J Vis (2024) 27:1143–1169
https://doi.org/10.1007/s12650-024-01011-y

REGULAR PAPER

Sergej Geringer · **Florian Geiselhart** · **Alex Bäuerle** · **Dominik Dec** · **Olivia Odenthal** · **Guido Reina** · **Timo Ropinski** · **Daniel Weiskopf**

mint: Integrating scientific visualizations into virtual reality

Received: 29 February 2024 / Accepted: 10 April 2024 / Published online: 5 August 2024

Abstract We present an image-based approach to integrate state-of-the-art scientific visualization into virtual reality (VR) environments: the mint visualization/VR inter-operation system. We enable the integration of visualization algorithms from within their software frameworks directly into VR without the need to explicitly port visualization implementations to the underlying VR framework—thus retaining their capabilities, specializations, and optimizations. Consequently, our approach also facilitates enriching VR-based scientific data exploration with established or novel VR immersion and interaction techniques available in VR authoring tools. The separation of concerns enables researchers and users in different domains, like virtual immersive environments, immersive analytics, and scientific visualization, to independently work with existing software suitable for their domain while being able to interface with one another easily. We present our system architecture and inter-operation protocol (mint), an example of a collaborative VR environment implemented in the Unity engine (VRAUKE), as well as the integration of the protocol for the visualization frameworks Inviwo, MegaMol, and ParaView. Our implementation is publicly available as open-source software.

Keywords Scientific visualization framework · Virtual reality environment · Texture sharing · Inter-process communication · Immersive analytics

1 Introduction

Today, the domains of scientific visualization software and virtual reality (VR) applications mainly operate on different levels of abstraction and have different goals. The interconnection of both enables immersive data analysis, which is researched in *immersive analytics* (Chandler et al. 2015; Fonnet and Prié 2021; Ens et al. 2021). This interconnection is desirable as it enables exploration of a broader design space with use cases in, e.g., collaborative exploration of data (Prodromou et al. 2020), educational applications (Kaufmann and Schmalstieg 2002), or even artistic approaches to visualized data (Keefe et al. 2008). Although, at first glance, the interests of the visualization and VR domains seem to overlap with regard to maximized performance, their thresholds for *interactivity* are quite different. For example, interactivity in visualization

S. Geringer (✉) · D. Dec · O. Odenthal · G. Reina · D. Weiskopf
University of Stuttgart, Allmandring 19, 70569 Stuttgart, Germany
E-mail: Sergej.Geringer@visus.uni-stuttgart.de

F. Geiselhart
Schwäbisch Gmünd University of Design, Rektor-Klaus-Straße 100, 73525 Schwäbisch Gmünd, Germany

F. Geiselhart · A. Bäuerle · T. Ropinski
Ulm University, James-Franck-Ring, 89081 Ulm, Germany

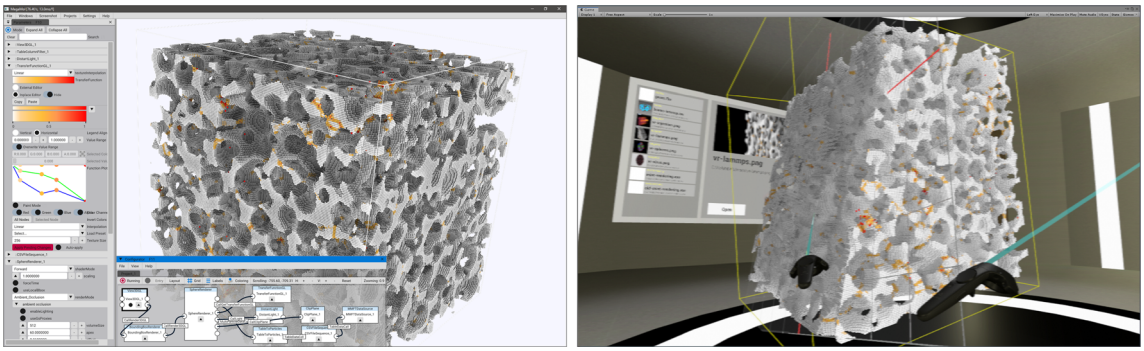


Fig. 1 Our system *mint* provides a generic way to integrate desktop-based scientific visualization frameworks like MegaMol (left) into easy-to-use collaborative VR environments, such as our *VRAUKE* system (right). Both scientific visualizations and VR environments can be developed in their respective software ecosystems using abstractions best suitable for their domain. In this example, the visualizations show a LAMMPS (Thompson et al. 2022) peridynamics simulation of exertion of force on a porous ceramic (dataset size 146 MB). Particles are colored by damage (yellow to red), measured in the number of broken bonds to neighboring particles. Note the clusters of cracks at the center bottom in VR. Ambient occlusion is applied for better visibility of the spatial structures. Dataset courtesy of Vinzenz Guski

could mean as low as ten frames per second (FPS), whereas current VR research recommends targeting 120 FPS or more if possible (Wang et al. 2023).

Visualization researchers want to ensure that their algorithms scale to relevant data sizes and enable users to gain insights from their data by using appropriate and, sometimes, novel representations. Abstractions are added when necessary, e.g., to address perceptual limitations (clutter) and performance issues (interactivity). To optimize for these needs, scientific visualization software often relies on low-level software environments that make use of hardware acceleration, built on *C++* and *OpenGL* (Humphrey et al. 1996; Bruckner and Gröller 2005; Hanwell et al. 2015) or similar Application Programming Interfaces (APIs) like *CUDA* or *Vulkan*. In contrast, VR research is predominantly characterized by topics related to the human user, such as perception, immersion, presence, and interaction. The goal in this field of research is to create an immersive, user-centered experience that users can navigate and interact with in a natural way, often comparable to physical reality while providing an increased richness compared to a 2D display. In turn, the prevalent software frameworks (or software ecosystems) in these two domains make different, sometimes opposing, choices regarding their focused user groups, use cases, goals, domain-specific metaphors and abstractions, and optimizations.

To bridge this gap and combine the benefits of scientific visualization frameworks with interactive VR environments, we propose **mint** (**m**inimal **int**erop), a software framework for inter-process rendering. Using a generic interface, *mint* loosely couples systems that rely on low-level programming, such as existing visualization frameworks, with systems that offer rich functionality and customizability at a higher level of abstraction, e.g., VR frameworks or game engines. Please note that we use the terms "VR framework," "VR authoring tool," and "game engine" interchangeably in this manuscript. We achieve loose coupling by integrating the rendered images of scientific visualizations into VR, with only little overhead with regard to both performance and implementation effort. We further present an example of a VR environment that receives and integrates these rendered visualizations, **VRAUKE** (***V**isualization **R**enderer-**A**ugmented **U**nity **K**olab **E**nvironment*). *VRAUKE* showcases a laboratory-like setting on top of the Unity engine that implements intuitive, hands-on interaction and collaboration features for exploring scientific datasets (Fig. 1).

Together, *mint* and *VRAUKE* combine the rapid prototyping capabilities of VR authoring tools and the advanced visualization algorithms of specialized visualization frameworks to facilitate scientific data analysis in VR. Due to the generic and loose coupling of the visualization and VR environments, our system is easier to maintain, more adaptable, and more stable than integrated solutions. Furthermore, it enables visualization researchers and human-computer interaction (HCI) researchers to use the appropriate tools to tackle the challenges in their field (Jönsson et al. 2019). Our evaluation of the resulting system shows that the interconnection of software systems from both domains requires reasonable integration effort, the resulting VR experience is usable and customizable, both for novice users and domain experts, while introducing inter-processing rendering overhead (lag) within acceptable bounds.

In summary, the contributions of this paper are:

1. *mint*, a system architecture that interconnects visualization frameworks and VR environments with little overhead,
2. *VRAUKE*, an example of a VR environment implementation as a feasibility study that showcases scenarios for which *mint* can be used,
3. an evaluation of the introduced rendering overhead, quantified as the lag between visualization generation and its presentation in VR, and
4. a discussion of the effort required to enable VR for existing visualization software using *mint*.

We open-source the *mint* and *VRAUKE* codes at https://github.com/UniStuttgart-VISUS/MWK-mint.

2 Related work

VR has been leveraged for analyzing immersive scientific data in different domains, such as math education (Kaufmann and Schmalstieg 2002), geoscience (Kreylos et al. 2006), medical data analysis (Egger et al. 2017; Wheeler et al. 2018; Prodromou et al. 2020), and chemical simulations (García-Hernández and Kranzlmüller 2019). In a broader context, there is a trend toward combining VR with visualization and visual analytics to arrive at *immersive analytics* (Marriott et al. 2018; Chandler et al. 2015; Fonnet and Prié 2021; Ens et al. 2021) and integrating interactive visualization in AR for *situated analytics* (Fleck et al. 2022).

2.1 Combining VR and visualization software

In this paper, we focus specifically on linking scientific visualization and VR. Approaches to integrating both usually implement either of the domains directly in a software framework of the other. Visualization algorithms are either ported to game engines that already provide VR integration (Doutreligne et al. 2014, User "mlavik1" on GitHub, 2023), or low-level VR hardware support (like OpenVR, OpenXR) is added to visualization frameworks (O'Leary et al. 2017; Shetty et al. 2011; Egger et al. 2017; Cordeil et al. 2019; García-Hernández and Kranzlmüller 2019; Prodromou et al. 2020; Günther et al. 2019). While this tight integration provides the optimal environment for one of the two domains, the other is generally not fully supported, less flexible, or not as mature.

An effective decoupling of VR and visualization frame rates is achieved by Shaw et al. (1993), in the form of the Decoupled Simulation Model (DSM) system architecture for VR rendering across multiple processes and machines connected via a network, along with the MR Toolkit as a framework to develop VR applications. ISVAS-VR by Haase (1996) is a symbiosis of a full-featured VR system and scientific visualization framework, both running as separate processes on the same machine using Unix pipes and shared memory for inter-process communication. Haase distinguishes five classes of possible integration/interconnection between VR and visualization systems, of which the loose coupling (via messages) of mature VR and visualization systems is the most capable one in terms of available features. VIVRE (Boyd et al. 1999a, b) provides a similar coupling of VR and visualization systems, but the visualization system passes geometry primitives to the coupled virtual environment for rendering. Similarly, Fuhrmann et al. (1997) implement the Studierstube AR system as separate display and visualization processes, where the latter provides updated geometry data based on modified visualization parameters. Since both approaches are geometry-based, they are limited in their generality with regard to rendering techniques. Our system falls into the loose coupling categorization by Haase, but since it is image-based instead of geometry-based, it is generic across visualization frameworks and techniques.

Recent work by Wheeler et al. (2018) and Prodromou et al. (2020) integrates the Visualization Toolkit (VTK) (Schroeder et al. 2006) with the Unity engine by sharing an OpenGL context between Unity and VTK. This is done by wrapping VTK in a Unity native plugin, effectively executing VTK from inside Unity and rendering into the Unity framebuffer. UnityMol (Doutreligne et al. 2014), DXR (Sicat et al. 2019), and the Immersive Analytics Toolkit (IATK) (Cordeil et al. 2019) implement algorithms for molecular and data visualization directly in the Unity engine, citing it as an advanced but also accessible authoring tool for interactive applications. Our work shares part of the intention behind RagRug (Fleck et al. 2022) in the sense that we aim to have VR and interaction as accessible as possible, but we still want to keep the low-level control that visualization developers require. Technically, the aforementioned methods that integrate visualizations into VR place the visualization computation inside the main render loop of the VR engine. The main drawback of this approach is that visualization rendering has to render at a performance level that

does not compromise the targeted FPS in VR environments (Wang et al. 2023). Otherwise, the VR render loop will stall, and a user in the VR session may experience discomfort, disorientation, or sickness (Stauffert et al. 2020; Liu and Heer 2014). Further, porting existing visualization implementations to VR frameworks may be time-consuming and error-prone. We want to enable researchers to use available software tools directly and thus focus on actual research questions and investigations they are interested in.

2.2 OpenGL interception and manipulation

Intercepting and manipulating the OpenGL command stream by replacing dynamically loaded libraries has been demonstrated (Stegmaier et al. 2002; Mohr and Gleicher 2002) as a feasible approach to extend use cases of existing OpenGL applications where the source code is not available for modification. This technique intercepts all OpenGL API calls, modifies data or parameters where necessary, and passes the result to the original OpenGL driver provided by the graphics hardware vendor. Initially, this method was used to apply stylized rendering techniques to the original rendering (Mohr and Gleicher 2002), but was later extended to connect existing OpenGL applications to VR environments for MATLAB visualizations and traffic simulations (Zielinski et al. 2013, 2014; Donatiello et al. 2021), as well as for distributed rendering over the network and on tiled displays (Humphreys et al. 2001; Stegmaier et al. 2002; Marino et al. 2007; Doerr and Kuester 2011). Commercial products like *TechViz*[1] (Verhille et al. 2014) and *moreViz*[2] based on this principle target industrial and engineering use cases such as design review of Computer Aided Design (CAD) data in VR.

