Eur. Phys. J. Special Topics **227**, 1817–1829 (2019)

https://doi.org/10.1140/epjst/e2019-800167-5

THE EUROPEAN PHYSICAL JOURNAL SPECIAL TOPICS

Regular Article

MegaMol – a comprehensive prototyping framework for visualizations

Patrick Gralka[1,a], Michael Becher[1], Matthias Braun[1], Florian Frieß[1], Christoph Müller[1], Tobias Rau[1], Karsten Schatz[1], Christoph Schulz[1], Michael Krone[2], Guido Reina[1], and Thomas Ertl[1]

[1] Visualization Research Center, University of Stuttgart, Allmandring 19, 70569 Stuttgart, Germany
[2] Big Data Visual Analytics in Life Sciences, University of Tübingen, Sand 14, 72076 Tübingen, Germany

Received 4 October 2018 / Received in final form 3 December 2018
Published online 8 March 2019

Abstract. We present MegaMol, a low-overhead prototyping framework for interactive visualization of large scientific data sets. We give a brief summary of related work for context and then focus on a comprehensive overview of the core architecture of the framework. This is followed by the existing and novel features and techniques in MegaMol that define its current functionality. MegaMol has originally been developed to support the visualization and analysis of particle-based data sets that, for instance, come from molecular dynamics simulations. Meanwhile, the software has evolved beyond that. New algorithms and techniques have been implemented to handle many diverse tasks, including information visualization. Additionally, improvements have been made on the software engineering side to make MegaMol more accessible for domain scientists, like an easy-to-handle scripting interface.

1 Introduction

The advancement of techniques and methods in scientific visualization often leads to proof-of-concept prototypes. Specialized analysis tools often start as small prototypes that are shared with domain experts so they can use them to solve their problems and provide feedback that helps to further improve the tool. Usually, all of these prototypes have a lot in common. They all include the same code for initializing a graphical context, for file I/O, for data pre-processing, user interaction, etc. It is an advantage to have a framework that supports all common processing steps, so one can focus on developing new algorithms. The joint development and use of a framework creates synergies, since it can be used in different projects and all common functions have to be developed only once. Having these advantages in mind, the development of MegaMol has started in 2006 in the context of the Collaborative Research Center 716. This research center brought together scientists from physics, material science, chemistry, biology, and computer science. Its goal was to advance technologies regarding particle-based simulations and so the initial focus of MegaMol

[a] e-mail: patrick.gralka@visus.uni-stuttgart.de

was to support particle-based visualizations. This focus has gradually been broadened over time. But the core paradigms that govern the development of MegaMol remain valid until now. We strive for an efficient usage of all available compute resources including CPU and GPU. The processing pipeline is adjusted to achieve scalability. The programming model is organized in a modular scheme to be highly adaptable. Modularization also allows re-use of existing algorithms in different contexts, and the duality of a stable core framework and cutting-edge (and partially experimental) algorithmic modules represent the core idea behind MegaMol.

In this paper, we extend upon the overview on MegaMol given by [1–3] by presenting new features and improvements to MegaMol that are interesting to this audience.

2 Related frameworks

We briefly introduce the most relevant visualization frameworks available. They are mostly more comprehensive and general than MegaMol, but conversely cannot be tailored as much toward low overhead.

2.1 OSPRay

OSPRay [4] is a framework for ray-tracing based visualization of particles, volumes, meshes, and more. Its kernels are optimized for CPUs to enable its usage on workstations and arbitrary HPC clusters, for instance, for in situ rendering purposes. The framework can run independently but is also embeddable into other visualization frameworks. While it contains a rather flexible data visualization tool, it lacks data processing and analysis functionality in comparison to MegaMol. However, it excels in rendering performance in scenarios where data sets do not fit entirely into the GPU memory. The main advantage of OSPRay is its scalability. Especially, considering extensions such as the *Pkd* tree [5], which scales the rendering to billions of spheres for point-based data without adding overhead to the source data. A sphere is the main glyph type in scientific visualization for the rendering of molecules, and is also used for astrophysical data or laser scans. OSPRay provides a simple API to extend its functionality, for instance, to enable the rendering of new glyph types. Performance-critical code portions are written in *ispc* [6] in order to utilize the vector extensions of modern CPUs. The ispc language is in principle similar to shader languages used to program GPUs.

2.2 VTK

The Visualization Toolkit (VTK) [7] is a library that contains algorithms for the scientific visualization of scalar, vector, and tensor fields. Different visualization frameworks rely on VTK, also for their internal data representation. ParaView [8] is such a framework that provides VTK functionality in a comprehensive application with GUI. It supports the distribution of work on clusters using data parallelism. VisIt [9] is another framework built on VTK. Its architecture is based on a client-server model. This allows for parallelization in a remote rendering scenario of large data. Both frameworks can be integrated into (or coupled with) simulation software for in situ visualization.

