The Gap between Visualization Research and Visualization Software (VisGap) (2026)
K. Blenman, C. Gillmann, M. Krone, G. Reina, T. Wischgoll (Guest Editors)

Jinsoku: A Data-Centric Visualization Engine

M. B. Jensen[1], J. R. Frisvad[2], A. Pranovich[2], S. Murthy[3], M. Kühl[3], and J. A. Bærentzen[2]

[1]Department of Engineering Technology and Didactics, Technical University of Denmark, Ballerup, Denmark
[2]Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kongens Lyngby, Denmark
[3]Marine Biology Section, Department of Biology, University of Copenhagen, Helsingør, Denmark

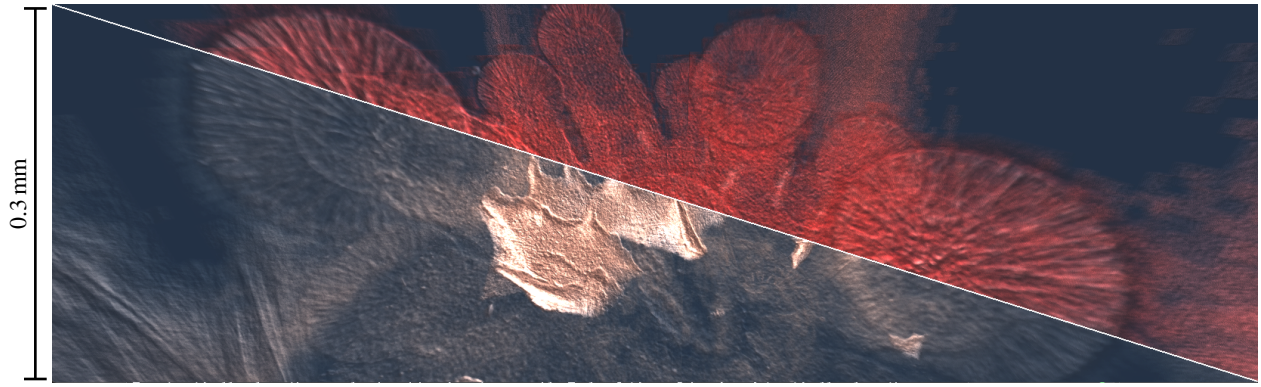

Figure 1: *A visualization of a volumetric scan of a coral polyp. The left side applies a high-density contrast filter to isolate the underlying skeletal structure, while the right side utilizes a targeted density window to expose the delicate, lower-density soft tissue of the coral. Interestingly, anisotropic stinging cells (nematocysts) appear at the tips of the polyp's tentacles. The raw memory footprint of the volume is 37 GB.*

Abstract

The consolidation of real-time visualization around General Purpose Game Engines (GPGEs) has democratized access to immersive technologies. However, as the scale and resolution of modern scientific datasets increase, the rigid abstraction layers of commercial engines introduce a layer of friction. Researchers are frequently forced to aggressively decimate data or compromise on interactive frame rates to satisfy the constraints imposed by GPGEs. To bridge this gap, we present a bespoke, data-centric visualization engine designed to minimize the abstraction distance between scientific data and GPU hardware. Jinsoku prioritizes a data-aware ingestion pipeline and explicit resource management, allowing developers to construct highly specialized visualization pipelines. We validate our architecture through two distinct case studies. First, we demonstrate that by bypassing GPGE overhead and leveraging hardware-level features like multi-view rendering and mesh shading, Jinsoku accelerates the virtual reality (VR) inspection of a 38-million-triangle aerodynamic model by up to 75% compared to commercial alternatives. Secondly, we present an out-of-core volumetric rendering pipeline for massive datasets from synchrotron and CT-scanning, utilizing a bespoke sparse-bricking memory model that facilitates the loading of arbitrarily large volumes that we were unable to load with standard tools. We argue that for high-fidelity visual analysis, returning to domain-specific software architectures is an essential prerequisite for maintaining scientific integrity. Jinsoku is available on GitHub at `https://github.com/senbyo/jinsoku`.

CCS Concepts

• ***Human-centered computing*** → ***Visualization toolkits; Scientific visualization;*** • ***Computing methodologies*** → ***Rendering;***

1. Introduction

The acquisition of scientific data is continuously increasing in resolution and detail [Mar13], characterized not only by increasing size but by significant heterogeneity [KH13]. Modern scanning and simulation modalities — ranging from computed tomography (CT) to topology optimization — generate datasets with increasingly finer resolution and larger vertex count. Crucially, this

DOI: 10.2312/visgap.20261001

diversity extends to the data formats themselves; each modality necessitates bespoke serialization and parsing routines that often rival the visualization pipeline in implementation difficulty. This drives requirements that are increasingly hard to accommodate by general-purpose software, which typically does not support the ad-hoc, proprietary, or optimized binary formats of domain-specific research [HBB*15]. In this landscape, the central challenge for visualization research is no longer merely visualizing these datasets, but architecting software that is capable of parsing and visualizing datasets without compromising their scientific fidelity through aggressive decimation or simplification.

While commodity hardware has advanced significantly, providing hardware-accelerated ray tracing and massive parallel throughput, the software layer has increasingly consolidated around General Purpose Game Engines (GPGEs) such as Unity[1] or Unreal Engine[2] [KGKG24]. These engines offer robust feature sets for entertainment, but they impose heavy abstraction layers designed for generic use cases [Ber25]. For high-density scientific visualization, these abstractions make it increasingly difficult to work with data that deviates from supported formats. This friction is evident both in the generic memory management strategies, which struggle to maintain the stringent frame rates necessary to avoid motion sickness in immersive environments [MAJ*22], and in the systemic inability to efficiently stream datasets that are larger than the available physical memory [ZJAW25].

We argue that to close the gap between data acquisition and interactive visualization, we must move beyond the "black box" limitations of commercial engines and return to bespoke, domain-specific software architectures. To this end, we present Jinsoku, a visualization engine designed explicitly to expose the domain-specific parts of the application for modification. This enables us to effectively modify the engine for different data formats and use-cases. Jinsoku is designed as a thin layer on top of platform specific APIs to ensure the ease of use, and to make sure that performance is not negatively affected by gradual expansion of the engine layer over time.