However, such OpenGL interception techniques need to rely on predefined rendering semantics, e.g., *glMatrixMode(GL_PROJECTION)* in OpenGL *immediate mode*, to locate relevant data in the command stream. In contrast, modern *Core Profile* OpenGL is designed to give the graphics programmer more freedom over the semantics of their rendering data. For example, scene camera parameters and resulting matrices may be constructed from uniform (generic) shader input parameters entirely on the graphics processing unit (GPU) during shader program execution. Similarly, triangle geometry used for rendering may be constructed by geometry, tessellation, or even compute shaders and dispatched for rendering indirectly, all during shader execution on the GPU.[3] Such features allow for maximizing GPU resource- and compute utilization, leading to better performance and scalability. However, it may be hard or even impossible for an intercepting OpenGL driver to track information like camera state to emulate stereo rendering for a VR environment. Furthermore, visualization software may rely on integrating other hardware acceleration APIs, like CUDA or OptiX, which may be harder to intercept and emulate but play an important role in realizing scalable visualization algorithms (Richer et al. 2022).

2.3 Scientific visualization frameworks

Scientific visualization aims to transform input data, i.e., a *dataset* stemming from measurements or simulations, to output in the form of images. Visualization frameworks can have a range of use cases, such as domain-specific applications (Grottel et al. 2015; Gralka et al. 2019; Doutreligne et al. 2014; Bruckner and Gröller 2005; Jönsson et al. 2019), large data and distributed computation support (Childs et al. 2005; Moreland et al. 2016; Wald et al. 2017; Günther et al. 2019), and generic visualization APIs (Bostock and Heer 2009; Ahrens et al. 2005; Schroeder et al. 2006; Cordeil et al. 2019).

In this work, we focus on scientific visualization systems that run interactively on desktop machines. Such systems resort to hardware acceleration to scale to large datasets. They commonly employ a GPU via OpenGL (Hanwell et al. 2015; Bruckner and Gröller 2005; Grottel et al. 2015; Jönsson et al. 2019), or use parallelism and low-level primitives on CPUs (Wald et al. 2017) and dedicated hardware accelerators (Moreland et al. 2016). Additionally, they take advantage of acceleration data structures tailored to the properties of the rendered datasets (Wald et al. 2015; Gupta et al. 2023). The common theme for these systems is that the design and implementation of even one visualization system for a given scope of use is a big undertaking, requiring expertise, developer resources, and time, with the goal of amortizing this investment over a longer software life cycle (Reina et al. 2020). As new use cases arise, such as integrating visualization frameworks into immersive environments (Shetty et al. 2011; O'Leary et al. 2017),

[1] https://www.techviz.net.

[2] https://www.more3d.com.

[3] OpenGL Indirect Rendering, https://www.khronos.org/opengl/wiki/Vertex_Rendering#Indirect_rendering.

repurposing established software architectures that proved to work for scientific visualization may present significant challenges. At the same time, designing and implementing a new visualization solution from the ground up for immersive environments may be unreasonable. Therefore, we specifically aim to reduce the effort required to extend existing solutions to support immersive use.

2.4 Impostor and remote rendering

Our approach is technically similar to a remote rendering system, where a server process provides renderable primitives to a rendering client via a communication layer (Shi et al. 2012; Shi and Hsu 2015; Mueller et al. 2018). However, in our system, the visualization and VR processes run on the same machine, whereas remote rendering systems often distribute rendering over a network.

Our approach to integrating visualization outputs into the VR environment resembles impostor objects textured with the visualization. Accelerating rendering performance by approximating complex geometry with images has been proposed by Aliaga (1996) and used in VR environments by Schaufler and Stürzlinger (1996). Dynamically, but asynchronously, updating the image for an impostor (e.g., by the rendering process) can save processing resources (Decoret et al. 1999) and has been used to improve rendering performance in VR environments (Schaufler 1996; Misiak et al. 2021).

3 Bridging visualization and VR authoring tools

In the following, we first outline use cases for integrating scientific visualizations into VR environments. To some degree, such use cases have also been covered by research in immersive analytics (Marriott et al. 2018; Chandler et al. 2015; Fonnet and Prié 2021; Ens et al. 2021). In this discussion, we focus on the practical implications of these use cases for end-users and software developers, i.e., how the usability paradigms and abstractions of the respective software influence the involved workflows. In comparing the different workflows, we point out the gap between the software ecosystems in the two domains that our system bridges.

3.1 Use cases

The use cases employ virtual environment authoring software, e.g., Unity, to create audience-tailored experiences or augment the capabilities of existing visualization frameworks. Potential audiences, or users, encompass a wide range of personas, e.g., children in museum exhibit contexts, the general public, domain experts (e.g., medical professionals), or visualization experts.

3.1.1 Visualization data exploration

VR-enabled visualization offers advantages when exploring spatial data, as observed, for example, with molecular data (Kozlikova et al. 2017), where spatial structure gives important clues about function. Besides immersive analytics scenarios, VR exploration can also be used in science communication to non-homogeneous audiences, both the general public and boards of experts alike. The goals in such scenarios can be to present correct and precise visualizations to users and not to overwhelm them with the user interface and interaction complexity. A simplified user interface (UI) compared to a desktop visualization (Ynnerman et al. 2016) can benefit both inexperienced users and domain experts.

3.1.2 Remote collaboration

Exploring visualizations in a collaborative setting can help build insights, e.g., for collaboration of domain scientists and data visualization experts, or even between expert and non-expert collaborators, e.g., a medical professional wanting to explain data to a patient (McGhee et al. 2015). VR authoring tools provide software packages for such remote and local collaboration, i.e., multiplayer gaming libraries and infrastructure, designed to integrate well with the engine and related workflows.

3.1.3 Higher education

Digital models or datasets are used in higher education to illustrate facts and foster better understanding, e.g., in medicine, engineering, or molecular biology. To gain insights related to the learning objective, students may benefit from collectively and collaboratively exploring datasets using an accessible visualization environment that *gets out of the way* in terms of usability and interaction, lets users focus on the data, and at the same time is engaging and fun.

3.1.4 Exhibits

Interactive exhibits in museums employ specifically tailored hardware/software setups to provide visitors with in-depth information on a topic. VR setups are becoming more common for such immersive exhibits. However, real-time data exploration of complex datasets is often impossible without building specific interactive software for this (Ynnerman et al. 2016; Ahsan et al. 2022). This is because interactivity, storytelling, scripting, and composition play significant roles in such an exhibit context. Usually, such features are outside the focus of visualization tools. At the same time, showing real-world datasets to users is highly desirable, but VR authoring tools rarely support the visualization of those.

3.2 Virtual (reality) environment authoring tools

Tools like Unity[4] or Unreal Engine[5] are not just being used for developing games anymore. Instead, they are also increasingly applied to create user-centered, interactive experiences for broad audiences, such as in use cases discussed above, for serious games, and even training-related software (Checa and Bustillo 2020).

3.2.1 Frameworks and tools

While software development using these tools sometimes involves writing code to define details of game logic, it usually relies on graphical editing tools. This set of features facilitates rapid prototyping and an agile development process that leads to faster feature iterations compared to a fully programming-based process. As such, game engines can be seen as *frameworks* themselves, as they help handle common challenges in developing interactive software, e.g., scene rendering and illumination, authoring of animations, or input handling. From a software developer's perspective, frameworks like Unity partially shift the responsibility regarding performance and optimization toward the engine. A drawback of this is that developers must adhere to the restrictions and abstractions the engine framework imposes and are thus limited in their influence on technical details. However, engines also provide usability layers, e.g., by means of graphical editing tools, that enable developers with less or no experience in low-level programming to use optimized state-of-the-art tools for their projects, e.g., manipulating game elements using simple, visual programming-based approaches for physics or animations. Such tools allow experts from different disciplines like 3D artists, user experience (UX) and user interface (UI) designers, and software developers to work closely together by using the same feature set of a game engine. These experts base their workflow largely on *assets*, modular and reusable elements created directly in the editor or imported from external tools, e.g., graphics for a player avatar, UI elements, or animations and transitions between them. Those mechanisms allow experts to focus on visual design, storytelling, and user experience and to test implemented changes immediately.

3.2.2 Developing VR experiences

While building VR experiences is a form of 3D software development using game engines, additional restrictions need to be considered to ensure an acceptable experience for end-users. In VR, rendering performance plays a central role, as degraded performance not only results in a degraded user experience and joy of use but is known to cause physiological effects like motion sickness, disorientation, and increased risk of falling and injury (Dużmańska et al. 2018; Stauffert et al. 2020). Thus, game engines with VR capabilities are designed to prioritize frame rates and may degrade rendering quality where appropriate, e.g.,

[4] https://www.unity.com.

[5] https://www.unrealengine.com.

using foveated rendering, where rendering in the peripheral field of vision is done at a lower resolution. Additionally, input modalities in VR are fundamentally different from those in desktop or laptop computers, nowadays mostly employing hand-held hardware controllers or gestural interaction. This fundamentally changes interaction and locomotion techniques, as traditional point-and-click metaphors from desktop computers and 2D screens are not necessarily optimal anymore.

3.3 Visualization software systems: design and goals

As discussed in Sect. 2, the visualization community has witnessed the development and deployment of several visualization software systems. Most visualization systems are developed with a focus on desktop performance and expert users, e.g., visualization researchers or domain experts. Consequently, optimizing such systems for usability for casual users is often considered out of scope, resulting in a steep learning curve for novice or inexperienced users. To somewhat alleviate this fact, many visualization systems employ modular visual prototyping (Burnett 1999; Koenig et al. 2006) to combine building blocks. These building blocks represent visualization primitives or algorithms, often based on a data flow metaphor (Haeberli 1988; Upson et al. 1989) inspired by the conventional definition of the four stages of the visualization pipeline (Haber and McNabb 1990; Moreland 2013). Visualization building blocks are configurable through parameters, or visualization properties, which define values and ranges of visual attributes, e.g., parameters may configure internal behavior of algorithms by defining texture resolutions, camera parametrization, or transfer functions. Since these parameters can become complex, e.g., in the case of a transfer function, visualization system designers need to make trade-offs regarding their level of abstraction and their place in the design of a visualization system. Further, in the data flow metaphor, system-internal data flow and management of such parameters are decoupled from the dataset to be visualized. The actual dataset is often orders of magnitude larger and especially changed less frequently by the user. Thus, it can be treated as mostly static and stored in privileged memory, usually directly on the GPU. Visualization parameters instead are meant to be manipulated in real-time by the user and thus need to be synchronized and transported inside a visualization system accordingly.

Visualization systems also serve as prototyping environments (Grottel et al. 2015; Jönsson et al. 2019) for research and interdisciplinary collaborations. Collaboration partners often provide datasets in custom data formats, with individual data characteristics relevant to their own research. Visualization researchers may need to write data readers for custom data formats by hand and from scratch so as to fit them correctly into the system architecture and data-flow paradigms of the used visualization system and to allow for optimal data placement and use. For example, to ensure interactive rendering performance in processing big data that does not entirely fit into GPU memory or even system memory (RAM), one may employ on-demand data streaming, automated level-of-detail computation, or utilization of hierarchical data structures. As such, analysis, processing, and rendering of datasets are not only individual to different research collaborations but also tightly coupled with the used and developed visualization algorithms, both on a source code level and on a conceptual level. This is in stark contrast to the aforementioned game engines, which, from a data-processing point of view, are focused on the construction, management, processing, and rendering of triangle-mesh data.

3.4 The gap

As outlined above, game engines provide authoring tools for expert users (artists, designers, game programmers) to quickly prototype and iterate ideas, aiming to optimize for a user-centered and enjoyable experience. The agile workflow employed by such experts is enabled by the abstractions and optimizations toward the usability of the authoring tools themselves. For example, custom 2D or 3D user interfaces for accessibility in data exploration, features for remote collaboration in the form of engine plugins and integration, scripting, and storytelling of engaging experiences for education, as well as composition and presentation for exhibit setups are some of the features that are well-supported by game engines.

On the one hand, artists and designers familiar with such authoring tools are specifically trained and experienced in implementing such user-centric and interaction-focused experiences. On the other hand, visualization systems provide usability abstractions aimed at visualization researchers and similar expert users in terms of visually composing visualization pipelines. However, integrating novel or experimental hardware, such as VR output devices and controllers, supporting multi-camera rendering or collaborative multi-user sessions, or even creating custom user interfaces for simplified data exploration by non-experts

are rarely central in designing and implementing visualization systems. As discussed in Sect. 2, visualization systems can add such features when demand arises, and visualization algorithms can also be successfully implemented in game engines, but there are drawbacks to both approaches. We argue that connecting mature virtual environment authoring tools and visualization frameworks on a system level is, in many cases, a more viable option.

4 mint

This section presents core concepts, the architecture, and implementation details of **mint**. As outlined above, our goal was to design a system where integrating scientific visualizations into VR environments takes little effort while preserving and leveraging the unique capabilities of the involved software frameworks.

In the following, we consider scientific visualizations of datasets with spatial extent, such that any *dataset* is enclosed by a 3D bounding box. We also assume that the VR setup consists of one head-mounted display (HMD), requiring rendering two images for stereoscopic vision. Extending our solution to support 2D visualizations would require only a single image to be transmitted and is not discussed in detail. We evaluate our *mint* implementation and integration into the MegaMol, Inviwo, and ParaView visualization frameworks in Sect. 6.

4.1 System design

Following the considerations in Sect. 3, we want the *mint* system to meet the following requirements:

- **R1** Preserve full capabilities (usability, prototyping, and collaboration functionalities) of VR frameworks.
- **R2** Enable integration of scientific visualizations, as provided by specialized visualization frameworks, without re-implementing visualization algorithms.
- **R3** Ensure loose coupling: integration of **R1** and **R2** should be generic and make minimal assumptions about either software system.