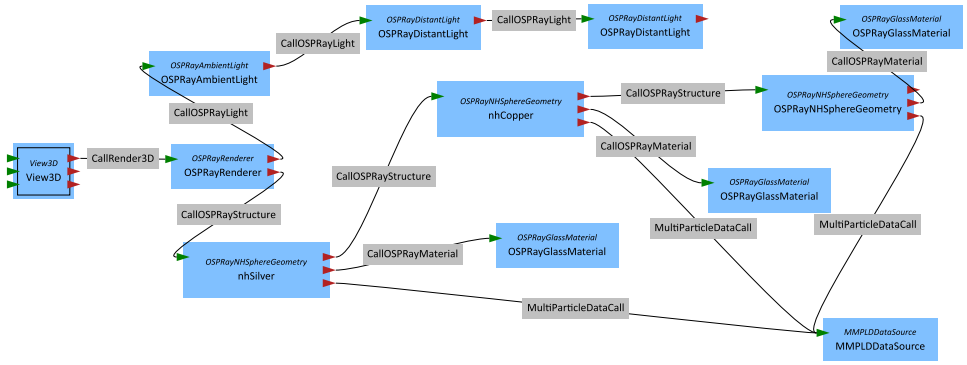

Fig. 1. MegaMol graph demonstrating the *daisy chaining* paradigm (see Sect. 4.3) that combines several geometries for rendering in the singleton OSPRayRenderer.

2.3 OVITO

The Open Visualization Tool (OVITO) [10] is a GUI-based framework for the visualization of atomistic data from, e.g. molecular dynamics. It provides a variety of rendering and processing methods that can be connected into a "data processing pipeline". The processing pipeline, as well as the extension interface and the Python-based scripting interface, aim at providing flexibility to users and developers, which is similar to MegaMol. It offers specific analysis functionality for material science, like dislocation detection and visualization.

2.4 Biomolecular visualization frameworks

Besides the more general visualization tools, there exist more specialized frameworks specifically designed for biomolecular visualization. Among these, the most established ones are VMD [11], Chimera [12], and PyMOL [13]. They all provide general visualization methods for biomolecules like proteins or DNA, although each of them has a different application area and, therefore, a different user group in focus. PyMol incorporates a variety of tools to analyze the data produced by crystallographers, like electron density maps, for example. In this aspect, Chimera acts similar as PyMol. Both offer similar possibilities to analyze small molecular interactions like the interaction of ligand molecules at a binding site. VMD on the other hand is designed towards the analysis of larger molecular trajectories. Especially when combined with the Molecular Dynamics software NAMD [14] it is able to perform not only visualization but also molecular modeling tasks due to the direct communication between the simulation and visualization. Opposed to the other two tools, VMD also has the capability of being executed on HPC systems. All three programs offer a Python-based scripting interface to allow more flexibility for the user.

3 Overview

MegaMol is designed to be primarily a prototyping framework for visualization techniques and visual analysis applications. To achieve that, the design choices adhere to paradigms, such as flexibility, re-usability, and zero-copy data management. The backbone of MegaMol is represented by the *Module graph* (see Fig. 1). It is an acyclic graph of functional entities as nodes including at least a data source and a data sink.

Note that a window on the screen depicting a rendered image is a data sink, as well. Edges in the graph represent data or information transported between nodes, usually bi-directionally. However, the initialization of data transport follows the pull paradigm. That means that the data sink always requests data from the source. That choice aims at the support for dynamic data sets and at the same time at the reduction of data updates to the minimum. The primary results of MegaMol are interactive visualizations. Therefore, data should only traverse the Module graph once a new image has to be rendered. With a push paradigm, new data can be rejected at the sink, if no new image is required to be rendered, for example, if data updates are too frequent. That would require the caching of intermediary data possibly congesting memory over time.

Each functional entity – the nodes in the Module graph – is organized as a *Module*. A Module represents a component in the functional logic of the visualization or data-processing pipeline depicted by the Module graph. The granularity of a Module is not enforced by MegaMol and is left open to the developer. A Module can just represent a single algorithm or an entire rendering pipeline. For instance, the entire functionality of a deferred shading renderer can be encapsulated in a single Module with the final image as output, or each rendering stage can be separated into its own Module transferring intermediary buffers between the respective Modules. Note that a fine-granular separation between Modules is best suited for the re-usability paradigm of MegaMol. In the previous example, for instance, the final lighting stage could be shared between different deferred shading renderers. Each data entity produced by a Module is owned by this Module. This is an important invariant of the zero-copy paradigm in MegaMol, which aims at scalability and at reducing memory footprint.