We validate our approach through two case studies, where Jinsoku is used to (1) visualize large geometric data in virtual reality (VR) and (2) visualize large out-of-core volume data. We demonstrate this by stripping away carefully selected layers of modern game engines. We expose the data-centric logic to the user, allowing for direct alignment between the data and the rendering pipeline. In the first case study (1), we achieve interactive performance on datasets that are intractable in standard commercial environments, building upon prior evidence that bespoke visualization engines can outperform GPGEs in raw polygon throughput [JJFB21]. In the second case study (2), we interactively visualize massive volumetric data, which is simply not possible when using standard GPGE or visualization tools, as they are not built for out-of-core data streaming. Ultimately, we show that correctly designed bespoke engines are a necessary prerequisite for closing the gap between specialized scientific research and general-purpose visualization software.

[1] https://unity.com/
[2] https://www.unrealengine.com/

2. Related Works

The dichotomy between general-purpose software and bespoke scientific tools is a recurring theme in visualization research. While commercial game engines have democratized access to high-fidelity rendering, recent work highlights the friction between their entertainment-focused architectures and the rigorous data requirements of scientific inquiry.

2.1. The Hegemony of Game Engines

Early games were largely written from scratch with very little content and few roles in the development process other than that of the programmer. However, since games generally require many similar facilities, software layers emerged between the bespoke code for the individual game and the APIs for graphics, sound, etc. These layers are generally called game engines [MS16]. Platforms such as Unity and Unreal Engine offer robust ecosystems for VR development, handling input standards, and asset pipelines with ease. However, their dominance comes at the cost of architectural opacity. As noted by Krüger et al. [KGKG24], the "black box" nature of these engines often forces researchers to adapt their data to the engine's limitations rather than the inverse, introducing significant overhead for high-order finite element data or non-standard topologies [CGS*13]. While plugins for tools like ParaView [JAG19] attempt to bridge this gap, they often inherit the performance penalties of the underlying abstraction layers [JJFB21], struggling with the "Motion-to-Photon" latency constraints critical for comfortable VR leading to the need for simplification of the data [MAJ*22].

2.2. Bespoke Scientific Visualization Engines

Despite the prevalence of General-Purpose Game Engines (GPGEs), domain-specific architectures remain superior for specialized tasks. While consumer-grade hardware has democratized scientific visualization, off-the-shelf software frequently struggles with the scale, dimensionality, and specialized low-level rendering requirements — such as building unique data pipelines — needed for complex scientific datasets. This limitation has driven researchers across various fields to abandon generalized tools in favor of developing bespoke visualization engines tailored to their specific data and hardware needs.

For abstract and multi-dimensional data, custom encoding strategies are required to move beyond standard XYZ-axes. Tools like *iViz* [DDC*14] and *SphereViz* [SDC07] were explicitly built to map high-dimensional image and dataset groupings into immersive 3D spaces, while other custom applications overlay complex social data, such as geo-tagged posts, onto physical 3D topologies [MGHK15].

In the medical, biomedical, and neuroscience domains, standard rasterization techniques fail to handle the nested, semi-transparent surfaces of cross-sectional imaging. Consequently, bespoke volume-rendering platforms like *DIVA* [EGC*20] and *ConfocalVR* [SLS18] have been created to provide real-time clipping and exploration of cellular architectures. The functional superiority of these custom tools is highly measurable: Usher et al. [UKF*18]

demonstrated with *NeuroTrace* that bespoke environments significantly reduce fatigue and error rates in neuron tracing compared to desktop or general-purpose solutions.

Similarly, engineering and materials science rely on custom engine solutions to navigate vast databases and inspect physical simulations. *NOMAD VR* [GK19] exemplifies how custom rendering tools allow for the visualization of massive material science datasets across heterogeneous hardware, from standard displays to Cave Automatic Virtual Environments (CAVE) setups to mobile Head-Mounted Displays (HMDs). This tailored architecture promotes an inside-out exploration style that accelerates the detection of chemical and structural anomalies. In applied engineering, bespoke environments allow for scale-accurate prototype inspections, such as realistic oceanic and acoustic simulations for yacht design [MEC14], which can entirely replace expensive physical prototyping [Wol19].

The complexity of scientific data often necessitates collaborative analysis, a feature that requires highly specialized networking architectures to synchronize massive datasets across custom engines. Bespoke tools support multi-user environments either locally or remotely. In a local context, CAVE systems allow multiple researchers to physically occupy the same space; developers of *NOMAD VR* noted that this setup is particularly successful for experienced users guiding novices toward regions of interest. For remote collaboration, tools like *ConfocalVR* and *TeraVR* [WLL*19] enable multiple users to co-annotate data over the internet. In one study, *TeraVR* allowed three researchers across different cities and countries to co-annotate in real-time, reducing annotation time to 20% of a single user's time without sacrificing quality.

In the cultural heritage domain, bespoke pipelines have enabled the rendering of massive point clouds and photogrammetry datasets that would overwhelm standard game engine importers [SW15]. Bespoke engines have allowed researchers to digitize fragile artifacts [GCM*15], build interactive exploration pipelines with contextual hotspots [JMR17], and reconstruct ancient historical sites with thousands of simulated agents [KJK*15].

Collectively, the above-mentioned work argues that for data at the extremes of scales, complexity, or collaborative need, the "do-it-yourself" approach yields functional benefits that far outweigh the initial development costs. However, there seems to be scattering, where each domain and each dataset ends up with its own visualization engine, which then needs to be supported and maintained. We attempt to consolidate this need into one visualization engine, effectively reducing the maintenance cost, while still providing the flexibility to adapt the visualization engine to specific domains.

2.3. Advances in High-Performance Rendering

Our work builds upon recent hardware-level advances that are often slow to propagate to commercial engines. The introduction of the Mesh Shading pipeline [Kub18a] represents a paradigm shift from vertex-bound processing to compute-centric geometry synthesis, allowing for per-cluster culling that is essential for massive Computer-Aided Design (CAD) and biological datasets [Kub18b]. Parallel to this, advances in sparse volumetric data structures—specifically Sparse Voxel Octrees (SVOs) and brick-based equivalents—have enabled out-of-core rendering of terabyte-scale volumes [LK10].

The visualization of high-resolution volumetric data is a foundational challenge in computer graphics, traditionally addressed through ray marching and texture-based volume rendering. While standard approaches work well for bounded domains, modern scientific datasets acquired through scanning, often exceed the memory capacity of commodity GPUs. To address this, out-of-core techniques like Sparse Voxel Octrees (SVOs) [LK10] and hierarchical data structures like OpenVDB [Mus13] have become industry standards for offline rendering. However, integrating these structures into real-time, and VR-capable pipelines remains a challenge. Recent work by Zellmann et al. [ZJAW25] demonstrates that leveraging low-level compute APIs to implement hierarchical compression (e.g., NanoVDB) can enable interactive path tracing of massive volumes. Our engine adopts a similar philosophy but specializes the memory layout further: rather than a generic tree structure, we utilize a domain-specific "brick-based" sparse binding that aligns directly with the distinct morphological features of the data, minimizing the traversal overhead typically incurred by generic VDB structures. By integrating these techniques directly into a bespoke Vulkan backend, Jinsoku bypasses the generic abstraction layers of GPGEs, affording the user control over memory residency, and the topological layout of the data.