The central design decisions of our system are (1) the inter-connection of the involved visualization and VR frameworks at a process level (as in *operating system process*) using an inter-process communication (IPC) mechanism for data exchange between them, (2) using the VR framework as the user-facing process providing VR integration and user interactions, (3) leveraging visualization frameworks as rendering backends to produce stereoscopic visualization renderings, and (4) running both processes asynchronously, decoupling the respective rendering frame rates.

Thus, *mint* realizes **R3** by running existing VR and visualization software solutions, each as individual, separate operating system processes. Rendering across both processes is coordinated by an IPC mechanism that can be integrated into existing tools with little intrusion. This supports loose coupling between the tools and introduces minimal interference with their existing feature set. An alternative to an IPC mechanism would be to statically or dynamically link the respective software frameworks to each other. We regard this as too limiting, as the involved system architectures may not be designed or not easily adjusted for such a use case (**R3**). For example, O'Leary et al. (2017) implement OpenGL context sharing to use VTK from within VR environments. However, doing so exposes VTK rendering- and application logic to the VR framework, requiring further modifications in VTK to prevent the clearing of color and depth buffers inside VTK from leaking into the VR rendering system.

Using the VR framework as the user-facing application (VR process) naturally realizes **R1**. This implies that visualizations will be *integrated* into the VR environment, ideally considering them simply another asset in the VR scene. Meanwhile, existing visualization software is going to act as a producer, or *renderer*, of visualizations (visualization process), realizing **R2**, while the VR process remotely controls, or *steers*, visualization rendering and fuses the results into the VR environment. We choose an *image-based* mechanism to exchange the final visualization results. Relying on the exchange of geometry data for visualization rendering would not be generic, as visualization frameworks may not immediately produce geometry data for rendering, e.g., in the case of volume rendering. In addition, exchanging image data retains visualization results exactly as they were produced by the visualization process (**R2**). To maintain the performance necessary for interactive VR environments, we use zero-copy on-GPU texture memory sharing between processes (Microsoft 2021; Gold and Subtil 2010).

In summary, the *mint* system is divided into modules (*mint* roles): VR frameworks (*steering*) and visualization frameworks (*rendering*). *mint* modules cooperate over IPC to produce stereoscopic renderings of visualizations and integrate them seamlessly into the VR environment. As examples of how our approach fits into existing visualization solutions, we provide *mint* rendering implementations for the OpenGL-based MegaMol,[6] Inviwo,[7] and ParaView[8] visualization frameworks. We also present the prototypical application *VRAUKE* as a user-facing VR environment integrating *mint* VR steering. Details regarding the setup and reasoning for the *VRAUKE* design are provided in Sect. 5.

4.2 Inter-operation rendering protocol

In the following, we discuss how rendering information is dynamically exchanged via IPC between a VR process (*steering*) and a visualization process (*rendering*) to achieve the integration of a visualization rendering into a VR scene. Listing 1 and Listing 2 in Fig. 2 show C++ pseudo-code as a temporal sequence, illustrating the *mint* inter-operation protocol, i.e., the *mint* API embedded in the render loops of steering and rendering modules and the resulting data flow between them.

We choose to run the VR and visualization processes asynchronously, i.e., *mint::receive* IPC function calls return the latest data available from corresponding *mint::send* calls, without blocking program execution by waiting for newer data to arrive. This decouples the process frame rates and ensures a responsive VR environment at all times. By relieving the visualization process from keeping up with the frame rate necessary for an interactive VR environment, we further reduce the potential to expose the user to effects that induce motion sickness. Lower frame rates in a visualization framework can happen for many reasons. For example, visualization algorithms may have difficulty scaling with dataset properties depending on changing camera positions (Bruder et al. 2020), which can happen easily in an interactive VR setting. However, a blocking mechanism to synchronize frame rates can be integrated into the *mint* backend without impacting the existing integration in VR and visualization frameworks.

Visualization rendering phases

In the following, *dataset* denotes the representation of the scientific data, *visualization result* denotes the resulting image of the visualization step as rendered by the *visualization process*, and *impostor* refers to a scene object representing the dataset in the *VR environment* (e.g., a game object in a Unity VR scene). The impostor object serves as the logical unit responsible for steering visualization rendering. As such, the impostor is used to mediate spatial dataset manipulations (position, orientation (pose), and scale) in VR to the visualization process. Further, the impostor serves as a projection surface for the visualization, i.e., the texture containing the visualization result is used to color the box-shaped impostor geometry, which is cheap to render in the VR scene. Figure 3a depicts the high-level architecture of the *mint* system and Fig. 3b the inter-operation protocol phases and involved IPC data exchanges.

The *mint* inter-operation protocol achieves dataset rendering in four phases, distributing the generation and integration of a visualization across the two involved processes:

1. Alignment of dataset and impostor (by both processes)
2. Sharing of stereo camera parameters (by VR process)
3. Dataset rendering (by visualization process)
4. Rendering of impostor (by VR process)

We outline the important details and semantics of the four phases in the following in terms of OpenGL and Unity terminology.

In *phase 1*, a common coordinate system for visualization rendering is established between the VR and visualization processes using the bounding box extents of the dataset. The goal is to provide the user with a dataset representation in VR that they can interact with, and that also acts as a projection surface for the visualization result in VR. The impostor is set up such that its Unity rendering mesh and physics collider mirror the axis-aligned bounding box of the dataset in *model space*. To ensure easy interaction, we rescale the impostor object to fit within a 1 m^3 volume in the VR scene. This scale presents a sufficient overview of the data and is a convenient starting point for interaction.

[6] https://megamol.org.

[7] https://inviwo.org.

[8] https://www.paraview.org.

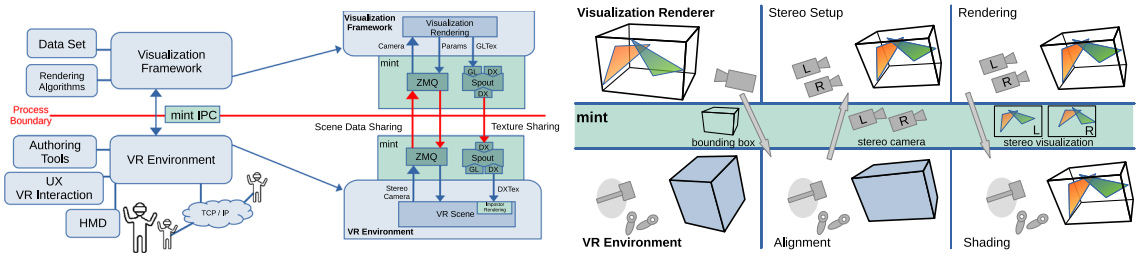
```
auto bbox = mint::receive<BoundingBox>();
scene.impostor.align(bbox);

while(true) {
 auto cam = scene.VR.stereo_camera();
 mint::send<StereoCamera>(cam);



 auto vis_image = mint::receive<Image>();
 scene.impostor.set_texture(vis_image);
 render(scene); // potentially another thread
}
```

Listing 1: *mint* steering loop

```
auto bbox = data_set.bounding_box();
mint::send<BoundingBox>(bbox);


while(true) {


 auto cam = mint::receive<StereoCamera>();

 auto left = render(data_set, cam.left);
 auto right = render(data_set, cam.right);
 mint::send<Image>(left, right);



}
```

Listing 2: *mint* rendering loop

Fig. 2 *mint* steering and rendering process exchanging data over *mint* IPC functions



(a) *mint* system architecture

(b) *mint* rendering data exchange protocol

Fig. 3 **a** The architecture of our proposed system. The visualization framework (top) and VR environment (bottom) each run as a separate process, asynchronously exchanging rendering data via the *mint* inter-process communication API. **Left**: High-level overview of system components. **Right**: Detailed architecture of the *mint* library and exchanged rendering information. Communication between the processes is abstracted by *mint*, which internally uses the ZeroMQ (ZMQ) and Spout libraries. **b** Data exchange protocol between the visualization and VR processes via *mint* and the resulting system state changes

In *phase 2*, the VR process broadcasts the camera pose for the left and right eye and camera projection parameters specific to the HMD in use to the visualization process. To allow generic integration into visualization frameworks, we reduce the potentially complex scene graph in Unity by re-framing the stereo camera pose *relative to the dataset impostor*, i.e., the camera pose is expressed in the *model space* of the dataset/impostor. This way, we avoid using model matrices, as scientific visualization frameworks may expect all positional data to be in *world space*. However, one could easily also provide a model matrix for the impostor along with world space camera parameters for visualization frameworks supporting this. VR rendering in Unity uses asymmetric view frustums to render for each eye. For generic integration into visualization frameworks, we provide camera parameters with a symmetric view frustum containing the original asymmetric frustum, taking this into account in further processing of the visualization rendering results.

In *phase 3*, the visualization process renders the visualization according to the stereoscopic camera parameters and provides the visualization rendering to the VR process. The visualization process is expected to provide two OpenGL color textures containing the stereo rendering results, which are sent to the VR process using the *mint* texture-sharing channel. To accommodate different rendering architectures of visualization frameworks, the result textures can be provided individually using two separate function calls to *mint* or jointly via a single function call. The latter method implements a single-pass stereo ("double-wide") texture-sharing mode, ensuring a synchronized transmission of visualization results for both eyes at once. Visualization parameters can be sent from the VR process to the visualization process via data channels, and the visualization process might also send back parameter state changes, e.g., to update user interface elements in the VR scene (see Sect. 5).

In *phase 4*, the VR process integrates the visualization result seamlessly into the VR environment with the help of the impostor object. The underlying idea is that the impostor screen-space footprint acts as a stencil to cut out the rendered visualization from the shared texture, while the impostor geometry embeds the visualization in the 3D space. Further, the 3D impostor object also fulfills the role of a physical representation of the scientific dataset that can be interacted with in the VR scene. We project the visualization result into the VR scene through a custom shader written in HLSL/Cg that we attach to the impostor as its object material. As such, the impostor is rendered by the Unity rendering pipeline in the same manner as all other game objects in the scene. The actual shading of the impostor is a simple RGBA-color texture access according to interpolated screen space coordinates of impostor fragments.

Since the shader only writes fragment colors, the original depth of the impostor geometry stays untouched, ensuring correct occlusion between the impostor and other objects in the VR environment. As a result, the impostor geometry embeds the dataset in the 3D scene and acts as a projection surface for the visualization rendering. When the camera position is inside the dataset, we disable backface culling for the impostor object, such that the backfaces naturally provide a projection surface for the visualization. Note that this way of texturing the impostor effectively and intentionally cuts out all other contents of the rendered visualization that lie *outside* the impostor silhouette. Meanwhile, alpha blending ensures that transparent parts of the visualization *inside* the impostor silhouette seamlessly blend into the VR scene. This approach is only an approximation for correct blending, i.e., virtual objects inside or behind the impostor in the VR scene are not considered by the visualization but simply overlaid with the visualization result. For cases where correct blending is required, one may include a feedback loop informing the visualization process of geometry inside the dataset. This information could then be used, e.g., for early ray termination in case of volume rendering.

4.3 Inter-process data and texture-sharing

The *mint* library implementation provides data structures, code routines, and conventions regarding how data is exchanged between a VR *steering* process and a visualization *rendering* process. The central design goal is to provide each process with data it can immediately use, with *mint* normalizing data semantics and performing conversions, e.g., camera parameters stemming from Unity (left-handed coordinate system) are converted to be usable for construction of camera matrices in OpenGL (right-handed coordinate system). Currently, *mint* provides implementations of the same, compatible data structures and data exchange routines in C# (used by Unity steering) and C++ (used by visualization rendering). We refer to Listing 3 in Appendix A.2 for C++ pseudo-code of the core *mint* data structures and IPC methods.

mint uses topic-based channels for data exchange, i.e., either process can broadcast and receive data identified by a topic name. For the data backend, we use the *ZeroMQ* library,[9] utilizing the *PUB/SUB* sender/receiver pattern. Data topics can be created and received on the fly by providing a topic name both sender and receiver agree upon. For the core data structures, *mint* provides automated serialization to and from JSON.[10] The actual semantics and structure of the exchanged data (data type) are either implied by the topic name (e.g., *CameraProjection*) or by convention as specified by the programmer of a feature. This approach fits our goal of facilitating easy and rapid prototyping. At the same time, we avoid breaking visualization frameworks that may not support all features of a given VR environment, e.g., a visualization process might not implement transfer function editing via *mint* but might still implement dataset rendering and other VR interactions.

Efficiently sharing textures between processes is critical to maintaining performance suitable for a VR experience. *mint* uses the C++ software library *Spout2*[11] (and the *Klak Spout Unity plugin*) to share OpenGL textures. Spout internally relies on DirectX features (Microsoft 2021; Gold and Subtil 2010) to exchange texture contents across processes while leaving the texture in GPU memory (a zero-copy mechanism). Due to the dependency on DirectX, Spout only supports machines running Microsoft Windows.

5 Collaborative virtual environment: VRAUKE

On top of the technical implementation of *mint*, we created the virtual environment *VRAUKE* using the Unity engine, which acts as a proof-of-concept for the *collaboration* use case outlined in Sect. 3. The goals

[9] https://zeromq.org.

[10] https://github.com/nlohmann/json.

[11] https://leadedge.github.io.

for this implementation were (1) testing and evaluating the technical feasibility of *mint*, and (2) exploring interaction paradigms with visualized datasets in VR. Based on the personas described in Sect. 3, the following requirements were identified:

- **R4** Provide different interactive visualizations to expert users in a unified VR environment with sufficient performance.
- **R5** Allow users to manipulate the visualization using typical tools from the visualization domain, like transfer functions or transformations.
- **R6** Have multiple users remotely act on a single visualization with a network-synchronized state to enable a collaborative experience.