Modules in MegaMol are interconnected via *Calls* that model intents. An intent specifies the reason for invoking a call, for instance, requesting new data, or asking for data extents. Again the granularity of Calls is a choice of the developer. However, here it is recommendable to bundle all intents that a Module can provide in a single Call. This way a Call models the complete interface to a specific data type, for instance, particle data, or volume data. Such a Call is reusable if the underlying data model is reusable, i.e. all Modules processing particle data should implement the intents of the generic particle data Call. In general, Calls transport only references to data owned by a Module. This is another invariant of the zero-copy paradigm. The exception is integral data types, which usually represent meta-information, that are directly stored in the Calls.

Each Module exposes *Parameters* that provide the means for a user to interact with the functionality encapsulated in a Module. Every Parameter represents a value, e.g., a scalar, a color, or a file path. It also stores meta-information such as value bounds, or can be associated with a callback that is triggered once the corresponding value has changed. The Parameters are accessible through the API of MegaMol such that a user can interact either through command line options, scripting, or through a GUI.

Every entity of data that traverses the Module graph, including intermediary results of processing algorithms, is attributed with a hash value. With that attribute, a processing or rendering method can decide whether it already has calculated or rendered results based on this specific data entity or not. For dynamic data sets, the combination of frame ID and data hash identifies the data and can be used to reduce unnecessary re-computations.

The native GUI of MegaMol exposes all Parameters of all Modules within a Module graph. It is based on the AntTweakBar library and is currently in the transition towards the more common ImGui [15]. The GUI is a minimalistic, straightforward front end that follows MegaMol's characteristic as prototyping framework.

MegaMol is written in C++ and provides a Lua-based scripting interface. The rendering is primarily done on the GPU via OpenGL. Additionally, MegaMol provides experimental DirectX rendering modules, and currently, the focus is shifted more towards CPU-based rendering.

4 Focus features

In this section, we want to highlight several existing as well as recent additions and extensions of the MegaMol visualization framework.

4.1 Particle-based visualization

The original design of MegaMol was oriented towards the rendering, visualization, and analysis of particle-based data. Therefore, most Modules available in MegaMol still handle that data type. There are a set of loader Modules that handle data input. MegaMol is able to read data produced by a variety of simulation software, e.g., ls1 [16], ESPResSo [17], and IMD [18]. The loaded data can be processed further within MegaMol prior to rendering. It provides Modules for filtering or clustering data. Missing attributes, for instance, the velocity or local temperature (in the form of kinetic energy), can be reconstructed (the quality depending on the granularity of time steps written by the simulation). Additionally, pre-processing Modules required for some rendering methods are included, as well, e.g., neighborhood search, density, or some order. Data sets from different sources can be concatenated. They are automatically aligned if they have similar spatial domains. Some data type conversions are also possible, such as particles to volume, or particles to trajectories.

The primary rendering primitive for particles is a van-der-Waals sphere. MegaMol provides several rendering Modules for that representation, each with a different purpose, e.g. approximating global illumination effects. Additionally, there exist Modules for particle trajectories rendering them as arrow glyphs or pathlines, and volume rendering Modules for a density-based representation. The most generic renderer for depicting spheres is the *SimpleSphereRenderer*, which is also best suited for a first look at unknown data. If the data set is too large in terms of particle count, we can leverage occlusion effects with respect to a specific camera view by utilizing occlusion queries, or early-z tests as shown by Grottel et al. [19]. If a better depth-perception is required to analyze the structure of a particle-based data set, we use object-space ambient occlusion, which is a comparatively fast approximation of global lighting as presented by Grottel et al. [20]. Shadows can further improve the perception of structure and shape in combination with ambient occlusion. Thus we developed the method of *Implicit Sphere Shadow Maps*, described by Krone et al. [21], which is an extension of the method presented by Story [22]. Additionally, MegaMol is able to combine transparency and ambient occlusion for particle-based data sets through a method by Staib et al. [23].

MegaMol's architecture allows using every render pipeline in a data-parallel distributed setup, leveraging the MPI integration in MegaMol. This has been mostly used to provide images for tiled and large displays, see Section 4.8 for details. Recent advances, such as MPI-based data exchange Modules, push MegaMol to support data-distributed rendering, as well, which is especially useful for in situ rendering setups.

For more details on MegaMol's capabilities with respect to particles, we refer the reader to [1].