3. Jinsoku: Data-Centric Engine

Jinsoku— named after the Japanese word for swiftness and speed—is a real-time visualization engine built from scratch as an abstraction layer on top of the platform-specific APIs needed for generating and presenting images to a screen. The consideration behind abstracting away the platform specific APIs is that it enables extension by plugging in new APIs on demand. Currently, only one graphics programming API is implemented, namely Vulkan [Bai18]. Vulkan was chosen, as it affords cross-platform capabilities and because its explicit nature enables us to make more informed decisions concerning the specific application. These decisions can result in better performance, which is important when Jinsoku has to work with large and detailed scientific datasets. The engine is created as a library to be included in the target application. This enables scientists and researchers to focus on data ingestion and designing the visualization pipeline for their specific dataset, rather than having to learn all the ins and outs of modern GPU programming.

3.1. Architecture

Figure 2 is an overview of Jinsoku's architecture and its integration into an application. Jinsoku's design philosophy is to expose exactly what the application needs to visualize data in any format. The engine includes triangle mesh processing to facilitate loading of standard formats, but also exposes the loading to the application so that it can overwrite it if needed. Jinsoku runs natively in a headless mode. This enables us to toggle and choose between different presentation modes, such that we can, for instance, present to desktop, VR, or mobile as requested by the user.

To facilitate the seamless integration of data-specific visualiza-

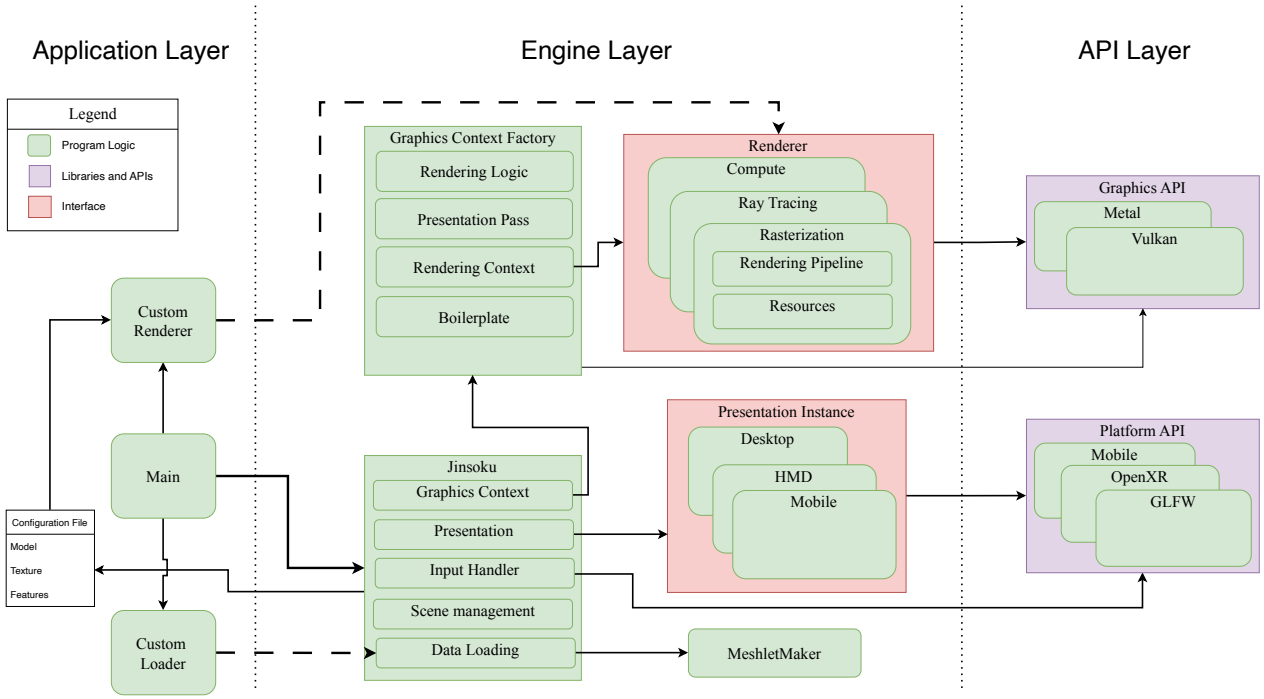

Figure 2: *Our engine's integration into a visualization application. The* Application Layer *is the specific application used for visualization. It uses a Custom Renderer and a Custom Loader (as in our Case Study 2) and makes our engine aware of the Custom Renderer via the configuration file that also provides details about the scene being visualized. In the* Engine Layer*, the green blocks show the program logic, the actual functionality of the engine. In the* API Layer*, the purple blocks contain the external dependencies that the engine layer abstracts away. The two red blocks show the parts of the engine that are built as interfaces. We have a* Graphics Context Factory *that spits directs and creates the appropriate graphics context for the platform that the engine is running on, enabling the features requested in the configuration file, and picking the most appropriate* Renderer*. The* Presentation Instance *provides an interface that defines where the rendered images are presented and depends on the platform. The* API Layer *shows the platform specific layers. Metal is shown here as an example of another graphics API, currently only Vulkan is supported.*

tion when Jinsoku is utilized as an external library; we diverge from the monolithic "Uber-Loader" pattern common in legacy game engines [Gre19]. Instead, we expose a customizable data-interception interface directly through the engine API. This architecture employs a Chain of Responsibility pattern, providing runtime callbacks that allow the Application Layer to intercept and entirely overwrite geometry loading, as well as dynamically select the optimal Renderer. This decoupling is critical for hardware-level optimization. By isolating the loading logic at the architectural level, we ensure that specialized data types can be transparently managed from disk to the GPU, entirely unconstrained by the engine's internal abstractions. As noted by Akenine-Möller et al. [AHH*18], volumetric ray-marching, for instance, requires significantly different memory access patterns compared to rasterization. Therefore, exposing the loading phase as a library hook empowers the user to dictate exact memory layouts tailored to their specific data structures.