5.1 Visual design

As the virtual setting for analyzing scientific datasets, we chose a circular laboratory scene inspired by modern anatomical theaters (**R4**). Figure 4 depicts the scene and different interaction scenarios with a visualization rendered by Inviwo. The laboratory contains the visualized dataset in the center, as well as interaction tools and avatars of other collaborators connected to the multi-user session around it. This room style and layout provide users with a familiar and minimalistic environment that does not need much exploration in itself, thus directing attention to the visualized dataset. Second, it also allows us to set up the scene lighting in a uniform, diffuse, and indirect way that roughly corresponds to the visualization result (which is itself not affected by Unity lighting). This prevents the visualization from seeming "out of place" because of incorrect or non-matching lighting. Third, the room layout also helps with collaborative interaction, as it provides a maximized overview of what others are doing. This is important because, as all collaborators share a common state, a notion of "ownership" of tools and the visualization itself (e.g., while zooming or rotating) is required and has to be communicated visually as well.

5.2 Interaction design and collaboration

5.2.1 Principles

For easy accessibility and intuitive user experience, we designed *VRAUKE* with a "grab and use" interaction paradigm in mind, following already established patterns from other VR applications and games. We encourage the user to explore the data by direct manipulation using the VR controllers or hand tracking if supported by the HMD. Interaction tools represented as virtual-physical objects (e.g., an actual cutting plane) in the lab scene provide access to specialized visualization features representing the capabilities of

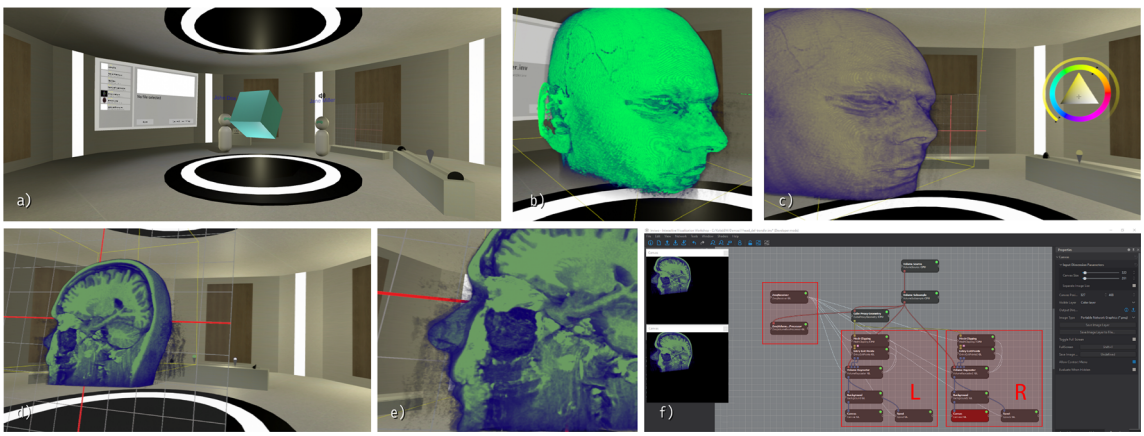


Fig. 4 *VRAUKE* laboratory example. **a** Laboratory overview with (left to right) file loader panel, participant avatar 1, animation player in the background, mint dataset, participant avatar 2 (currently talking), cutting plane, and transfer function editor. **b** Dataset loaded, scaled up, and rotated slightly. **c** Transfer function modification via UI visible in the background. **d** Cutting plane applied to obtain a head cross-section. **e** Close-up of dataset rendering received from Inviwo. **f** Inviwo desktop application rendering stereo frames; additional *mint* processors and stereo subgraphs are highlighted. This figure shows an example of an MRI head scan distributed as an Inviwo example workspace (dataset size 13 MB). Dataset courtesy of Stefan Röttger

individual visualization backends and datasets. This way, we mostly avoid using desktop-like 2D menus in the VR environment while allowing for immersive, hands-on exploration (**R5**).

5.2.2 Interactions

As explained in Sect. 4, the setup and rendering of the *Unity impostor* ensure that pose and scale manipulations of the impostor game object translate to appropriate visualization rendering. Thus, in *VRAUKE*, established VR interaction paradigms to grab, move, rotate, and scale the dataset with controllers, as well as locomotion, work without special support by the visualization process.

5.2.3 Tools

Advanced user interaction features are implemented in the form of *interaction tools* that are backed by capabilities of individual visualization applications for their respective datasets (**R5**). Interaction tools are objects placed in the *VRAUKE* laboratory and follow our hands-on interactivity goal. Following the grab-and-use paradigm, they provide strong affordances (Norman 2013) on how they can be used. The currently implemented tools are a cutting plane, a transfer function editor, and an animation player (Fig. 4a). The interaction tools transmit their internal state via individual *mint* data topics and may depend on information provided by the visualization application. If the currently active visualization process supports the interaction tool, the visualization of the dataset will be adjusted accordingly. This loose coupling of interaction tool and visualization features ensures that *VRAUKE* can be easily used with different visualization backends. Visualization frameworks are free to support multiple, all, or none of the available interaction tools.

5.2.4 Collaboration

As outlined in our use cases, a major goal was to facilitate collaborative visualization by utilizing features of the VR framework (**R6**). In *VRAUKE*, we achieve remote and local collaboration using multiplayer session management features of Unity, made available through the Photon Unity Networking plugin (PUN). The multiplayer integration for *VRAUKE* consists of adding game object state synchronization via the Photon API to the impostor, the loader panel, and the interaction tools. Following an "ownership" principle, collaborative objects in *VRAUKE* have only one user manipulating its state at a time to avoid conflicts. However, it is possible for different users to use different tools simultaneously, e.g., one user could change the transfer function properties while another could work with the cutting plane tool. Depending on the actual use case, constraints on this behavior could be implemented in Unity, i.e., actively used tools could automatically exclude the use of others, if necessary, for usability reasons or because of limitations of the employed visualization rendering backend. Users participating in a collaborative session are represented as minimalistic avatars wearing HMDs in the scene. Further, users are labeled by their name and are able to communicate through voice chat following a push-to-talk scheme using a VR controller button.

5.2.5 File handling

The project loader panel user interface on the wall provides an overview of visualization project files that are available for exploration in the VR session. It also allows switching between different visualizations without leaving VR. Loading a project file starts the visualization application associated with the project file (identified by file extension) as a new process in the background, transparent to the user. As soon as the visualization framework begins rendering, the dataset impostor shows the rendered dataset visualization. Visualization projects suitable for *VRAUKE* need to be prepared beforehand for correct VR rendering. This preparation mainly involves enabling stereo rendering in the target visualization framework and connecting VR interaction tools to appropriate visualization features/parameters. As all rendering is done locally on each user's machine, the collaborative scenario requires the appropriate visualization frameworks and visualization project files to be shared and installed beforehand. However, it would be trivial to support such data sharing within the Unity application as well, e.g., through a cloud storage provider.

5.3 Use of the system

The *mint* and *VRAUKE* systems have three groups of target users at different levels of expertise and abstraction: (1) end users inside a *VRAUKE* VR session, e.g., domain experts or the general public, (2) VR environment developers and designers, e.g., artists or UX designers using Unity, and (3) visualization framework developers and visualization researchers.

Users of *Group 1* use *VRAUKE*, or a similar *mint* VR steering integration, for data exploration or analysis through the interaction tools and paradigms provided by the VR environment. This does not require knowledge or expertise about VR or visualization software. Instead, they may be informed by users of Group 2 or 3 about the use of the VR environment, the dataset they see in VR, and available modes of interaction. *Group 2* users design and implement VR settings according to use cases and specific needs of target users (Group 1). They use an existing *mint* integration to add visualizations to the VR environment and, e.g., connect visualization capabilities to interaction tools or other elements in the scene. These users do not necessarily need to know in detail how *mint* is integrated into the visualization software. However, depending on the use case and how visualization capabilities are exposed by the respective software, they may modify the *mint* integration with the visualization framework or coordinate with Group 3 users to expose additional visualization features for use in *VRAUKE*. *Group 3* users integrate the *mint* inter-operation rendering protocol into their visualization software using the *mint* API. They have the necessary background to decide how *mint* can be best integrated into the visualization framework's architecture and source code, e.g., how to set up stereo rendering in a framework or how the data flow for *mint* data channels exposing visualization properties should be routed.

While the main concerns and responsibilities of the user groups appear to be clear, they may also become blurred as a result of close collaboration. Users may request features they require at their own level of system use from others, recognize relevant features they could provide to others, or at some point become familiar enough with the system to implement features themselves, e.g., when domain experts become invested in an immersive analytics tool and want to modify the VR environment to better suit their needs.

6 Evaluation

We evaluate our system in three ways. First, we report measurements for the inter-process frame rendering round-trip time, i.e., the frame lag introduced by the *mint* protocol, from broadcasting camera parameters in the steering process to receiving the corresponding visualization results in the impostor shader. Second, we cover the integration of the *mint* protocol into the visualization frameworks Inviwo, MegaMol, and ParaView, implementing message exchange and stereo rendering for VR integration. Third, we report the results of an informal evaluation with two domain experts inspecting datasets from their domain in *VRAUKE*. We did not conduct a controlled user study of *VRAUKE* as we focused on assessing the general need for such a system, its technical feasibility, and the development of prototypical use cases.

6.1 Frame lag in inter-process rendering

This test was conducted using a minimal rendering application, *mint*-rendering, which integrates the *mint* C++ rendering API and renders a colored quad (two triangles). We chose a minimal application with negligible overhead to be able to explicitly control the rendering process performance, as well as data logging and benchmark setup. Additional frame lag measurements of VR sessions with real-world datasets rendered by MegaMol are available in Fig. 7 (Appendix A.1). However, analyzing frame lag using a fully featured visualization framework influences measurements depending on the dataset, the configured visualization pipeline, and the exact camera animation path (Bruder et al. 2020). To isolate the effects introduced by the *mint* rendering protocol, in the following, we analyze the minimal controlled setup outlined above.

The steering process in these measurements was the *VRAUKE* application implemented in Unity (version 2019.1). In terms of GPU workload, the *VRAUKE* scene and steering overhead can be regarded as lightweight, as it consists of few simple assets with textured triangles, i.e., no heavy rendering workloads for features like level-of-detail for high-poly meshes, scene culling, or real-time global illumination are present. For the benchmark, we used native Unity tooling to animate a circular camera path around the dataset (lasting 15 seconds), followed by a rotation, up-scaling, and down-scaling of the dataset (also lasting

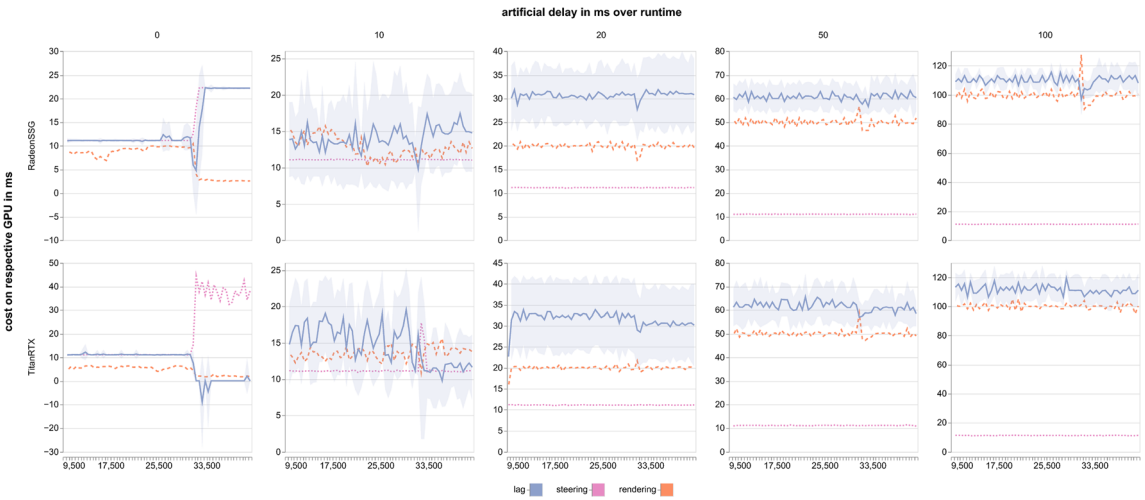


Fig. 5 Frame lag and rendering performance for *mint*-rendering (*rendering process*, dotted orange line) and *VRAUKE* (*steering process*, dotted purple line) over 30 seconds. Mean and standard deviation (blue line and error band) of frame lag is measured for artificial rendering delays of 0 ms/f, 10 ms/f, 20 ms/f, 50 ms/f, and 100 ms/f. All measurements are in milliseconds per frame, lower is better. Note that each plot has an individual y-axis scaling. **Top:** Results on AMD Radeon SSG, Vega GPU architecture (AMD Ryzen 2700X 8c/16t CPU, 32 GB DDR4 RAM, Windows 10). **Bottom:** Results on NVIDIA Titan RTX, Turing GPU architecture (Intel Core i7 9700K 8c/8t CPU, 64 GB DDR4 RAM, Windows 10). For more details, please zoom in on the digital version

15 seconds). On startup, the *mint*-rendering process was configured with a specific target rendering performance between 0 ms/f (unbounded FPS) and 100 ms/f (10 FPS). Frame lag measurements started after a 10-second warm-up period.