4.2 Lua scripting

We decided to use *Lua* [24] as scripting interface of MegaMol because it is lightweight and can be effectively sandboxed. We use this scripting language to interface directly with the MegaMol core functionality. All core Parameters, as well as all Parameters of Modules in an active Module graph, are exposed through the Lua interface. The Module graph and all its elements can be retrieved and manipulated at runtime. The scripting interface can also be accessed remotely via a command line client or the graphical Configurator utility. Since changes to entities in the graph, while it is being traversed, results in undefined behavior, graph updates are queued for execution in the main thread. Configuration files and project files to initialize a Module graph upon start-up of MegaMol are written in Lua using the same API, in order to keep all MegaMol functionality encapsulated within the same script engine. This is a major change with respect to the original XML-based interface. The last XML-based functionality is technique files for shaders, which will also be replaced by Lua in the near future.

4.3 Sofware-defined visualization

Running MegaMol on HPC systems in order to enable in situ visualization, render large amounts of data, and run remote visualizations from a cluster to a large tiled display was not possible on clusters without a GPU. Since the majority of HPC systems are affected by this problem, we enabled software-defined visualization (SDVis) for MegaMol [25]. As an SDVis framework, we chose to integrate OSPRay [4] – an interactive ray tracing engine – into MegaMol. OSPRay offers an efficient renderer for scientific visualization applications, distributed rendering, advanced rendering effects (e.g. ambient occlusion), and it is extensible. With the OSPRay ray tracing engine, a new Module graph paradigm was introduced to MegaMol (cf. Sect. 3). The *daisy-chaining* paradigm allows stacking of an arbitrary amount of geometries and lights for lazy evaluation (see Fig. 1).

We utilize the capability of software rendering in MegaMol to run in situ visualizations of muscle fiber simulations [26] and thermodynamic simulations [16]. Additionally, the rendering of high-quality images for publications [25,26] and exhibitions [27] using OSPRay's SciVis renderer or path tracer is a useful new feature of MegaMol. Figure 2 shows a high-quality rendering (16k resolution) that was produced for the SFB 716 exhibition [27].

4.4 Video creation/movie making

Usually, researchers want to communicate gained insight through interesting visualizations. MegaMol provides support for this through an integrated movie maker tool, the *Cinematic Camera*. It is fully integrated into MegaMol and can be applied to any Module graph that renders images. Within the Cinematic Camera, a user can define an arbitrary tracking shot based on keyframes. The Cinematic Camera distinguishes two types of time. There is the animation time, which relates to the time of the motion path. Additionally, there is the simulation time which describes different time steps of the data set. By distinguishing these two types of time, effects like slow motion or freeze frames can directly be rendered by MegaMol without relying on external movie editing tools. The rendering of high-quality movies can be very time-consuming, especially when using advanced shading techniques, high output resolutions, or the OSPRay path tracer. The Cinematic Camera circumvents this

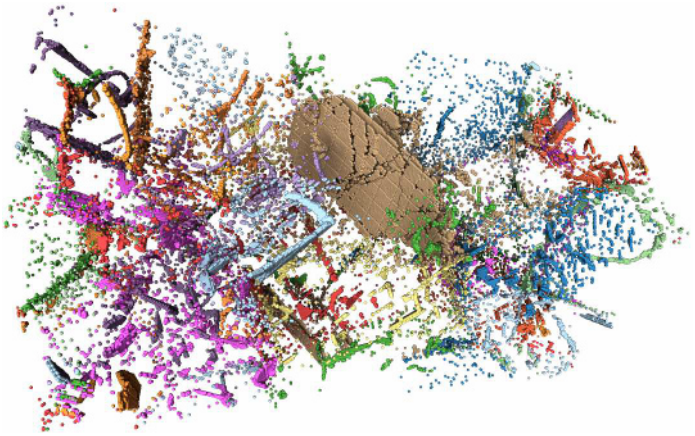

Fig. 2. Visualization of crystal structure defects of an aluminum-nano-polycrystal during a pulling test simulation. Only the atoms that are not in the face-centered cubic structure are clustered and colored. The ambient occlusion of the OSPRay SciVis renderer improves the impression of depth, for example, the brown cluster consists of two thin plates that can only be distinguished because of the illumination.

limitation with a significant reduction of the overall rendering time by utilizing the support of synchronized execution on a GPU cluster and applying distributed rendering on image-space subdivided tiles. The Cinematic Camera has been successfully used to produce the movie material for the full dome movie used in the SFB 716 exhibition [27].

4.5 Information visualization

While MegaMol was initially built with scientific visualization in mind, several information visualization techniques have turned out to be useful in the presence of high dimensional point data. Unlike other tools for abstract visualization of point data such as *Excel* or *R*, MegaMol is capable of rendering large quantities of high dimensional points at interactive frame rates. We achieve such a good performance through the adaption of particle rendering techniques, e.g., point-sprite-based splatting, and SSBO-based streaming. Figure 3 shows the result from rendering modules for parallel coordinates [28,29] (PCP) and scatter plot matrices [30] (SPLOM). Both of these techniques correlate two dimensions while maintaining a detailed impression of data distribution. Moreover, the information visualization Module supports several dimension reduction techniques such as *principal component analysis* (PCA), *metric multi-dimensional scaling* (MDS) and *t-Distributed Stochastic Neighbor Embedding* (t-SNE). Technically speaking, these techniques project a high dimensional space down onto a two or three-dimensional space. This approach is useful if users want to analyze high dimensional distances without existing dimensions providing a good low dimensional insight.

