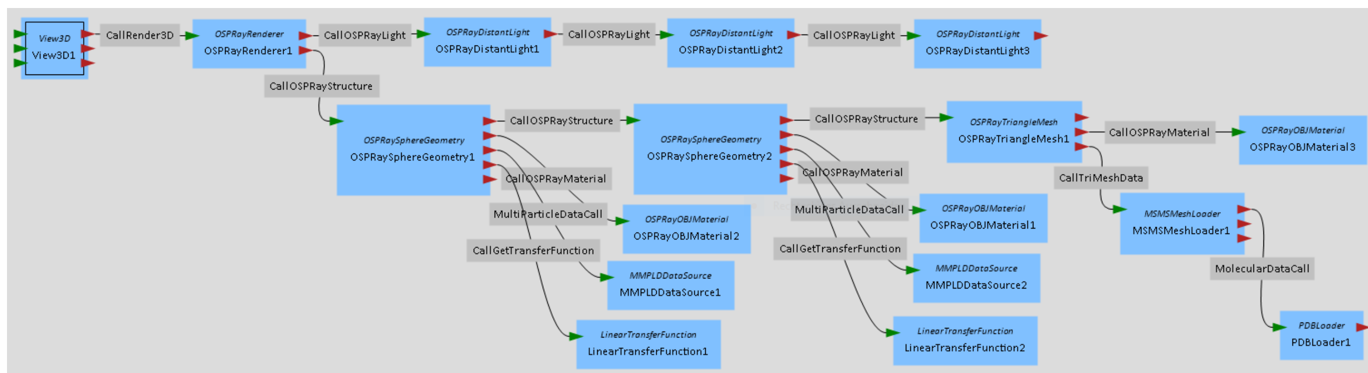

Fig. 3. Simple OSPRay module graph generated with the configuration tool provided by MegaMol. The `OSPRayRenderer` module provides two connections for the new module communication paradigm: one for lights and one for geometry.

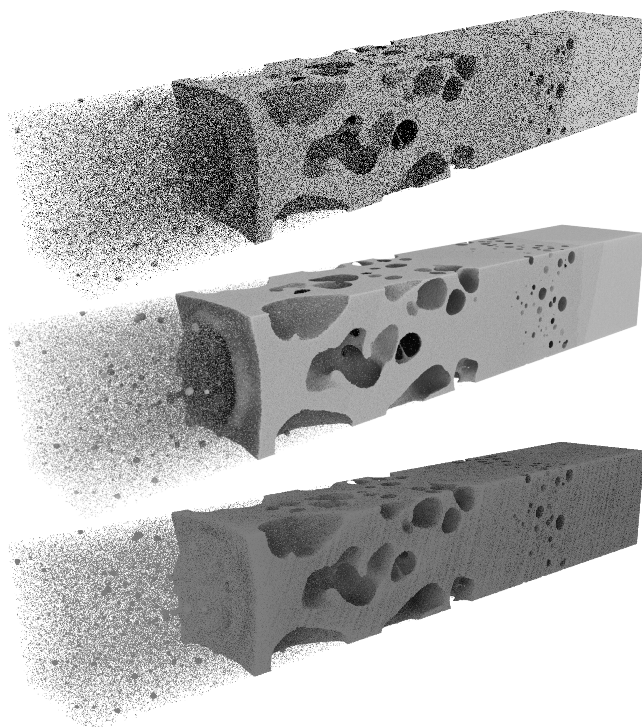

Fig. 4. Visualization of a laser ablation simulation of aluminum. Each of the 199.9 M atoms is rendered as a sphere. The top image shows a ray tracing obtained using the OSPRay sphere geometry and the built-in scivis renderer with one ray per pixel. The center image shows the full quality provided by OSPRay after convergence. The bottom image shows the results of a MegaMol sphere rendering module with approximate ambient occlusion lighting implemented in OpenGL.

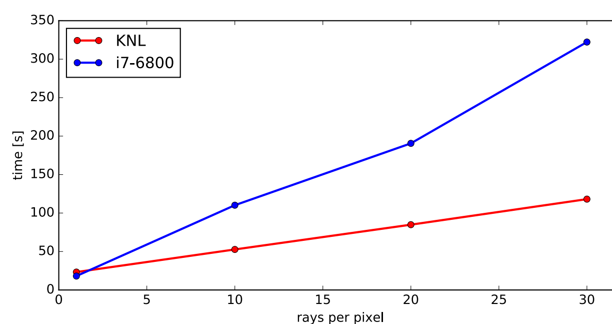

Fig. 5. Performance while rendering a 16k image with an Intel i7-6700 and the Intel Xeon Phi 7210 (Knights Landing, denoted as *KNL*).

daisy-chaining paradigm extends the established module graph of MegaMol so that is has become more versatile by making the whole scene available for rendering. The established OpenGL renderers can still be used in concert with the new OSPRay renderer. To achieve this, the result of the OSPRay renderer has to be copied into an OpenGL framebuffer object with depth information enabled. Due to the depth test, all geometry rendered to this framebuffer using OpenGL will be correctly composited with the OSPRay content.