To track frame lag in the system, we attach the current Unity frame ID to the *mint StereoCameraView* update message. The *mint*-rendering application embeds the received frame ID into a pixel of the shared texture that the Unity impostor shader extracts and writes into a GPU buffer. We track the frame IDs during the VR session and download them from GPU memory in one batch. This avoids impacting rendering performance during measurement due to repeated GPU–CPU data transfers.

For rendering in the *VRAUKE* steering process, we define *discrete frame lag (integral frames)* and *frame lag (time)* in terms of the frame IDs where rendering of a camera configuration has been requested via *mint*, and presented on the screen during impostor shading:

$$discreteFrameLag = frameID_{presented} - frameID_{requested}$$
$$frameLag = frameEndTime_{presented} - frameStartTime_{requested}.$$

We measure the frame start and end time once as wall-clock time in milliseconds at the start of a frame, such that for a frame with ID n holds $frameEndTime_n = frameStartTime_{n+1}$.

Figure 5 documents the frame lag time (mean and standard deviation) as plots over a 30-second period for different *mint*-rendering artificial rendering delays and two different graphics cards. The plots average 200 ms of measurements for mean and deviation, i.e., the 30-second measurement period is binned into 150 bins. The average runtime costs of the *VRAUKE* steering and *mint*-rendering processes are shown as orange and purple dotted lines, respectively. *VRAUKE* rendering costs are measured to be consistently at 11 ms/f, reaching the commonly used 90 FPS target for VR applications (Wang et al. 2023). This is expected because Unity, or rather the SteamVR runtime managing the VR hardware and rendering, is designed to keep a steady frame rate to ensure a good user experience. The *mint*-rendering application achieves the targeted average rendering costs for the 20 ms/f, 50 ms/f, 100 ms/f rendering delays on both GPUs. For the 0 ms/f artificial delay, *mint*-rendering averaged rendering costs between 5 ms/f and 8 ms/f are measured for the Radeon SSG and Titan RTX GPUs, respectively. For the 10 ms/f artificial delay, the rendering costs vary between 11 ms/f and 15 ms/f, overshooting the targeted costs. This may be due to our simplistic target delay implementation or due to workload scheduling effects in the graphics driver, i.e., the GPU driver is unaware of our rendering cost intentions.

The mean frame lag time and standard deviation (blue line and error band) mostly scale with the corresponding *mint*-rendering frame cost. The frame lag plots show lots of jitter and cover a wide range of possible values, e.g., on the Radeon SSG with a 10 ms/f artificial delay (RadeonSSG/10 ms/f), frame lag has values in the range 7 ms/f up to 24 ms/f. However, the absolute frame lag time values need to be interpreted in relation to the rendering performance of the steering process, i.e., discrete frame lag. Since *VRAUKE* renders at 11 milliseconds per frame, frame lag time will be measured as multiples of 11 ms/f. We use frame lag time to establish a common scale for all measurements shown in the figure. In the case of RadeonSSG/10 ms/f, with 11 ms/f rendering costs for *VRAUKE*, this means the discrete frame lag is between zero, one, or two frames: the rendering response either came back within the same frame it was requested (lag time $\leq$ 11 ms/f), the next frame (11 ms/f $\leq$ lag time $<$ 22 ms/f), or two frames later (22 ms/f $\leq$ lag time $<$ 33 ms/f). When looking at the discrete frame lag, the rendering response is presented predominantly within one frame in the steering process: e.g., for RadeonSSG/10 ms/f discrete frame lag is on average 1, for RadeonSSG/20 ms/f discrete frame lag is 2.5, for RadeonSSG/50 ms/f discrete frame lag is 5.5, and for RadeonSSG/100 ms/f discrete frame lag is 10.5.

Thus, on average, the *mint* inter-operation protocol delivers frames within expected bounds. However, as the two processes run simultaneously on one machine, their individual rendering performance is also influenced by other factors that the *mint* implementation cannot control, like process scheduling by the operating system, CPU utilization, GPU driver intrinsics, and GPU workload.

We interpret the measurements for RadeonSSG/0 ms/f and TitanRTX/0 ms/f as examples of such systemic factors: In the 0 ms/f artificial delay scenarios, for the first twenty seconds, the *mint*-rendering process runs faster than the steering process, delivering rendering results within the same steering frame it was requested, i.e., the frame lag time and steering lines are identical, with little jitter. During measurements, on startup, we manually clicked into the steering process window to make it the focused window of the desktop session, i.e., a foreground process, making *mint*-rendering a background process. At second 20 into the benchmark, we manually clicked into the *mint*-rendering window to switch process priorities, i.e., making rendering the foreground process. This priority switch is visible as a spike in the plots for all measurements, leading to a slightly lower average frame lag time for most scenarios on the Titan RTX GPU. For the 0 ms/f artificial delay scenarios however, focusing the rendering process improves *mint*-rendering performance but drastically worsens steering process performance (purple line shoots up), i.e., the cost to render the *VRAUKE* scene went up 2x (from 11 ms/f to 23 ms/f) on the Radeon SSG, and 4x (from 11 ms/f to roughly 40 ms/f) on the Titan RTX. We think this is best explained by a form of resource starving: the *mint*-rendering process, being a foreground process, is likely given priority access to CPU and GPU resources by the operating system. As such, the rendering process is able to submit large amounts of work to the GPU, occupying GPU and other system resources to such a degree that the background steering process is starved. This explanation is also supported by the observation that for all other artificial delay scenarios, the process priority switch does not influence *VRAUKE* performance.

Due to our definition of frame lag, a lag less than the rendering cost of a steering frame should not be possible, i.e., a rendering result cannot be presented before the steering requested it. The RadeonSSG/0 ms/f scenario adheres to this assumption. In the TitanRTX/0 ms/f scenario however, the priority switch leads to a frame lag time of 0 ms/f and even negative lag up to -10 ms/f. This can be explained by the Unity rendering architecture and GPU driver intrinsics. The Unity engine employs a multi-threaded rendering system, where a CPU thread processes user inputs, game logic, and render data preparation. A dedicated GPU thread receives these prepared GPU workloads and submits them to the GPU, e.g., via OpenGL commands. This enables the CPU thread to start processing data for the next frame while GPU rendering of the current frame is happening. A frame lag of 0 ms/f occurs when a frame with frame ID n has been processed by the CPU thread and passed for rendering to the GPU thread, subsequently starting CPU processing of frame $n+1$. If *mint*-rendering receives the camera parameterization for frame $n+1$ and provides the resulting image before Unity GPU processing of frame n is finished, the shared texture contains frame ID $n+1$, while being processed on the GPU for impostor shading of frame n. During data analysis, this leads to a frame ID difference calculation of

$$\begin{aligned} discreteFrameLag &= frameID_{presented} - frameID_{requested} \\ &= n - (n+1) = -1 \end{aligned}$$

producing a cancellation of timestamps upon frame lag time calculation

$$\begin{aligned} frameLag &= frameEndTime_{presented} - frameStartTime_{requested} \\ &= frameEndTime_{n} - frameStartTime_{n+1} \\ &= frameEndTime_{n} - frameEndTime_{n} \\ &= 0. \end{aligned}$$

A negative frame lag time can be explained by the Unity multi-threaded rendering architecture in interaction with GPU driver intrinsics and the *VRAUKE* steering process being a background process with high rendering cost. GPU drivers may asynchronously queue rendering requests instead of immediately submitting requested OpenGL work to the GPU. This lets an OpenGL application perform iterations of the main render loop without any rendering actually being done on the GPU, until the GPU driver emits a larger GPU work package at a later point in time.[12] We hypothesize that due to the GPU driver asynchronously queuing GPU work, the Unity CPU thread is able to perform processing of multiple frames and send corresponding camera requests via *mint*, while the Unity GPU thread is starved by *mint*-rendering for GPU resources and fails to progress rendering state. Meanwhile, *mint*-rendering, a foreground process with little rendering cost of roughly 3 ms/f, is able to receive and process the latest rendering requests from the Unity CPU thread, providing a rendered image with an embedded frame ID far ahead of the resource-starved Unity GPU thread, i.e., it appears as if *mint*-rendering rendered in the future.

6.2 Visualization renderer integration

Easy and non-intrusive integration of the *mint* library into existing visualization frameworks is a central design goal of our system. In this section, we briefly report on our integration of the *mint* visualization rendering mechanism into the Inviwo, MegaMol, and ParaView source codes, how to achieve stereo rendering, and integration of interaction tools with these visualization frameworks for use in *VRAUKE* sessions. Detailed descriptions regarding the integration are available in Appendix A.3. Source code excerpts are provided in the supplemental material (Geringer et al. 2024).

6.2.1 Inviwo

Figure 4 shows our *mint* integration into Inviwo. We implemented new Inviwo processors that expose functionality for receiving and sending data packages (*ZmqReceiver, ZmqVolumeBoxProcessor*) and sharing textures (*Spout*) (Fig. 4f). The dataset bounding box is provided by data source processors, e.g., the *Volume Source* processor, while incoming render parameters like camera and interaction tool state (cutting plane and transfer function editor) are linked to relevant processors using property links. The final visualization image is passed to the Spout processor for texture-sharing. Stereo rendering is achieved by manually setting up a processor network that renders the dataset for each eye separately. This implementation is based on an early prototype of the *mint* concept in Inviwo.

6.2.2 MegaMol

Figure 1 shows our *mint* integration into MegaMol. We integrated *mint* into MegaMol by interfacing with *View3D* visualization graph modules, passing the VR camera parameters to the *View3D* camera, and broadcasting the visualization result contained in the *View3D* framebuffer. Stereo rendering is achieved automatically by executing the existing module graph once for each eye. Interaction tools are also set up automatically for synchronization via *mint* when corresponding graph modules are recognized. This implementation did not require writing new MegaMol modules but could be fitted into the MegaMol frontend code, which prepares and executes the module graph. The benefit of this approach is that *VRAUKE* VR capabilities can be used for any MegaMol project containing a View3D without special preparation. The user only needs to activate the *mint* integration via a configuration flag on MegaMol startup.

[12] Word on the street says this mechanism sometimes may lead to unexpected and difficult-to-explain performance measurements of OpenGL visualization algorithms on some GPUs.

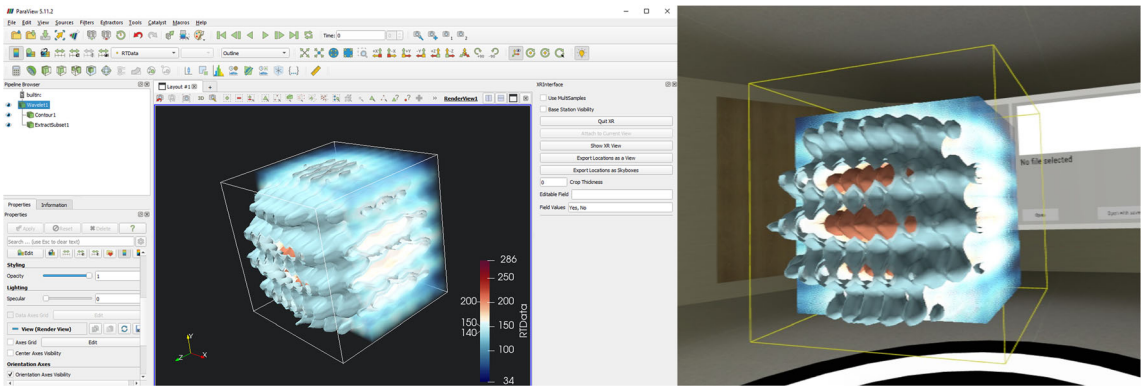


Fig. 6 The volume from the ParaView example data is rendered (left), sent via *mint*, and displayed in the *VRAUKE* application (right). For our prototype, we used ParaView's *XrInterface* plugin and modified its rendering backend, VTK. This figure shows the *Wavelet* example visualization distributed with ParaView (dataset size 270 kB). The dataset itself is dynamically generated during runtime using a periodic function

6.2.3 ParaView/VTK

Our *mint* integration into ParaView/VTK can be seen in Fig. 6. We modified the ParaView *XRInterface Plugin*, which uses VTK's OpenVR module to render stereo images into dedicated framebuffers. We inject *mint* stereo camera data into the *vtkOpenVRCamera* class and intercept the rendering result textures in *vtkOpenVRRenderWindow*. We do not need to explicitly emulate stereo rendering, as this is already handled by the plugin. In order for *VRAUKE* to function properly with ParaView, we further needed to mark ParaView as a background OpenVR process and disable automatic rendering of a ground floor drawn by ParaView into the VR framebuffers, as in our setting *VRAUKE* provides the user with a virtual environment. With only previous knowledge about ParaView from the user perspective, most of the time and effort went into understanding the visualization pipeline implementation and finding the appropriate sections to make the necessary modifications.

6.3 User tests with domain experts

We conducted two dedicated user tests with domain experts from the respective domains covered by the existing *VRAUKE* integrations: medical science and material science. Both users performed various tasks while employing a concurrent think-aloud method to gain insights after receiving a basic introduction to the use of the system. After the VR session, an unstructured closing interview was conducted, mainly based on the question of whether a system like *VRAUKE* might provide additional value when viewing datasets. The *VRAUKE* collaborative features were not subject to testing.