4.6 Biomolecular visualization

MegaMol offers many possibilities to process and visualize biomolecules, especially proteins. It is able to load files obtained from the Protein Data Bank [32] as well as

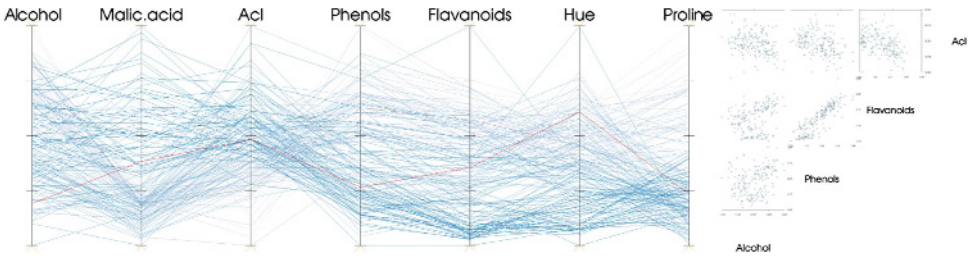

Fig. 3. Excerpt from wine data set. The parallel coordinates (left) shows correlation using parallel lines between dimensions and anti-correlation using crossed lines. The scatter plot matrix (right) depicts correlation and anti-correlation using an angle of a thought linear regression line. The visual impression of density stems from splatting, i.e. kernel density estimation. The red line (left) denotes a selected high-dimensional point.

from simulation tools like GROMACS [33]. This covers single files and even whole trajectories.

Proteins can be shown in many of the classical representations, like the *licorice* model or the *ball-and-stick* representation. Especially for molecular surfaces, like the Solvent Excluded Surface [34,35] or Gaussian density surfaces [36], many state-of-the-art implementations are available. The used algorithms are accelerated using OpenGL shaders or the CUDA framework to achieve interactive frame rates even for large molecules. More abstract representations of proteins, like *Cartoon* renderings [37] are also supported using various techniques, like tessellation, for example. It is also possible to enrich this representation by showing an additional value, e.g. characterizing uncertainty [38]. All of the mentioned representations can be arbitrarily combined by the user.

The more advanced protein visualization capabilities of MegaMol include detection and tracking of molecular cavities, the tracing of solvent and ligand molecules over the course of a simulation, and various other techniques. Cavities in the molecular surface are recognized using either volume-based techniques [39] or ambient occlusion, which is typically used for illumination [40]. A special method worth mentioning is the Molecular Surface Maps method [31]. It creates a two-dimensional map by projecting a protein onto a circumscribed sphere. An example of a Molecular Surface Map is given in Figure 4. To make this possible, cavities in the molecule have to be detected and closed beforehand. Opposed to the originally published paper, our current implementation now uses Voronoi diagrams, utilizing the method of Lindow et al. [41], to perform this operation.

4.7 Mesh rendering

Modern mesh rendering in MegaMol is designed around a strong decoupling of render logic and the actual mesh data. It offers very high flexibility with almost arbitrary vertex attribute layouts and shader inputs. Unlike most render modules in MegaMol, the mesh renderer does not use a specific shader program, but rather requires the data source Module to supply a matching shader program to use with the given mesh data.

The mesh data, all necessary format descriptors as well as any additional input data for the shader are prepared and batched by the data source Module and then simply passed on to the renderer, which directly uploads the data to various large OpenGL buffer objects. Therefore, the renderer itself requires very little knowledge about the given mesh data and the core render loop is reduced to a minimal amount

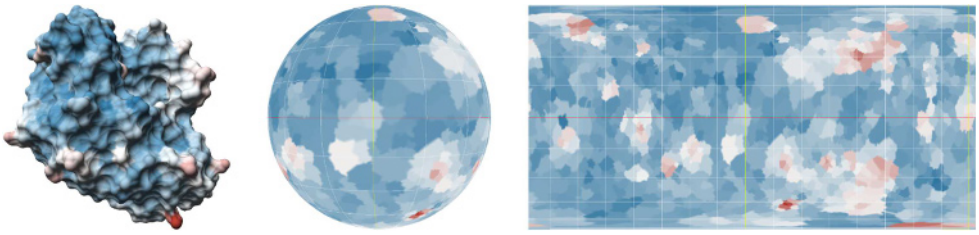

Fig. 4. Overview of the pipeline of the Molecular Surface Maps [31]. The Solvent Excluded Surface (left) is projected onto a sphere (middle). Using general map projection techniques, this is then projected onto a map (right). Colors encode the B-factor of the atoms, where a blue color corresponds to a low value and red to a high one.