Rather than hard-coding a specific render loop, we treat renderers as modular plugins that self-register upon application startup. Each renderer implementation (e.g., a standard Vertex Shader pipeline or a Mesh Shader pipeline) exposes a Registry Class containing its hardware requirements and a relative priority score. At runtime, Jinsoku iterates through this registry, filtering for pipelines supported by the host GPU and selecting the candidate with the highest priority. This enables us to automatically negotiate the optimal execution path—defaulting to high-throughput Mesh Shading on modern hardware, while seamlessly falling back to standard rasterization on legacy systems—without requiring manual intervention from the application developer. The render loop is contained in a *Renderer*, a class that can be overwritten by the application layer. This allows the application to write its own providing full transparency and control over how the data is visualized and uploaded to the GPU.

Once the necessary parts of the engine are extended to handle the data in question, the application can be managed from a configuration file, making the application accessible and usable for collaborating scientists who wish to use it for visualization. The configuration file is written in the Lua scripting language. The configuration file can be used to set the scene and prioritize different *Renderers*.

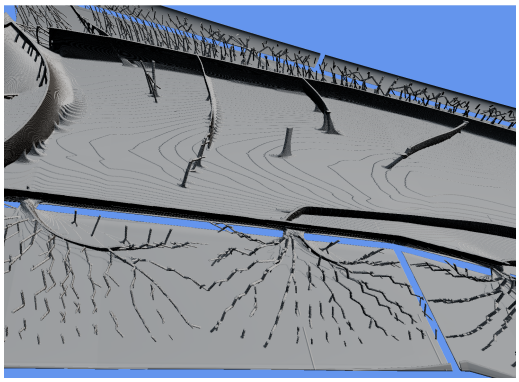

Figure 3: *The wing model seen up close (rendered with Jinsoku), showing the view that we used to record frame time.*

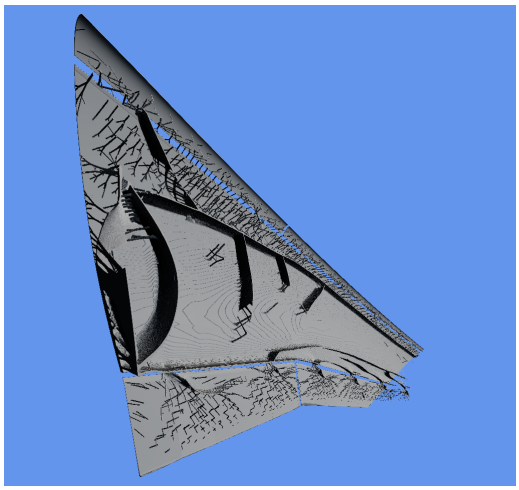

Figure 4: *The wing model seen from afar (rendered with Jinsoku), showing the view that we used to record frame time.*

4. Case Study 1: Rendering for VR

Similarly to the increase in resolution of data acquisition, simulation is also increasing in resolution. An example of this is a topology optimized airplane wing by Aage et al. [AALS17; ASLA20]. They used a topology optimization process based on the density approach, discretizing the fixed-shape wing into 1.1 billion finite elements to achieve unprecedented structural resolution. To model the physical system and redistribute material, they solved the linear Navier-Lamé equations for static elastic behavior, while employing the Solid Isotropic Material with Penalization (SIMP) model to smoothly transition between solid and void spaces. Finally, they minimized structural compliance to maximize stiffness using an optimality criterion algorithm, which required solving massive systems of linear equations via a supercomputer and a custom multilevel preconditioner. The resulting model consists of 38,629,758 triangles and 92,010,363 vertices. Renderings of the wing are shown in Figures 3-4.

For this case study, a new *Renderer* was built, which simply extends the existing rasterization in Jinsoku. A simple *forward rendering pass* was chosen. The *forward rendering path* processes each object that we wish to render, one at a time, for each light source in the scene. For most visualization tools, one light source is enough. To facilitate a more intuitive interaction the model was visualized in VR.

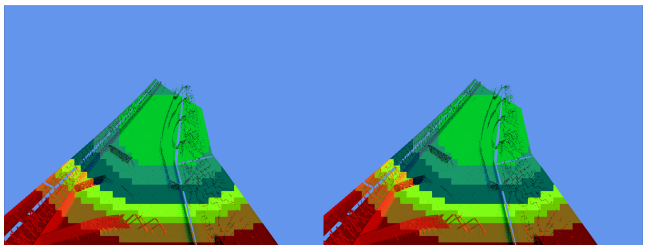

Figure 5: *A render from inside a VR headset showing both the left and right view of the wing. VRS is enabled with a circular pattern that moves outwards from the middle of the image, reducing the number of fragment calls as we move further out. This is indicated by the change in color.*

To further enhance performance, we enable two hardware extensions that increase the triangle throughput. The first is *Variable Rate Shading* (VRS), this feature essentially allows us to render images to a frame buffer that emulates a varying pixel resolution. We use a circular fall-off as we move out towards the periphery of the eye. The circle is at the center of the image, expecting users to always look straight ahead, and move the head instead of the eyes when exploring the Virtual Environment. That way we focus our rendering efforts on the fovea area and spend less computational power on the part of the image that is only visible to the peripheral vision. We can get away with this because the peripheral vision is less acute. If the eyes are moved instead of the head, then the effect breaks down. Figure 5 shows an example of the wing rendered with VRS. We have overlaid a color onto the geometry to show how we have created different zones in the image that vary the number of fragment calls per pixel.

The next extension is *Multi-View Rendering* (MVR). This allows us to change the order in which we call the mesh and fragment shaders for consecutive views. The normal way to render two views is to go through the pipeline twice, that is, calling the mesh shader and then fragment shader for each view. When MVR is enabled, the pipeline runs the mesh shader for both views at the same time, and then the fragment shader for each view. This reduces some overhead when loading shader programs, but more importantly, it allows the pipeline to reuse vertices that are already processed and in memory. This can provide a good performance improvement when a lot of the rendered geometry is visible in both views. This feature is ideal for improving performance in VR, since we always need to render two views instead of one, and the views are almost identical, meaning that nearly all the visible geometry is visible in both views. For this, we have to consider that we only call the Task Shader once at the start, so we need to expand the meshlet culling abilities to take both views into account.

An interesting little detail is that, when working with MVR, we essentially only use one image buffer, which is twice as wide, so

that it can hold two images next to each other. This is clever because by having the two images in the same memory block it becomes more efficient when writing to it at render time. However, the OpenVR API expects the user to deliver two distinct images every time, so we cannot just give it one large image instead. These images are Vulkan objects. Each *Image Object* is first created and then backed by memory via an allocation process. To get around this, we first create one *Image Object*, and back it with a memory allocation for the entire multi-view image. For the second *Image Object*, we create it but do not allocate any memory for the object. Instead, we let it point to the middle of the memory allocation of the first *Image Object*.