User A, a clinical neurologist with limited VR experience, was shown a dataset of a magnetic resonance imaging (MRI) head scan using *VRAUKE* (Fig. 4) and was asked to compare the data exploration process to their current clinical practice. In terms of user experience, the participant missed an easy approach to handling the dataset, coming from a 2D screen display that only provides cut perspectives from top, side, and front, and reported that a limit in degrees of freedom, as well as predefined views and snapping to axes, would be useful for the dataset and the cutting plane tool. Additionally, the user criticized the need to be very close to the dataset to interact with it and suggested additional controls for scaling and rotating the dataset while standing further away. Still, the user reported that for specific tasks, e.g., from the area of neurosurgery but also for the education of medical students, they believe that total freedom of movement and the possibility to "dive into" the dataset could be beneficial.

During the test, the rendering quality was impaired due to the specifications of the on-site test system. Therefore, the corresponding frame rates were lower than the desirable limits for VR. User A still reported a sufficient visual quality for clinical use while highlighting the benefits of 3D rendering. The user did not exhibit or report any effects of motion sickness. Using the system was described as "fun" and "inviting to explore." The user also highlighted that using the system in clinical practice might help improve privacy regarding patient data, which would otherwise be visible on-screen to others.

User B, a material scientist, reported a passing experience with immersive virtual environments from short CAVE demonstrations. The user noted that they felt grounded and immersed in the *VRAUKE* laboratory setting. They inspected a porous media dataset (Fig. 1) stemming from a simulation in their domain. The user characterized the immersive interaction with the data as "cool." During data exploration, they compared the setting and interaction methods to their daily data analysis workflow using a desktop-based analysis and visualization software established in their field (Stukowski 2009). Utilizing the cutting plane, the user was interested in localizing cracking patterns emerging from force exerted on the porous medium. Using controllers, they stated that localizing and inspecting clusters of stressed or cracked material felt more natural than with the desktop solution, where they slice the dataset by entering numerical offset values by hand. However, they also emphasized their need to reliably reproduce analysis and visualization settings related to interesting findings, for which they usually note down the involved numerical parameters. Thus, they suggested adding saving and loading mechanisms of visualization session state to *VRAUKE*. During the session, occasionally, the visualization rendering lagged noticeably. The user commented that despite the visualization lag, they still felt immersed in the scene due to the surrounding laboratory setting.

User B loaded and inspected other visualization projects available in the *VRAUKE* project loader panel by themselves and unprompted. They experimented with the interaction tools in the scene and how they interact with the current dataset. While not considering daily use, the user could imagine presenting research findings collaboratively to colleagues and students in VR or inspecting the data in VR and exporting interesting findings as images for further use.

Finally, in the context of public demonstrations, we presented *VRAUKE* with different visualization demos to school students, journalists, colleagues, industry representatives, and research collaborators. Depending on the degree of familiarity with VR environments, users were quickly able to interact with the dataset and apply the interaction tools, e.g., move the cutting plane into the dataset.

7 Discussion

Our informal tests with domain experts and public demonstrations at different venues indicate that *VRAUKE* is usable for the general public and domain scientists, as stated in our goals. The laboratory environment sparks curiosity, and people explore the datasets themselves as well as the environment and the interaction affordances. While the overall usability is acceptable for domain experts, the current *VRAUKE* interaction tools and features are too generic, and the experts commented that domain-specific abstractions and especially manipulations are still required. This also aligns well with our approach: user interactions can be designed by HCI experts in the *VRAUKE* environment, passing only parameter values to visualization tools, which can adjust rendering accordingly or be extended separately, if required.

We observe that most of the requested features would only require adjustments and additional features on the VR side, i.e., the user interface, and would not change the visualization frameworks at all. For example, saving and loading visualization session state to and from disk are supported by Inviwo, MegaMol, and ParaView. Connecting this feature to VR only requires implementing the respective tool, a corresponding *mint* message, and code that invokes the respective functionality in each of the backends. This finding, in a way, replicates user reactions to the interactions available in the OpenVR integration of ParaView.[13] Due to *VRAUKE* being a very generic proof of concept, it should serve as a starting point for collecting more (and more specific) interaction requirements and, as a consequence, extend and refine the available tools in VR. Some of these requirements can be as trivial as using proper metaphors and terminology (e.g., using terms like "tissue density" instead of "transfer function"), which are irrelevant for the employed visualization method but essential for domain scientists to lower the barrier to entry as much as possible.

Thus, depending on the actual tasks, goals, and proficiency of a user in a VR session, interaction tools need to be deliberately designed to suit the application use case and workflow. For example, simply translating all possible 2D UI elements from a visualization framework to VR interaction tools would require time and effort and may not help the user achieve their goals if the interaction tools overwhelm them, e.g., by overcrowding the VR scene or not helping with specific tasks. A semi-automated translation from visualization 2D UIs to VR user interfaces may be an option to minimize manual labor. Current solutions to integrate applications, e.g., web browsers, into VR environments use image-based embedding of

[13] https://www.kitware.com/navigation-basics-in-virtual-reality-with-paraview/.

the respective graphical user interfaces (GUIs) and forward user interactions (point and click) back to the original process. With an existing *mint* integration for a visualization framework, sending the GUI images to VR would also be easy to achieve. However, we place this concern outside of *mint's* core responsibilities.

Both the loose coupling in our design and the extensible interface to visualization renderers facilitate extending the currently available features. *mint* data communication channels exchange structured data (JSON) but do not enforce strict semantics and thus are more robust against breaking changes and added features, as mismatched data topics or parameters can simply be ignored and can lower the bar for rapid prototyping. On the downside, these open semantics have to be parsed into the specific parameters and implementations of the respective visualization frameworks, i.e., they require integration into the source code.

Frame lag and performance

The measured frame lag reported in Sect. 6.1 is within the expectations of roughly one to three frames,[14] given asynchronous and independent frame rendering for visualization and VR. In practice, there are additional factors that are hard to control. The operating system or graphics driver can prioritize applications differently, depending on whether they are focused, as seen in Fig. 5 for artificial rendering delay 0 ms/f, where a focus switch changes the performance of rendering and steering. Real-world use of visualization software can lead to similar situations, e.g., particle rendering with MegaMol can reach hundreds of FPS, given a small dataset and powerful GPU.

Our evaluation also hints at a practical solution: running the VR process as the focused process avoids starving it (first twenty seconds of all measurements), with the tradeoff of slowing down the rendering process slightly, i.e., accepting higher frame lag. However, even with VR as the focused process, for high-performance visualization rendering, *mint* provides results mostly within the same frame to VR, i.e., "instantly." On paper, the resulting system behavior is equivalent to tightly integrated solutions, i.e., where visualizations are embedded within the VR rendering loop. When visualization rendering is slower than VR rendering, higher frame lag occurs. Our user tests suggest that this is tolerated by users to some degree, where in the presence of frame lag, an expert explicitly mentioned that the *VRAUKE* environment remaining interactive "provides an immersive feeling." However, the limits of tolerable frame lag are unclear and probably depend on the individual user. We plan to study this effect in more detail in the future.

With tightly integrated solutions, however, slow visualization rendering also slows down rendering of the VR environment itself, which is known to break immersion and cause disorientation or sickness (Stauffert et al. 2020; Liu and Heer 2014). Thus, both for fast and slow visualization rendering, the design of *mint* offers benefits over a tight integration. Distributing the rendering load of the VR and visualization processes across additional dedicated GPUs could also avoid resource starving while maximizing performance for each process, but this would need explicit support from the involved graphics APIs and the operating system, and would incur transmission costs over the PCIe bus to integrate renderings. Using Unity and SteamVR seems to always imply late frame reprojection, a motion-to-photon latency optimization technique commonly used by VR backends (Waveren 2016). Such reprojection is detrimental to perceived visualization lag in *mint*, as the VR backend is unaware of our image-based rendering approach and introduces a slight but noticeable distortion in VR.

Several optimizations can improve the frame lag and user experience. There are existing methods for predicting camera positions based on HMD motion (Saad et al. 1999) or reprojection of rendered frames with a depth buffer (Zellmann et al. 2012). We have prototyped both, but there are several caveats to the latter: the current approach using Spout as a texture-sharing backend in *mint* does not support depth or more than 8 color bits per channel, and bits packing behaves differently across OpenGL and Direct3D, making this solution unstable and error-prone. Additionally, modern OpenGL renderers can override (Gilg et al. 2021) or neglect depth buffer semantics arbitrarily, such that transported depth cannot be generally relied upon for reprojection. Explicitly stalling the visualization renderer until the next steering frame request comes in from *mint* could also be beneficial for very low visualization frame rates, as it avoids waiting for the slower visualization renderer in the steering process in case a (probably outdated) visualization frame is still being computed. Such explicit synchronization of the inter-operation between the steering and rendering processes requires further investigation.

[14] For details, we refer to the supplemental material (Geringer et al. 2024).

API and integration

Our evaluation shows that integration of *mint* into existing visualization software has low complexity but still depends on knowledge of the respective software architecture and rendering pipeline design. Integration of the *mint* API worked easily with MegaMol, as it fits the design of the respective rendering pipeline very well. For Inviwo, we used an older, less abstract *mint* API, but integration was also fast. The integration prototype of *mint* inside ParaView/VTK has different limitations. Here, the main hurdle for integration was missing familiarity with the ParaView code base, and most of the time was spent reading code and documentation to find the correct spot for the integration.

8 Conclusion

In this paper, we presented *mint*, a generic framework to connect the domains of scientific visualization and virtual immersive environments. We identified several scenarios in which such a connection could be fruitful, but we also noticed that existing approaches to bring the two domains together make significant compromises in at least one of them. We designed and implemented the *mint* inter-process rendering protocol as a C++ library that is open source and provides the *mint* API as a rendering interface between visualization and VR tools. Our *mint* integrations into three existing visualization frameworks (ParaView, Inviwo, and MegaMol) and the Unity environment *VRAUKE* show that integrating *mint* into existing tools requires little implementation effort.

With the loose coupling approach central to *mint*, we contribute an easy-to-use and technically up-to-date implementation to bring scientific visualizations into VR and, at the same time, also preserve the domain-specific advantages of both visualization frameworks and VR authoring tools. Our system allows HCI researchers, artists, and UX designers to access visualization systems from within their domain-specific authoring tools and workflows. This brings the potential to more easily translate state-of-the-art visualization research to interactive and immersive experiences, making it more accessible to a broad spectrum of audiences like domain experts, educators, or the general public, tailored to their specific needs.

From a technical perspective, improvements for *mint* need to address frame lag in the system. Here, future work could apply techniques from image-based, mobile, and remote rendering research to reduce frame lag perceived by a user inside a VR session.

Appendix

A.1 Frame lag on particle datasets

Figure 7 shows plots for frame lag with real-world datasets, using MegaMol as the rendering process and *VRAUKE* as the steering process. We measured frame lag with four different particle datasets rendered by MegaMol for a VR session. The datasets are:

- *Virus*, looping animation, 498 MB file size, 214,440 particles per animation step
- *Injection*, looping animation, 1.71 GB file size, particle count increasing up to 393,672 per step during animation
- *Ceramic* (see Fig. 1), static, 146 MB file size, 2,027,467 particles
- *Laser*, static, 732 MB file size, 48,000,000 particles

Renderings of the datasets are shown in Fig. 8. MegaMol was configured to render the particle data using ray casting of spheres within the fragment shader of the OpenGL rasterization pipeline and applying object-space ambient occlusion for better visibility of spatial features. Since no acceleration data structures like k-d trees or BVH are used, rendering performance is largely determined by the amount of particles the GPU has to process for rendering. The benchmark setup and shown measurements in the plots are the same as in Sect. 6.1. As such, the interpretation of the plots is analogous to our analysis in Sect. 6.1.

It is important to note that the benchmark setup analyzed in Sect. 6.1 uses a CPU-based artificial rendering delay for the *mint*-rendering process without heavy GPU workload to cause frame lag in terms of delayed rendering result texture messages.