Currently, MegaMol has more different rendering capabilities than OSPRay (e.g., protein surface visualization [16]). In the future, we want to extend OSPRay by such functionality.

As mentioned above, the integration of OSPRay into MegaMol is a first step to perform in-situ visualizations on HPC clusters without GPUs. This also requires a distributed execution of MegaMol on multiple nodes, where each MegaMol instance renders the data available on the same node. In a final pass, all individual images have to composited using color and depth information. Note that this is currently only possible if local lighting is used and no node-spanning visualization primitives like streamlines are generated. This is a common problem of distributed visualization that uses global information [17]. Thus, as a next step, we want to extend our implementation to be able to share data across domains via MPI to enable such global effects. For fast rendering, OSPRay can be already executed in parallel on multiple nodes using MPI. This is especially useful if the cluster offers more nodes than are required for the simulation.

Acknowledgments

This research was partially supported by the Intel®Parallel Computing Center program, by the German Research Foundation (DFG) as part of SFB 716, and by BW Stiftung as part of project "Digital Human". We also want to thank Johannes Roth for providing the laser ablation data.

REFERENCES

[1] K. Moreland, C. Sewell, W. Usher, L. t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K. L. Ma, H. Childs, M. Larsen, C. M. Chen, R. Maynard, and B. Geveci, "VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures," *IEEE Computer Graphics and Applications*, vol. 36, no. 3, pp. 48–58, May 2016.

[2] S. W. Skillman, M. S. Warren, M. J. Turk, R. H. Wechsler, D. E. Holz, and P. M. Sutter, "Dark Sky Simulations: Early Data Release," *arXiv:1407.2600 [astro-ph]*, Jul. 2014.

[3] S. Grottel, M. Krone, C. Müller, G. Reina, and T. Ertl, "MegaMol – A Prototyping Framework for Particle-Based Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, no. 2, pp. 201–214, Feb. 2015.

[4] S. Zhukov, A. Iones, and G. Kronin, "An Ambient Light Illumination Model," in *Eurographics Workshop on Rendering*, ser. Eurographics, G. Drettakis and N. Max, Eds. Springer, 1998, pp. 45–55.

[5] M. Tarini, P. Cignoni, and C. Montani, "Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1237–1244, 2006.

[6] J. Staib, S. Grottel, and S. Gumhold, "Visualization of Particle-based Data with Transparency and Ambient Occlusion," *Computer Graphics Forum*, vol. 34, no. 3, pp. 151–160, 2015.

[7] P. Hermosilla, V. Guallar, A. Vinacua, and P. P. Vázquez, "High quality illustrative effects for molecular rendering," *Computers & Graphics*, vol. 54, pp. 113–120, 2016.

[8] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil, "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 931–940, Jan. 2017.

[9] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A Kernel Framework for Efficient CPU Ray Tracing," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 143:1–143:8, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2601097.2601199

[10] G. Reina and T. Ertl, "Hardware-Accelerated Glyphs for Mono- and Dipoles in Molecular Dynamics Visualization," in *EG/IEEE VGTC Symposium on Visualization*, 2005, pp. 177–182.

[11] S. Grottel, G. Reina, and T. Ertl, "Optimized Data Transfer for Time-dependent, GPU-based Glyphs," in *IEEE Pacific Visualization Symposium*, 2009, pp. 65–72.

[12] J. Ahrens, B. Geveci, and C. Law, "ParaView: An End-User Tool for Large-Data Visualization," in *Visualization Handbook*, C. D. Hansen and C. R. Johnson, Eds. Butterworth-Heinemann, 2005, pp. 717 – 731.

[13] W. Humphrey, A. Dalke, and K. Schulten, "VMD: Visual Molecular Dynamics," *J. Mol. Graph.*, vol. 14, pp. 33–38, 1996.

[14] J. E. Stone, W. R. Sherman, and K. Schulten, "Immersive Molecular Visualization with Omnidirectional Stereoscopic Ray Tracing and Remote Rendering," in *IEEE International Parallel and Distributed Processing Symposium Workshop*, 2016.

[15] M. Pharr and W. R. Mark, "ISPC: A SPMD compiler for high-performance CPU programming," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–13.

[16] M. Krone, K. Bidmon, and T. Ertl, "Interactive Visualization of Molecular Surface Dynamics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1391–1398, 2009.

[17] W. Usher, I. Wald, A. Knoll, M. Papka, and V. Pascucci, "In Situ Exploration of Particle Simulations with CPU Ray Tracing," *Supercomputing Frontiers and Innovations*, vol. 3, no. 4, pp. 4–18, Oct. 2016. [Online]. Available: http://superfri.org/superfri/article/view/112