We compare our render times with Unity, a GPGE that is widely used both in research and game making. To do this, we recreate an experiment reported in related work [JJFB21]. We use the *Valve Index* and desktop PC with an NVIDIA RTX 2080 TI. We measure the time it takes to render a single frame in two conditions: seen from afar and up close. The measurement is in milliseconds and is shown in Figure 6. We record 2 minutes worth of frames and then average these. For a fair and simple comparison, we use Unity's Universal Render Pipeline (UnityURP) with a simple surface illumination model, namely the Phong shader [Pho75], and turn off all global effects. We include the performance obtained for this wing mesh in existing work [JJFB21] and compare with our engine in three configurations: MVR, MVR + VRS, and MVR + VRS + meshlet clustering. The latter is a triangle mesh optimization strategy [JFB23] that we implemented into Jinsoku.

UnityURP runs with an average rendering time per frame of around 10 ms. When using our MVR configuration and inspecting the wing from afar the render time decreases by roughly ∼2.0 ms compared to existing work (UnityURP and [JJFB21]). We use the meshlet clustering algorithm [JFB23] to shave another ∼0.5 ms off the average render time when inspecting the wing from afar, bringing it down to ∼7.5 ms. Existing work already improved significantly on the performance of UnityURP when rendering a large mesh up close [JJFB21]. With Jinsoku, this improved performance is further improved from ∼4.0 ms down to an average render time of ∼2.5 ms. Adding VRS, provides a small decrease in render time when inspecting the mesh from far away, but this is not enough to justify the deterioration in the image quality. We outperform existing work. When inspecting the wing from afar we render the images ∼25% faster and, when inspecting the wing up close, we render the images ∼37.5% faster.

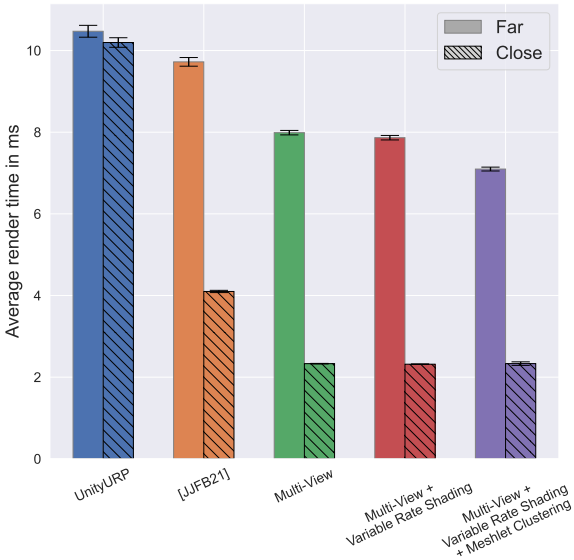

Figure 6: *The rendering performance when rendering the high poly wing model from far away and up close (Figures 3-4). The plot shows a bar for each condition grouped for each visualization tool/configuration. The rendering time is measured in milliseconds (lower is better). UnityURP refers to Unity's Universal Render Pipeline. We also compare with related work [JJFB21]. Our method is shown in three configurations: Multi-View Rendering (MVR), MVR + Variable Rate Shading (VRS), and MVR + VRS + meshlet clustering. We recommend MVR + mesh clustering, as VRS degrades the quality with only very little performance improvement.*

5. Case Study 2: Out-of-Core Volume Data Visualization

At the MAX IV synchrotron facility in Lund, Sweden, computed tomography relies on an ultra-bright, hard X-ray full-field imaging system. In this process, X-rays traverse a precisely rotating sample and strike a thin scintillator screen, which converts the radiation into visible light that is then magnified by an optical microscope and captured by a high-speed sCMOS camera. Thanks to the intense synchrotron radiation, this setup can acquire a complete, high-resolution 3D tomographic scan within seconds, enabling rapid volumetric reconstruction of a material's internal structure. In this case study, a set of reef-building corals (with a tissue-covered calcium carbonate skeleton) went through this process, resulting in volumetric datasets ranging from 28 GB to 205 GB.

For this case study, and in order for us to be able to visualize the volume data, we have overwritten the Custom Loader (see Figure 2), such that we can correctly ingest data stored in the Hierarchical Data Format version 5 (h5). This is intentionally done in the application layer instead of the engine layer, as it introduces a new dependency into the application. Such additional dependencies would bloat the engine layer over time as more data is supported. We load the data one slice at a time, so that we can correctly handle volumes exceeding the available RAM.

We also created a new *Renderer*, which uses a hybrid rendering pipeline that includes two passes. The entire pipeline is presented in Figure 7. The first step is a visibility pass that utilizes the *Mesh Shading Pipeline* to dynamically handle memory management on the GPU. The output from this pass is the distance to the closest visual virtual bricks. The second pass uses a compute shader to ray march through the volume, but instead of starting at the edge of the volume it starts at the first visible brick. This effectively introduces empty space skipping, allowing the ray marching to start at the surface of the coral rather than the edge of the volume.

We maintain the high resolution of the volume through the use

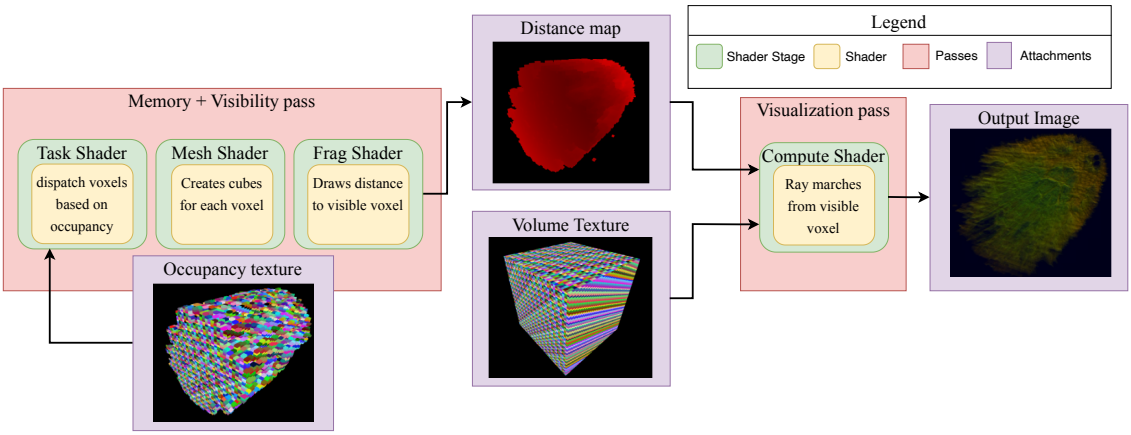

Figure 7: *The custom rendering pipeline for the visualization of the scanned coral, as implemented via the Custom Renderer interface (see Figure 2). This architecture overwrites the default engine pipeline to facilitate out-of-core volumetric data. It is divided into two passes: a visibility pass that manages a custom virtual bricking memory model, and a subsequent visualization pass that renders the volume data.*