In the visualization benchmarks presented in Fig. 7, particle rendering with MegaMol induces real GPU workloads. In turn, the case where the rendering process is the focused window (last 10 s of measurements)

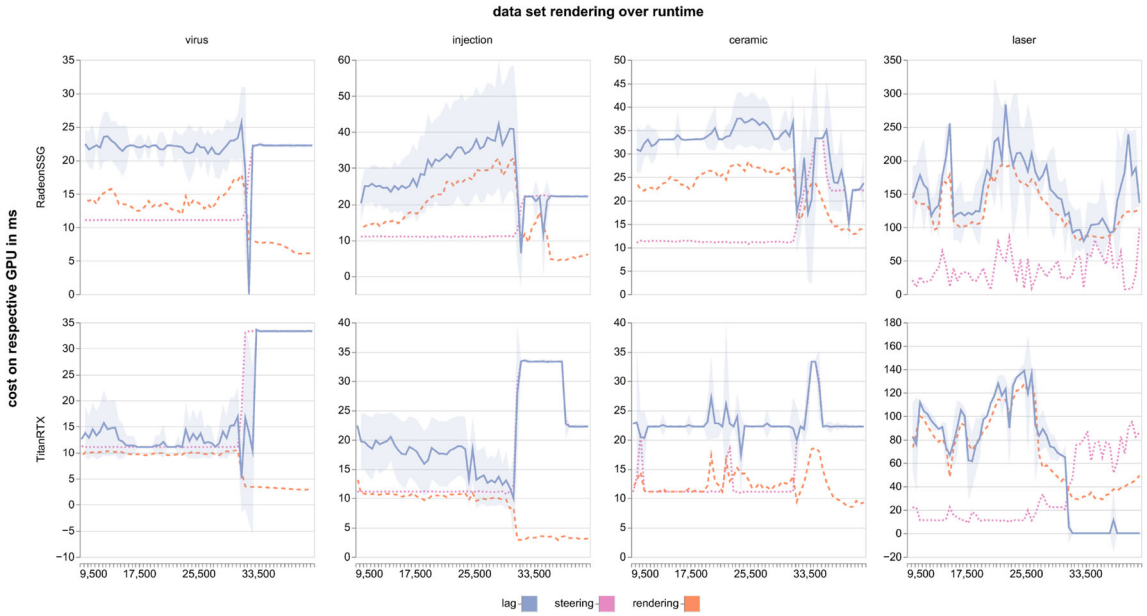


Fig. 7 Frame lag and rendering performance for particle datasets rendered by MegaMol (*rendering process*, dotted orange line) and *VRAUKE* (*steering process*, dotted purple line) over 30 seconds. The mean and standard deviation (blue line and error band) of frame lag is measured for rendering of different datasets. All measurements in milliseconds per frame, lower is better. Note that each plot has an individual y-axis scaling. **Top:** Results on AMD Radeon SSG, Vega GPU architecture (AMD Ryzen 2700X 8c/16t CPU, 32 GB DDR4 RAM, on Windows 10). **Bottom:** Results on NVIDIA Titan RTX, Turing GPU architecture (Intel Core i7 9700K 8c/8t CPU, 64 GB DDR4 RAM, on Windows 10). For more details, please zoom in on the digital version

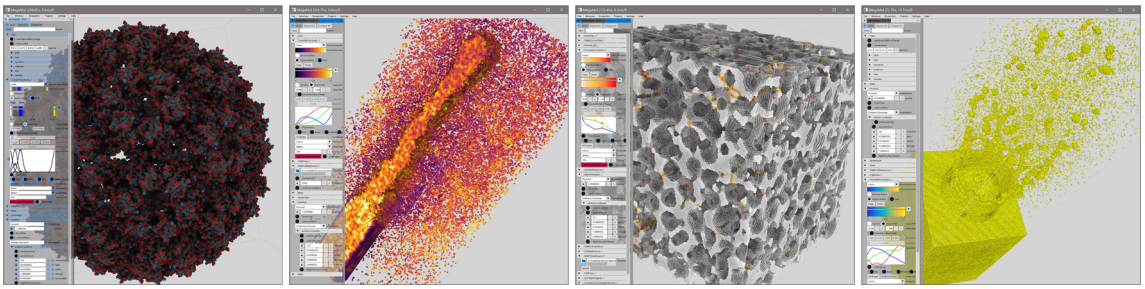


Fig. 8 The four real-world particle datasets rendered by MegaMol on the NVIDIA Titan RTX system. In these screenshots, MegaMol renders in desktop mode (VR stereo rendering off). MegaMol stereo rendering performance is shown in Fig. 7. From left to right: CCMV virus capsid rendering at 299 FPS, dataset from Speir et al. (1995), trajectory courtesy of P. Chacón; nozzle injection rendering at 331 FPS, dataset courtesy of Heinen and Vrabec (2019); ceramic fracture simulation rendering at 122 FPS, dataset courtesy of V. Guski; laser ablation rendering at 72 FPS, dataset courtesy of Sonntag et al. (2009)

induces resource starving of the VR process also for low frame rates of the rendering process, e.g., for the *Laser* dataset on the TitanRTX GPU. Thus, both for fast visualization rendering with high frame rates (*Virus* on TitanRTX) and for low visualization frame rates due to heavy GPU workload with a larger dataset (*Laser* on TitanRTX), the background VR process is starved for GPU resources due to high GPU load from the rendering process.

A.2 *mint* API

Listing 3 depicts pseudo-code of the *mint* API, i.e., the core data structures and data exchange routines of *mint*.

```
struct CameraView { vec4 eyePos, lookAtPos, camUpDir; };
struct StereoCameraView { CameraView leftEye, rightEye; };
struct CameraProjection {
  float fieldOfViewY_rad, aspectRatio, nearClip, farClip;
  uint pixelWidth, pixelHeight;
};
struct BoundingBoxCorners { vec4 min, max; };

struct DataReceiver { bool receive<Data>(Data& val, std::string const& name); };
struct DataSender { bool send<Data>(std::string const &name, Data const &val); };
struct StereoTextureSender { void send(const gltex left, const gltex right); };
```

Listing 3: mint data structures

A.3 *mint* visualization renderer integration

This section presents a detailed report on our integration of the *mint* visualization rendering mechanism into the Inviwo, MegaMol, and ParaView source codes. The source code for these integrations is available in the supplemental material (Geringer et al. 2024).

All three frameworks are written in C++, use OpenGL for GPU-accelerated rendering, and implement rendering algorithms for scientific data like volumes, molecular data, particle data, and information visualization techniques.

A.3.1 *Inviwo*

Figure 4 shows our *mint* integration into Inviwo. Inviwo implements visualizations in the form of a processor network. Hereby, visualization data is passed between interconnected *processor nodes*, each of which performs computations or data manipulations on its input data and provides output data for others to use. Evaluation of the processor network is driven by updated input data, which triggers subsequent execution of depending processors. The visualization result of a processor network is a 2D image, which is the output of a renderer processor. Image contents can be viewed using a *Canvas processor*, which takes image data as input and presents the image contents in a new window. We implemented new processors that expose functionality for receiving and sending data packages (*ZmqReceiver, ZmqVolumeBoxProcessor*) and sharing textures (*Spout*) (Fig. 4f). The dataset bounding box is provided by data source processors, e.g., the *Volume Source* processor, while incoming render parameters, like camera and interaction tool state, are linked to relevant processors (e.g., renderers) using property links. The final visualization rendering image is passed to the Spout processor for texture-sharing. Stereo rendering is achieved by setting up a processor network that renders the dataset for each eye separately. Implementing these Inviwo processors to participate in *mint* data sharing took 925 lines of C++ code. This implementation is based on an early prototype of the *mint* concept in Inviwo.

A.3.2 *MegaMol*

Figure 1 shows our *mint* integration into MegaMol. In MegaMol terminology, a visualization is defined by the module graph, which is a collection of *modules* interconnected by *calls* defining the order of rendering code execution. Module graph evaluation starts at a single *entry point* module, calling subsequent connected modules to process or render input data. For 3D datasets, the entry module is a `View3D` module. The View3D manages the 3D camera and navigation via mouse and keyboard, provides a bounding box of the 3D dataset, and holds the rendering results of the graph evaluation in a framebuffer. We integrated *mint* into MegaMol by interfacing with View3D modules, passing the VR camera parameters to the View3D camera, and broadcasting the visualization result contained in the framebuffer. Stereo rendering is achieved by executing the existing module graph once for each eye. Integrating the *mint* rendering mechanism into MegaMol took 599 lines of C++ code. This implementation did not require writing new MegaMol modules but could be fitted into the MegaMol frontend code, which prepares and executes the module graph. The benefit of this approach is that *VRAUKE* VR capabilities can be used for any MegaMol project containing a

View3D without special preparation. The user only needs to activate the *mint* integration via a configuration flag on MegaMol startup. *VRAUKE* interaction tools like the animation player and cutting plane need matching visualization modules in the visualization project to function. However, interaction tools are set up for synchronization via *mint* automatically when corresponding graph modules are recognized.

A.3.3 *ParaView/VTK*

ParaView uses the Visualization Toolkit (VTK) for rendering, and most modifications had to be applied directly to VTK, as it provides access to low-level OpenGL objects. In a first attempt, we used ParaView's built-in stereo mode and integrated *mint* into the OpenGL2 module of VTK. The idea was to use the `SplitViewportHorizontal` configuration, which displays both views side by side in the *RenderView*. In `vtkOpenGLRenderWindow::Frame()`, we split the resulting framebuffer into two separate framebuffers, then sent them via *mint's* stereo sender. Before rendering, the camera parameters are overwritten to match the configuration received from *mint*. The OpenGL2 module defines a stereo camera (`vtkCamera`) by a focal point and an offset between the eye positions while *mint* defines separate cameras per eye. Instead of converting the received parameters to a single `vtkCamera`, we decided to set the camera parameters twice, once before rendering each view, inside `vtkRenderWindow::DoStereoRender()`. However, the main problem with this first approach is that ParaView renders only on demand. Without a continuous update loop, we were unable to update to the received camera parameters and trigger ParaView's rendering again. Additionally, the default *RenderView* is designed to show a downsampled version during interactions and only render full scale on still images, which is not well suited for continuous rendering as needed for VR applications. These aspects led us to consider other approaches.

We used the *XRInterface Plugin* for our second attempt, which facilitates continuous stereo rendering. This plugin uses VTK's OpenVR module to render its views into dedicated framebuffers before submitting them to the HMD. This module's VR camera (`vtkOpenVRCamera`) conveniently has individual view and projection matrices for each eye that we could compute from the parameters provided by *mint*. After the camera is set, rendering is performed from the new point of view. In `vtkOpenVRRenderWindow::StereoRenderComplete()`, we send the rendered textures through *mint* right before they are given to the OpenVR compositor. Since both the *XRInterface Plugin* and *VRAUKE* try to output to the same HMD through OpenVR, it was necessary to change the OpenVR application type in the *XRInterface Plugin* from `VRApplication_Scene` to `VRApplication_Background`. This enabled us to run both applications simultaneously without terminating one by starting the other application. Furthermore, we had to disable automatic rendering of a floor in the ParaView VR scene since it obstructed the rendered dataset. This was achieved by setting `vtkVRRenderer::SetShowFloor()`. It took 181 lines of C++ code to integrate *mint* into the VTK OpenVR module. The working prototype can be seen in Fig. 6. Since our prototype does not consider the bounds of the rendered objects, we set the bounding box that is sent through *mint* manually. With only previous knowledge about ParaView from the user perspective, most of the time and effort went into understanding the visualization pipeline implementation and finding the appropriate sections to make the necessary modifications.

Acknowledgements The authors wish to thank Sebastian Hartwig and Dominik Engel (for Inviwo support) and Alexander Straub and Moritz Heinemann (for tireless ParaView and CMake support).

Funding This work was funded by the Ministerium für Wissenschaft, Forschung und Kunst (MWK, Ministry of Science, Research and Arts) Baden-Württemberg, Germany, as part of the project Virtuelle Kollaborations Labore Baden-Würtemberg, Phase 1-3 (KoLabBW) and the project InnovationsCampus Mobilität der Zukunft (ICM, InnovationCampus Future Mobility) and also funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB-TRR 161 – Project ID 251654672. Open Access funding enabled and organized by Projekt DEAL.

Data availability Supplemental material is available in the data repository: https://doi.org/10.18419/darus-4060

Code Availability The source code for the presented system is available in the *mint* repository: https://github.com/UniStuttgart-VISUS/MWK-mint

References

Ahrens J, Geveci B, Law C (2005) ParaView: an end-user tool for large-data visualization. In: Hansen CD, Johnson CR (eds.) Visualization Handbook, pp. 717–731. Butterworth-Heinemann, Burlington. https://doi.org/10.1016/B978-012387582-2/50038-1

Aliaga DG (1996) Visualization of complex models using dynamic texture-based simplification. IEEE Vis. https://doi.org/10.1109/VISUAL.1996.567774

Ahsan M, Marton F, Pintus R, Gobbetti E (2022) Audio-visual annotation graphs for guiding lens-based scene exploration. Comput Graph 105:131–145. https://doi.org/10.1016/j.cag.2022.05.003

Bruckner S, Gröller ME (2005) VolumeShop: an interactive system for direct volume illustration. IEEE Vis. https://doi.org/10.1109/VISUAL.2005.1532856

Boyd DRS, Gallop JR, Palmen KEV, Platon RT, Seelig CD (1999) VIVRE: User-centred visualization. In: Sloot P, Bubak M, Hoekstra A, Hertzberger B (eds.) High-Performance Computing and Networking, pp. 807–816. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0100641

Boyd DRS, Gallop JR, Palmen KEV, Platon RT, Seelig CD (1999) Using virtual environments to enhance visualization. In: Gervautz M, Schmalstieg D, Hildebrand A (eds.) Eurographics Workshop on Virtual Environments. https://doi.org/10.1007/978-3-7091-6805-9_9

Bostock M, Heer J (2009) Protovis: a graphical toolkit for visualization. IEEE Trans Vis Comput Graph 15(6):1121–1128. https://doi.org/10.1109/TVCG.2009.174

Bruder V, Müller C, Frey S, Ertl T (2020) On evaluating runtime performance of interactive visualizations. IEEE Trans Vis Comput Graph 26(9):2848–2862. https://doi.org/10.1109/TVCG.2019.2898435

Burnett MM (1999) Visual Programming. Wiley, New York. https://doi.org/10.1002/047134608X.W1707

Checa D, Bustillo A (2020) A review of immersive virtual reality serious games to enhance learning and training. Multim Tools Appl 79(9–10):5501–5527. https://doi.org/10.1007/s11042-019-08348-9

Childs H, Brugger E, Bonnell K, Meredith J, Miller M, Whitlock B, Max N (2005) A contract based system for large data visualization. IEEE Vis. https://doi.org/10.1109/VISUAL.2005.1532795