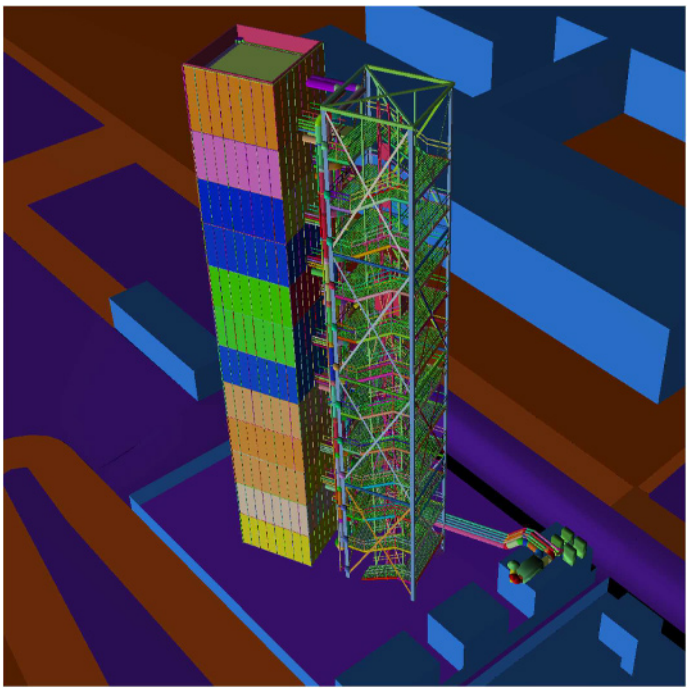

Fig. 5. Mesh data set that consists of 3624 individual objects (shown by randomized colors for each object) and a total of 5.5 million triangles. Using a single draw call, it is rendered in less than 4ms.

of OpenGL API calls, centered around the *glMultiDrawElementsIndirect* draw call that is available since OpenGL 4.3. Costly OpenGL state changes are thus limited to switching the shader program or vertex layout, making it possible to process complex scenes with thousands of unique and individual objects with a single draw call, such as in Figure 5. This modern, optimized approach follows the *Approaching Zero Driver Overhead* paradigm that is also frequently adopted by modern high-performance game engines [42,43].

So far, support for STL and glTF [44] mesh formats have been implemented, but more formats can be easily added as additional data source Modules. Due to the data-driven and minimalistic approach of the core rendering routine, a high geometry throughput and overall high rendering performance are achievable. On a consumer

grade GPU (Nvidia GTX 1070), we measured a throughput of 1.46×10^9 triangles per second.

4.8 Powerwall support

MegaMol also supports large high-resolution displays and stereo rendering on such displays. This is realized by means of a synchronized execution approach on a GPU cluster and built-in image-space subdivision based on a display topology provided by the user. Technically, MegaMol uses MPI for synchronization and communication, leveraging the high-bandwidth and low-latency networks usually available on such clusters. At startup, all instances perform a node coloring step, which allows MegaMol to work with other applications also using MPI as their means of communication. This paves the way for integrating MegaMol as in situ visualization into simulation code.

The instances in the cluster are controlled by a single master instance, called the operator node. This operator node distributes the project files to the rendering nodes and also serializes and transmits all parameter changes, including the camera state, to keep the slaves in sync. The mechanism can also be used to attach other applications controlling the MegaMol cluster, for instance, mobile devices or spatial input devices.

4.9 Web and remote visualization

While MegaMol is typically used as a standalone application that directly renders the data, it can also be used for remote visualization on the web. To this end, we implemented a Module that establishes a WebSocket server. If a client issues an HTTP request via the browser, a two-way connection is established. We defined a protocol that allows streaming data from MegaMol to the JavaScript client application running in the browser. This protocol supports particle data as well as triangle meshes [45]. We implemented a prototypical client application that renders molecular data in the browser using WebGL. Particle data gets rendered using GPU-based ray casting, similar to MegaMol [46]. Meshes can be used to render complex molecular representations in the browser by computing them in MegaMol first and sending just the triangle data to the client. This, for example, allows the visualization of molecular surfaces on devices that lack the hardware capabilities to compute them. The data transfer protocol was subsequently extended by using quantization, which reduces the amount of data that has to be transferred from the server to the client [47]. Rendering performance is not on par with native executables. For data sets of around one million atoms, with the optimized transfer scheme, rendering performance never drops below 20 frames per second on the tested hardware (Intel Core i7-2600, Nvidia GeForce GTX 660 Ti PC and Intel Core i7-3520M, Nvidia GeForce GT 640M LE laptop).