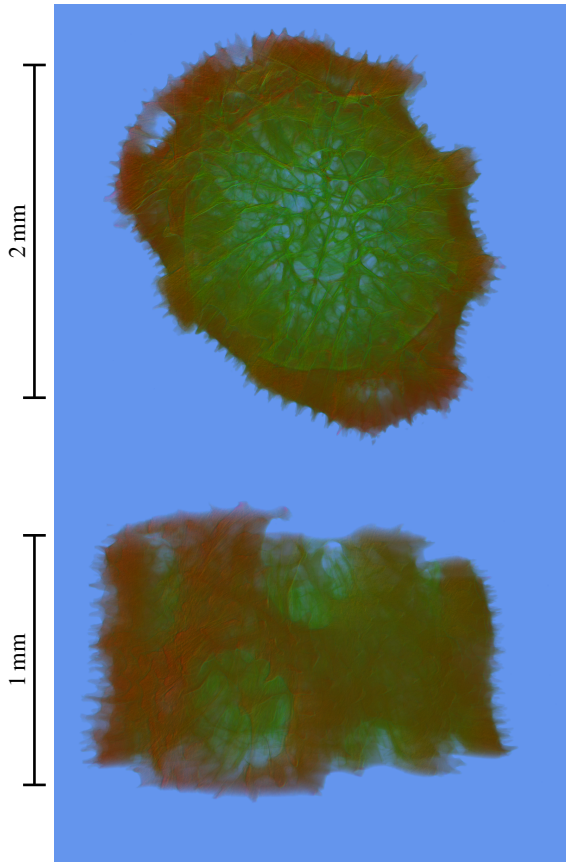

Figure 8: *Visualizations of the inner and outer structure of a Stylophora coral. The view of the internal structure (top row) is from the bottom and up in the outer structure visualization (bottom row). The dimensions of the volume is 1062×2664×2664, and consists of 7,536,903,552 voxels (28 GB).*

of virtual bricking. This enables us to map the entire volume to virtual memory addresses without having to actually allocate physical memory for it. We are further able to mark the 3D texture as using non-contiguous memory, which gives us the ability to more easily fill up the GPU's VRAM, as we do not need to worry about fragmentation or VRAM usage of other programs. Then we divided the volume into virtual bricks consisting of 64x32x32 voxels. All voxels of each block are explored to see if they contain noise, or actual information about the coral. This alone allows us to disregard about half of the voxels on average. For the coral in Figure 8, the resulting number of bricks was 119,952, and of these 58,442 were deemed to contain actual information, the rest contained only host medium and did not need to be uploaded. We stream the remaining voxels to the GPU on demand. This is handled by the visibility pass. For each virtual brick we dispatch a task shader, which checks if the brick is visible. If visible, we check whether it resides in memory or should be requested. If a brick has not been seen for a couple of frames, we mark it as unloaded and its physical memory is returned to the pool to allow other bricks to take its physical memory.

Another visualization example from this dataset is in Figure 1, where the scanned volume is 1.2 mm on each side and has a resolution of 2144×2176×2176. The voxels thus have a physical size of around 0.55 μm. The scan captures an individual polyp of a stylophora coral and some of the bone structure supporting it. For this example, our goal is to enable interactive adjustment of the visualization to ease inspection of both bone and soft tissues.

6. Discussion

Working with two very different types of data demonstrates the flexibility of Jinsoku and its ability to be customized by exposing only the essential components required for dataset ingestion. While this comes at the cost of a steeper learning curve and the need for a deeper understanding of computer graphics, such expertise is often necessary when operating with large and unique datasets that are incompatible with GPGEs without substantial rewrites and exten-

sions. We opt to properly facilitate the data instead of simplifying it. Simplification is rarely appropriate for scientific data.

For Case Study 1 (optimized wing), the datasets could be visualized without adding a Custom Renderer, as the data was already in a supported format. This enabled us to focus on the visualization, while Jinsoku was used almost out of the box. Thus, we avoided setting up hardware extensions and adding them into the rendering. Furthermore, since Jinsoku allows us to overwrite the standard memory model, we could alias the memory for the presentation in VR effectively, enabling us to use MVR. Here, we gained significant performance over the GPGEs, and this frees up precious time for potential interactions with the models, enabling researchers to mark or annotate the model as they explore it. Alternatively, the freed up time can be used to create a more demanding visualization. The size of the model leads to a divergence from GPGEs, where the focus is to handle several smaller models simultaneously rather than one large model. While these engines have VR support, this is also not the main focus, which leaves some performance on the table. We were able to pick up this performance through use of a thinner Application Layer combined with direct access to modern hardware features such as MVR and VRS. The improvement offered by VRS may well be larger if the surface illumination model is more complex than the Phong shader we used. This is evident from the observation that the rendering time does not decrease when inspecting the mesh up close for any of our three most optimized solutions. The reason is most likely that we have managed to shift the bottleneck in the rendering elsewhere in the pipeline.

For Case Study 2 (coral volume data), more work went into the customization, as it required adding support for a new file format and introducing a new hybrid renderer that utilized two hardware pathways: compute and rasterization. We also had to introduce a custom memory model to enable out of core voxel streaming, as the datasets exceeded the physical memory available on the GPUs we had available. The "gap" here is not a lack of rendering capability per se, but a misalignment between the generic memory management strategies of commercial tools and the specific, uncompromising requirements of high-density scientific data. Making the decision to include new dependencies directly in the Application Layer allows us to keep the Engine Layer thin and easier to get up and running for future users.

We tested our system on two datasets with markedly different characteristics and visualization requirements. In both cases, Jinsoku proved useful. Instead of having to build the full abstraction layers ourselves, we only needed to override the parts that interface directly with the data. In Case Study 1, little extension was needed. In Case Study 2, an entire custom renderer was created specifically for the data. Nevertheless, both cases benefited from some of the boilerplate functionality provided by the engine. While the visualization engine shifts some of the required expertise onto the user, this is almost unavoidable as novel acquisition methods and more specialized data formats continue to emerge for better capture and compression of scientific data.