5 Conclusion and outlook

The development of MegaMol started with a focus on particle-based visualization on workstations. Thanks to the flexible architecture for rapid prototyping, a lot of different Modules for particle-based data set were soon available within the MegaMol environment. Over time the functionality grew to support general information visualization, biomolecules, and other data types. Starting from the original paradigm

of renderers pulling data from sources and contributing to a common frame buffer, we extended upon this via daisy-chaining, as required by the singleton OSPRay renderer, and via the shifted responsibilities represented by the new mesh renderer: Here data sources have to contribute more than metadata about buffer contents, for example shader code for accessing the data to make sure the renderer is lightweight and universally applicable.

We already experimented with data distribution and in situ visualization, which is in our opinion important to tackle future challenges with respect to supercomputing and large simulation runs. During these experiments, we learned that the current architecture of MegaMol limits our possibilities to extend it in these directions: The Module graph is designed to be traversed on a single thread. Only the functionality within Modules can be executed in parallel. We are currently re-designing the core architecture of MegaMol towards a model that can also bridge between single-threaded and multi-threaded parts of the code.

First and foremost, we want to thank Sebastian Grottel, who started the MegaMol project in 2006 and was the main developer until 2012. We also want to thank the German Science Foundation (DFG) for funding MegaMol development as part of CRC (SFB) 716 project D.3 and via the sustainability call for research software in project ER 272/12-1 "Research Software Sustainability for the Open-Source Particle Visualization Framework MegaMol". We also want to thank Joachim Staib, Alexander Straub, Oliver Fernandes, and all students for their contributions to MegaMol.

References

1. S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl, IEEE Trans. Visual Comput. Graphics **21**, 201 (2015)
2. S. Grottel, G. Reina, M. Krone, C. Müller, T. Ertl, in *Workshop on visualization in practice* (2016)
3. M. Krone, S. Grottel, G. Reina, C. Müller, T. Ertl, IEEE Comput. Graphics Appl. **38**, 109 (2018)
4. I. Wald, G.P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, P. Navrátil, IEEE Trans. Visual Comput. Graphics **23**, 931 (2017)
5. I. Wald, A. Knoll, G.P. Johnson, W. Usher, V. Pascucci, M.E. Papka, in *2015 IEEE scientific visualization conference* (2015), pp. 57–64
6. Intel SPMD Program Compiler, https://ispc.github.io/, Accessed: 2018-09-27
7. W.J. Schroeder, K.M. Martin, in *Visualization handbook*, edited by C.D. Hansen, C.R. Johnson (Butterworth, Heinemann, 2005), Chap. 30, pp. 593–614
8. J. Ahrens, B. Geveci, C. Law, in *Visualization handbook*, edited by C.D. Hansen, C.R. Johnson (Butterworth, Heinemann, 2005) Chap. 36, pp. 717–731
9. H. Childs, E. Brugger, B.J. Whitlock, J.S. Meredith, S. Ahern, K. Bonnell, M. Miller, G.H. Weber, C. Harrison, D. Pugmire, T. Fogal, C. Garth, A. Sanderson, E.W. Bethel, M. Durant, D. Camp, J.M. Favre, O. Rubel, P. Navratil, M. Wheeler, P. Selby, "VisIt: an end-user tool for visualization and analyzing very large data", 1st edn., in *High performance visualization: enabling extreme-scale scientific insight*, edited by E.W. Bethel, H. Childs, C. Hansen, (CRC Computational Science Series, Taylor and Francis, Boca Raton, 2012), Vol. 1, p. 520
10. A. Stukowski, Model. Simul. Mater. Sci. Eng. **18**, 015012 (2010)
11. W. Humphrey, A. Dalke, K. Schulten, J. Mol. Graph. **14**, 33 (1996)
12. E.F. Pettersen, T.D. Goddard, C.C. Huang, G.S. Couch, D.M. Greenblatt, E.C. Meng, T.E. Ferrin, J. Comput. Chem. **25**, 1605 (2004)
13. W.L. DeLano, CCP4 Newsletter on Protein Crystallography, No 40 (2002)

14. J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Schulten, J. Comput. Chem. **26**, 1781 (2005)
15. O. Cornut, *ImGui Project Page*, https://github.com/ocornut/imgui, Accessed: 2018-10-02
16. C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C.W. Glass, H. Hasse, et al., J. Chem. Theory Comput. **10**, 4455 (2014)
17. H.J. Limbach, A. Arnold, B.A. Mann, C. Holm, Comput. Phys. Commun. **174**, 704 (2006)
18. J. Stadler, R. Mikulla, H.-R. Trebin, Int. J. Mod. Phys. C **08**, 1131 (1997)
19. S. Grottel, G. Reina, C. Dachsbacher, T. Ertl, Comput. Graphics Forum **29**, 953 (2010)
20. S. Grottel, M. Krone, K. Scharnowski, T. Ertl, in *IEEE pacific visualization symposium* (2012), pp. 209–216
21. M. Krone, G. Reina, S. Zahn, T. Tremel, C. Bahnmüller, T. Ertl, in *IEEE pacific visualization symposium* (2017), pp. 275–279
22. Hybrid Ray Traced Shadows, https://developer.nvidia.com/content/hybrid-ray-traced-shadows, Accessed: 2018-10-02
23. J. Staib, S. Grottel, S. Gumhold, Comput. Graphics Forum **34**, 151 (2015)
24. The Programming Language Lua, https://www.lua.org/home.html, Accessed: 2018-10-02
25. T. Rau, M. Krone, G. Reina, T. Ertl, in *7th workshop on visual analytics, information visualization and scientific visualization*, http://sibgrapi2017.ic.uff.br/e-proceedings/assets/papers/WVIS/WVIS2.pdf (2017)
26. C.P. Bradley, N. Emamy, T. Ertl, D. Göddeke, A. Hessenthaler, T. Klotz, A. Krämer, M. Krone, B. Maier, M. Mehl, T. Rau, O. Röhrle, English Front. Physiol. **9**, 816 (2018)
27. M. Krone, K. Schatz, N. Hieronymus, C. Müller, M. Becher, T. Barthelmes, A. Cooper, S. Currle, P. Gralka, M. Hlawatsch, L. Pietrzyk, T. Rau, G. Reina, R. Trefft, T. Ertl, in *Proceedings of SIGRAD 2017* (2017), pp. 17–24
28. J. Heinrich, D. Weiskopf, in *STAR proceedings of eurographics 2013* (2013), pp. 95–116
29. A. Inselberg, *Parallel coordinates: visual multidimensional geometry and its applications* (Springer-Verlag, New York, 2009)
30. J.A. Hartigan, J. Stat. Comput. Simul. **4**, 187 (1975)
31. M. Krone, F. Friess, K. Scharnowski, G. Reina, S. Fademrecht, T. Kulschewski, J. Pleiss, T. Ertl, IEEE Trans. Visual Comput. Graphics **23**, 701 (2017)
32. H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, P.E. Bourne, Nucl. Acids Res. **28**, 235 (2000)
33. M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess, E. Lindahl, SoftwareX **1–2**, 19 (2015)
34. F.M. Richards, Annu. Rev. Biophys. Bio. **6**, 151 (1977)
35. M. Krone, S. Grottel, T. Ertl, in *IEEE symposium on biological data visualization* (2011), pp. 17–22
36. M. Krone, J.E. Stone, T. Ertl, K. Schulten, in EuroVis – Short Papers (2012), pp. 67–71
37. J.S. Richardson, Adv. Protein Chem. **34**, 167 (1981)
38. C. Schulz, K. Schatz, M. Krone, M. Braun, T. Ertl, D. Weiskopf, in *IEEE pacific visualization symposium* (2018), pp. 96–105
39. M. Krone, D. Kauker, G. Reina, T. Ertl, in *2014 IEEE pacific visualization symposium* (2014), pp. 301–305
40. M. Krone, G. Reina, C. Schulz, T. Kulschewski, J. Pleiss, T. Ertl, Comput. Graphics Forum **32**, 331 (2013)
41. N. Lindow, D. Baum, H.-C. Hege, IEEE Trans. Visual Comput. Graphics **17**, 2025 (2011)
42. C. Everitt, OpenGL Efficiency: AZDO, https://www.khronos.org/assets/uploads/developers/library/2014-gdc/Khronos-OpenGL-Efficiency-GDC-Mar14.pdf, Accessed: 2018-10-01

43. G. Wihlidal, Optimizing the Graphics Pipeline with Compute, https://www.ea.com/frostbite/news/optimizing-the-graphics-pipeline-with-compute, Accessed: 2018-10-01
44. glTF – Runtime 3D Asset Delivery, https://github.com/KhronosGroup/glTF, Accessed: 2018-10-02
45. F. Mwalongo, M. Krone, M. Becher, G. Reina, T. Ertl, in *Proceedings of the 20th International Conference on 3D Web Technology* (2015), pp. 115–122
46. F. Mwalongo, M. Krone, G. Karch, M. Becher, G. Reina, T. Ertl, in *Proceedings of the 19th International ACM Conference on 3D Web Technologies* (2014), pp. 133–141
47. F. Mwalongo, M. Krone, M. Becher, G. Reina, T. Ertl, Graphical Models **88**, 57 (2016)