7. Conclusion

We presented a flexible visualization engine capable of supporting different scientific data formats. The modular software design enables extension of Jinsoku to accommodate new data types without requiring extensive rewrites of the components. Although this approach places greater demands on users when their data deviates significantly from standardized formats, the framework also enables them to work with common formats while still leveraging the newest hardware features. The resulting improvements in rendering performance free up resources for more complex visualizations or interactions. General purpose game engines tend to prioritize the hardware features that address the needs of their larger user base. As the gap widens between conventional game-oriented data and the increasingly specialized requirements of scientific datasets, the likelihood of game engines fulfilling all the needs diminishes. By preparing Jinsoku for the two case studies, we strengthened its capabilities and moved it toward becoming a more robust and well-rounded tool for scientific visualization. We hope future adoption of Jinsoku will continue this process.

7.1. Acknowledgment

This research was supported by Advokat Bent Thorbergs Fond (ref. 66.531). This work was supported by a research grant (VIL57413) from VILLUM FONDEN. We acknowledge the MAX IV Laboratory for beamtime on the DanMAX beamline under proposal 20241377. Research conducted at MAX IV, a Swedish national user facility, is supported by Vetenskapsrådet (Swedish Research Council, VR) under contract 2018-07152, Vinnova (Swedish Governmental Agency for Innovation Systems) under contract 2018-04969 and Formas under contract 2019-02496. DanMAX is funded by the NUFI grant no. 4059-00009B.

References

[AALS17] AAGE, NIELS, ANDREASSEN, ERIK, LAZAROV, BOYAN S., and SIGMUND, OLE. "Giga-voxel computational morphogenesis for structural design". *Nature* 550.7674 (2017), 84–86. DOI: 10.1038/nature23911 5.

[AHH*18] AKENINE-MÖLLER, TOMAS, HAINES, ERIC, HOFFMAN, NATY, PESCE, ANGELO, IWANICKI, MICHAŁ, and HILLAIRE, SÉBASTIEN. *Real-Time Rendering*. fourth edition. A K Peters / CRC Press, 2018 4.

[ASLA20] AAGE, NIELS, SIGMUND, OLE, LAZAROV, BOYAN B., and ANDREASSEN, ERIK. *TopWingData*. Dataset. Technical University of Denmark, 2020. DOI: 10.11583/dtu.12581615.v1 5.

[Bai18] BAILEY, MIKE. "Introduction to the Vulkan graphics API". *SIGGRAPH Asia 2018 Courses*. ACM, 2018. DOI: 10.1145/3277644.3277800 3.

[Ber25] BEREGOVYI, KINDRAT. "Focal Engine: Filling a Gap Between Game Engines and Big-Data Scientific Visualization Packages". PhD thesis. University of New Hampshire, 2025. URL: https://scholars.unh.edu/dissertation/2961/ 2.

[CGS*13] CHILDS, HANK, GEVECI, BERK, SCHROEDER, WILL, MEREDITH, JEREMY, MORELAND, KENNETH, SEWELL, CHRISTOPHER, KUHLEN, TORSTEN, and BETHEL, E. WES. "Research Challenges for Visualization Software". *Computer* 46.5 (2013), 34–42. DOI: 10.1109/MC.2013.179 2.

[DDC*14] DONALEK, CIRO, DJORGOVSKI, S. G., CIOC, ALEX, WANG, ANWELL, ZHANG, JERRY, LAWLER, ELIZABETH, YEH, STACY, MAHABAL, ASHISH, GRAHAM, MATTHEW, DRAKE, ANDREW, et al. "Immersive and collaborative data visualization using virtual reality platforms". *Proceedings Big Data*. IEEE, 2014, 609–614. DOI: 10.1109/BigData.2014.7004282 2.

[EGC*20] EL BEHEIRY, MOHAMED, GODARD, CHARLOTTE, CAPORAL, CLÉMENT, MARCON, VALENTIN, OSTERTAG, CÉCILIA, SLITI, OUMAIMA, DOUTRELIGNE, SÉBASTIEN, FOURNIER, STÉPHANE, HAJJ, BASSAM, DAHAN, MAXIME, et al. "DIVA: Natural Navigation Inside 3D Images Using Virtual Reality". *Journal of Molecular Biology* 432.16 (2020), 4745–4749. DOI: 10.1016/j.jmb.2020.05.026 2.

[GCM*15] GONIZZI BARSANTI, SARA, CARUSO, GIANDOMENICO, MICOLI, L. L., COVARRUBIAS RODRIGUEZ, M., and GUIDI, GABRIELE. "3D visualization of cultural heritage artefacts with virtual reality devices". *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XL-5/W7 (2015), 165–172. DOI: 10.5194/isprsarchives-XL-5-W7-165-2015 3.

[GK19] GARCÍA-HERNÁNDEZ, RUBÉN JESÚS and KRANZLMÜLLER, DIETER. "NOMAD VR: Multiplatform virtual reality viewer for chemistry simulations". *Computer Physics Communications* 237 (2019), 230–237. DOI: 10.1016/j.cpc.2018.11.013 3.

[Gre19] GREGORY, JASON. *Game Engine Architecture*. 3rd. A K Peters / CRC Press, 2019. DOI: 10.1201/9781315106946 4.

[HBB*15] HELBIG, CAROLIN, BILKE, LARS, BAUER, HANS-STEFAN, BÖTTINGER, MICHAEL, and KOLDITZ, OLAF. "MEVA - An Interactive Visualization Application for Validation of Multifaceted Meteorological Data with Multiple 3D Devices". *PLoS ONE* 10.4 (2015), e0123811. DOI: 10.1371/journal.pone.0123811 2.

[JAG19] JOURDAIN, SEBASTIEN, AYACHIT, UTKARSH, and GEVECI, BERK. "HPC Enabled Immersive and non-Immersive Visualization of Large-Scale Geo-Scientific Data". *Proceedings of SPIE - The International Society for Optical Engineering*. Vol. 11013. 2019, 110130P. DOI: 10.1117/12.2519315 2.

[JFB23] JENSEN, MARK BO, FRISVAD, JEPPE REVALL, and BÆRENTZEN, J. ANDREAS. "Performance Comparison of Meshlet Generation Strategies". *Journal of Computer Graphics Techniques (JCGT)* 12.2 (Dec. 2023), 1–27. ISSN: 2331-7418. URL: http://jcgt.org/published/0012/02/01/ 6.

[JJFB21] JENSEN, MARK B., JACOBSEN, EGILL I., FRISVAD, JEPPE REVALL, and BÆRENTZEN, J. ANDREAS. "Tools for Virtual Reality Visualization of Highly Detailed Meshes". *VisGap - The Gap between Visualization Research and Visualization Software*. The Eurographics Association, 2021. DOI: 10.2312/visgap.20211088 2, 6.

[JMR17] JIMÉNEZ FERNÁNDEZ-PALACIOS, BELEN, MORABITO, DANIELE, and REMONDINO, FABIO. "Access to complex reality-based 3D models using virtual reality solutions". *Journal of Cultural Heritage* 23 (2017), 40–48. DOI: 10.1016/j.culher.2016.09.003 3.

[KGKG24] KRÜGER, MARCEL, GILBERT, DAVID, KUHLEN, TORSTEN W., and GERRITS, TIM. "Game Engines for Immersive Visualization: Using Unreal Engine Beyond Entertainment". *PRESENCE: Virtual and Augmented Reality* 33 (July 2024), 31–55. DOI: 10.1162/pres_a_00416 2.

[KH13] KEHRER, JOHANNES and HAUSER, HELWIG. "Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey". *IEEE Transactions on Visualization and Computer Graphics* 19.3 (2013), 495–513. DOI: 10.1109/TVCG.2012.110 1.

[KJK*15] KIM, KYUNGYOON, JACKSON, BRET, KARAMOUZAS, IOANNIS, ADEAGBO, MOSES, GUY, STEPHEN J., GRAFF, RICHARD, and KEEFE, DANIEL F. "Bema: A multimodal interface for expert experiential analysis of political assemblies at the Pnyx in ancient Greece". *Symposium on 3D User Interfaces (3DUI)*. IEEE, 2015, 19–26. DOI: 10.1109/3DUI.2015.7131720 3.

[Kub18a] KUBISCH, CHRISTOPH. *Introduction to Turing Mesh Shaders*. NVIDIA Developer Technical Blog. Sept. 2018. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/ 3.

[Kub18b] KUBISCH, CHRISTOPH. *Vulkan & OpenGL CAD Mesh Shader Sample*. Accessed: 2022-04-16. 2018. URL: https://github.com/nvpro-samples/gl_vk_meshlet_cadscene 3.

[LK10] LAINE, SAMULI and KARRAS, TERO. "Efficient Sparse Voxel Octrees". *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. 2010, 55–63. DOI: 10.1145/1730804.1730814 3.

[MAJ*22] MESSER, DOLORES, ATCHAPERO, MICHAEL, JENSEN, MARK B., SVENDSEN, MICHELLE S., GALATIUS, ANDERS, OLSEN, MORTEN T., FRISVAD, JEPPE R., DAHL, VEDRANA A., CONRADSEN, KNUT, DAHL, ANDERS B, et al. "Using virtual reality for anatomical landmark annotation in geometric morphometrics". *PeerJ* 10 (2022), e12869. DOI: 10.7717/peerj.12869 2.

[Mar13] MARX, VIVIEN. "Biology: The big challenges of big data". *Nature* 498.7453 (2013), 255–260. DOI: 10.1038/498255a 1.

[MEC14] MARKS, STEFAN, ESTEVEZ, JAVIER, and CONNOR, ANDY. "Towards the Holodeck: Fully Immersive Virtual Reality Visualisation of Scientific and Engineering Data". *Proceedings of Image and Vision Computing New Zealand (IVCNZ)*. 2014, 42–47. DOI: 10.1145/2683405.2683424 3.

[MGHK15] MORAN, A., GADEPALLY, V., HUBBELL, M., and KEPNER, J. "Improving Big Data visual analytics with interactive virtual reality". *IEEE High Performance Extreme Computing Conference (HPEC 2015)*. 2015, 1–6. DOI: 10.1109/HPEC.2015.7322473 2.

[MS16] MISHRA, PRERNA and SHRAWANKAR, URMILA. "Comparison Between Famous Game Engines and Eminent Games". *International Journal of Interactive Multimedia and Artificial Intelligence* 4.1 (2016), 69–77. DOI: 10.9781/ijimai.2016.4113 2.

[Mus13] MUSETH, KEN. "VDB: High-Resolution Sparse Volumes with Dynamic Topology". *ACM Transactions on Graphics (TOG)* 32.3 (2013), 27:1–27:22. DOI: 10.1145/2487228.2487235 3.

[Pho75] PHONG, BUI TUONG. "Illumination for Computer-generated Pictures". *Communications of the ACM* 18.6 (June 1975), 311–317. DOI: 10.1145/360825.360839 6.

[SDC07] SOLDATI, MARCO, DOULIS, MARIO, and CSILLAGHY, ANDRE. "SphereViz - Data Exploration in a Virtual Reality Environment". *2007 11th International Conference Information Visualization (IV '07)*. 2007, 680–683. DOI: 10.1109/IV.2007.105 2.

[SLS18] STEFANI, CAROLINE, LACY-HULBERT, ADAM, and SKILLMAN, THOMAS. "ConfocalVR: Immersive Visualization for Confocal Microscopy". *Journal of Molecular Biology* 430.21 (2018), 4028–4035. DOI: https://doi.org/10.1016/j.jmb.2018.06.035 2.

[SW15] SCHÜTZ, MARKUS and WIMMER, MICHAEL. "High-Quality Point-Based Rendering Using Fast Single-Pass Interpolation". *Digital Heritage*. IEEE, 2015. DOI: 10.1109/DigitalHeritage.2015.7419525 3.

[UKF*18] USHER, WILL, KLACANSKY, PAVOL, FEDERER, FREDERICK, BREMER, PEER-TIMO, KNOLL, AARON, YARCH, JEFF, ANGELUCCI, ALESSANDRA, and PASCUCCI, VALERIO. "A Virtual Reality Visualization Tool for Neuron Tracing". *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), 994–1003. DOI: 10.1109/TVCG.2017.2744079 2.

[WLL*19] WANG, YIMIN, LI, QI, LIU, LIJUAN, ZHOU, ZHI, RUAN, ZONGCAI, KONG, LINGSHENG, LI, YAOYAO, WANG, YUN, ZHONG, NING, CHAI, RENJIE, et al. "TeraVR empowers precise reconstruction of complete 3-D neuronal morphology in the whole brain". *Nature communications* 10.1 (2019), 1–9 3.

[Wol19] WOLFARTSBERGER, JOSEF. "Analyzing the potential of Virtual Reality for engineering design review". *Automation in Construction* 104 (2019), 27–37. DOI: https://doi.org/10.1016/j.autcon.2019.03.018 3.

[ZJAW25] ZELLMANN, STEFAN, JAROS, M., AMSTUTZ, J., and WALD, INGO. "GPU Volume Rendering with Hierarchical Compression Using VDB". *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. The Eurographics Association, 2025 2, 3.