Cordeil M, Cunningham A, Bach B, Hurter C, Thomas BH, Marriott K, Dwyer T (2019) IATK: an immersive analytics toolkit. IEEE Virtual Real 3D User Interfaces. https://doi.org/10.1109/VR.2019.8797978

Chandler T, Cordeil M, Czauderna T, Dwyer T, Glowacki J, Goncu C, Klapperstueck M, Klein K, Marriott K, Schreiber F, Wilson E (2015) Immersive analytics. In: Big Data Visual Analytics. https://doi.org/10.1109/BDVA.2015.7314296

Doutreligne S, Cragnolini T, Pasquali S, Derreumaux P, Baaden M (2014) UnityMol: interactive scientific visualization for integrative biology. IEEE Large Data Anal Vis. https://doi.org/10.1109/LDAV.2014.7013213

Donatiello L, Gasparini L, Marfia G (2021) Towards an immersive visualization of consumer-level simulations of vehicular traffic. IEEE/ACM Distrib Simul Real Time Appl. https://doi.org/10.1109/DS-RT52167.2021.9576155

Doerr K-U, Kuester F (2011) CGLX: a scalable, high-performance visualization framework for networked display environments. IEEE Trans Vis Comput Graph 17(3):320–332. https://doi.org/10.1109/TVCG.2010.59

Dużmańska N, Strojny P, Strojny A (2018) Can simulator sickness be avoided? A review on temporal aspects of simulator sickness. Front Psychol 9:2132. https://doi.org/10.3389/fpsyg.2018.02132

Decoret X, Sillion F, Schaufler G, Dorsey J (1999) Multi-layered impostors for accelerated rendering. Comput Graph Forum 18(3):61–73. https://doi.org/10.1111/1467-8659.00328

Ens B, Bach B, Cordeil M, Engelke U, Serrano M, Willett W, Prouzeau A, Anthes C, Büschel W, Dunne C, Dwyer T, Grubert J, Haga JH, Kirshenbaum N, Kobayashi D, Lin T, Olaosebikan M, Pointecker F, Saffo D, Saquib N, Schmalstieg D, Szafir DA, Whitlock M, Yang Y (2021) Grand challenges in immersive analytics. In: Conference on Human Factors in Computing Systems, pp. 1–17. Association for Computing Machinery, New York. https://doi.org/10.1145/3411764.3446866

Egger J, Gall M, Wallner J, Boechat P, Hann A, Li X, Chen X, Schmalstieg D (2017) HTC Vive MeVisLab integration via OpenVR for medical applications. PLOS ONE 12(3):1–14. https://doi.org/10.1371/journal.pone.0173972

Fuhrmann A, Löffelmann H, Schmalstieg D (1997) Collaborative augmented reality: exploring dynamical systems. IEEE Vis. https://doi.org/10.1109/VISUAL.1997.663921

Fonnet A, Prié Y (2021) Survey of immersive analytics. IEEE Trans Vis Comput Graph 27(3):2101–2122. https://doi.org/10.1109/TVCG.2019.2929033

Fleck P, Sousa Calepso A, Hubenschmid S, Sedlmair M, Schmalstieg D (2022) RagRug: A toolkit for situated analytics. IEEE Trans Visual Comput Graph. https://doi.org/10.1109/TVCG.2022.3157058

Gralka P, Becher M, Braun M, Frieß F, Müller C, Rau T, Schatz K, Schulz C, Krone M, Reina G, Ertl T (2019) MegaMol—a comprehensive prototyping framework for visualizations. Eur Phys J Special Topics 227(14):1817–1829. https://doi.org/10.1140/epjst/e2019-800167-5

Geringer S, Geiselhart F, Bäuerle A, Dec D, Odenthal O, Reina G, Ropinski T, Weiskopf D (2024) Supplemental material for: mint: integrating scientific visualizations into virtual reality. DaRUS https://doi.org/10.18419/darus-4060

Gupta A, Günther U, Incardona P, Reina G, Frey S, Gumhold S, Sbalzarini IF (2023) Efficient raycasting of volumetric depth images for remote visualization of large volumes at high frame rates. Pacific Vis. https://doi.org/10.1109/PacificVis56936.2023.00014

García-Hernández RJ, Kranzlmüller D (2019) NOMAD VR: Multiplatform virtual reality viewer for chemistry simulations. Comput Phys Commun 237:230–237. https://doi.org/10.1016/j.cpc.2018.11.013

Grottel S, Krone M, Müller C, Reina G, Ertl T (2015) MegaMol—A prototyping framework for particle-based visualization. IEEE Trans Vis Comput Graphi 21(2):201–214. https://doi.org/10.1109/TVCG.2014.2350479
Günther U, Pietzsch T, Gupta A, Harrington KIS, Tomancak P, Gumhold S, Sbalzarini IF (2019) scenery: flexible virtual reality visualization on the java VM. IEEE Vis. https://doi.org/10.1109/VISUAL.2019.8933605
Gold M, Subtil N (2010) NV_DX_interop OpenGL Extension. Technical report, Khronos OpenGL Registry. https://registry.khronos.org/OpenGL/extensions/NV/WGL_NV_DX_interop.txt Accessed 2024-02-28
Gilg J, Zander S, Schneegans S, Ahlers V, Gerndt A (2021) Comparison of depth buffer techniques for large and detailed 3D scenes. In: GI VR / AR Workshop. Gesellschaft für Informatik e.V., Bonn. https://doi.org/10.18420/vrar2021_12
Haase H (1996) Symbiosis of virtual reality and scientific visualization system. Comput Graph Forum 15(3):443–451. https://doi.org/10.1111/1467-8659.1530443
Haeberli PE (1988) ConMan: A visual programming language for interactive graphics. Comput Graph Interact Techniq 22(4):103–111. https://doi.org/10.1145/54852.378494
Humphrey W, Dalke A, Schulten K (1996) VMD: visual molecular dynamics. J Mol Graph 14(1):33–38. https://doi.org/10.1016/0263-7855(96)00018-5
Humphreys G, Eldridge M, Buck I, Stoll G, Everett M, Hanrahan P (2001) WireGL: A scalable graphics system for clusters. Comput Graph Interact Techniq. https://doi.org/10.1145/383259.383272
Haber RB, McNabb DA (1990) Visualization idioms: a conceptual model for scientific visualization systems. In: Nielson GM, Shriver BD, Rosenblum LJ (eds) Visualization in Scientific Computing. IEEE Computer Society Press, London, pp 74–93
Hanwell MD, Martin KM, Chaudhary A, Avila LS (2015) The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. SoftwareX 1–2:9–12. https://doi.org/10.1016/j.softx.2015.04.001
Heinen M, Vrabec J (2019) Evaporation sampled by stationary molecular dynamics simulation. J Chem Phys 151(4):044704. https://doi.org/10.1063/1.5111759
Jönsson D, Steneteg P, Sundén E, Englund R, Kottravel S, Falk M, Ynnerman A, Hotz I, Ropinski T (2019) Inviwo—A visualization system with usage abstraction levels. IEEE Trans Vis Comput Graph 26(11):3241–3254. https://doi.org/10.1109/TVCG.2019.2920639
Keefe DF, Acevedo D, Miles J, Drury F, Swartz SM, Laidlaw DH (2008) Scientific sketching for collaborative VR visualization design. IEEE Trans Vis Comput Graph 14(4):835–847. https://doi.org/10.1109/TVCG.2008.31
Kreylos O, Bawden G, Bernardin T, Billen MI, Cowgill ES, Gold RD, Hamann B, Jadamec M, Kellogg LH, Staadt OG, Sumner DY (2006) Enabling scientific workflows in virtual reality. ACM Virt Real Contin Appl. https://doi.org/10.1145/1128923.1128948
Kozlikova B, Krone M, Falk M, Lindow N, Baaden M, Baum D, Viola I, Parulek J, Hege H-C (2017) Visualization of biomolecular structures: state of the art revisited. Comput Graph Forum 36(8):178–204. https://doi.org/10.1111/cgf.13072
Kaufmann H, Schmalstieg D (2002) Mathematics and geometry education with collaborative augmented reality. Comput Graph Interact Techn. https://doi.org/10.1145/1242073.1242086
Koenig M, Spindler W, Rexilius J, Jomier J, Link F, Peitgen H-O (2006) Embedding VTK and ITK into a visual programming and rapid prototyping platform. In: Cleary KR, Robert L Galloway J (eds.) Medical Imaging 2006: Visualization, Image-Guided Procedures, and Display, vol. 6141, pp. 796–806. SPIE, Bellingham. https://doi.org/10.1117/12.652102
Liu Z, Heer J (2014) The effects of interactive latency on exploratory visual analysis. IEEE Trans Vis Comput Graph 20(12):2122–2131. https://doi.org/10.1109/TVCG.2014.2346452
Misiak M, Fuhrmann A, Latoschik ME (2021) Impostor-based rendering acceleration for virtual, augmented, and mixed reality. ACM Virt Real Softw Technol. https://doi.org/10.1145/3489849.3489865
Mohr A, Gleicher M (2002) HijackGL: Reconstructing from streams for stylized rendering. Non-Photoreal Anim Render. https://doi.org/10.1145/508530.508533
Microsoft: ID3D11Device::OpenSharedResource method. Technical report, Direct3D 11 Graphics API (2021). https://learn.microsoft.com/en-us/windows/win32/api/d3d11/nf-d3d11-id3d11device-opensharedresource Accessed 2024-02-28
Moreland K (2013) A survey of visualization pipelines. IEEE Trans Vis Comput Graph 19(3):367–378. https://doi.org/10.1109/TVCG.2012.133
Marriott K, Schreiber F, Dwyer T, Klein K, Riche NH, Itoh T, Stuerzlinger W, Thomas BH (eds.): (2018) Immersive Analytics. Springer, Cham. https://doi.org/10.1007/978-3-030-01388-2
Moreland K, Sewell C, Usher W, Lo L-T, Meredith J, Pugmire D, Kress J, Schroots H, Ma K-L, Childs H, Larsen M, Chen C-M, Maynard R, Geveci B (2016) VTK-m: accelerating the visualization toolkit for massively threaded architectures. IEEE Comput Graph Appl 36(3):48–58. https://doi.org/10.1109/MCG.2016.48
McGhee J, Thompson-Butel AG, Faux S, Bou-Haidar P, Bailey J (2015) The Fantastic Voyage: an arts-led approach to 3D virtual reality visualization of clinical stroke data. Visual Inf Commun Interact. https://doi.org/10.1145/2801040.2801051
Mueller JH, Voglreiter P, Dokter M, Neff T, Makar M, Steinberger M, Schmalstieg D (2018) Shading atlas streaming. ACM Trans Graph 37(6):199. https://doi.org/10.1145/3272127.3275087
Marino G, Vercelli D, Tecchia F, Gasparello PS, Bergamasco M (2007) Description and performance analysis of a distributed rendering architecture for virtual environments. Artif Real Telexistence. https://doi.org/10.1109/ICAT.2007.58
Norman D (2013) The Design of Everyday Things: Revised and Expanded Edition. Basic Books, New York
O'Leary P, Jhaveri S, Chaudhary A, Sherman W, Martin K, Lonie D, Whiting E, Money J, McKenzie S (2017) Enhancements to VTK enabling scientific visualization in immersive environments. IEEE Virt Real. https://doi.org/10.1109/VR.2017.7892246
Prodromou E, Leandrou S, Schiza E, Neocleous K, Matsangidou M, Pattichis CS (2020) A multi-user virtual reality application for visualization and analysis in medical imaging. Bioinform Bioeng. https://doi.org/10.1109/BIBE50027.2020.00135
Reina G, Childs H, Matković K, Bühler K, Waldner M, Pugmire D, Kozlíková B, Ropinski T, Ljung P, Itoh T, Gröller E, Krone M (2020) The moving target of visualization software for an increasingly complex world. Comput Graph 87:12–29. https://doi.org/10.1016/j.cag.2020.01.005
Richer G, Pister A, Abdelaal M, Fekete J-D, Sedlmair M, Weiskopf D (2022) Scalability in visualization. IEEE Trans Vis Comput Graph. https://doi.org/10.1109/TVCG.2022.3231230

Shetty N, Chaudhary A, Coming D, Sherman WR, O'Leary P, Whiting ET, Su S (2011) Immersive ParaView: a community-based, immersive, universal scientific visualization application. IEEE Virt Real. https://doi.org/10.1109/VR.2011.5759487
Schaufler G (1996) Exploiting frame-to-frame coherence in a virtual reality system. IEEE Virt Real. https://doi.org/10.1109/VRAIS.1996.490516
Saad EW, Caudell TP, Wunsch DC (1999) Predictive head tracking for virtual reality. In: International Joint Conference on Neural Networks, vol. 6, pp. 3933–3936. IEEE, Washington. https://doi.org/10.1109/IJCNN.1999.830785
Shaw C, Green M, Liang J, Sun Y (1993) Decoupled simulation in virtual reality with the MR toolkit. ACM Trans Inf Syst 11(3):287–317. https://doi.org/10.1145/159161.173948
Shi S, Hsu C-H (2015) A survey of interactive remote rendering systems. ACM Comput Surv 47(4):1–29. https://doi.org/10.1145/2719921
Sicat R, Li J, Choi J, Cordeil M, Jeong W-K, Bach B, Pfister H (2019) DXR: A toolkit for building immersive data visualizations. IEEE Trans Vis Comput Graph 25(1):715–725. https://doi.org/10.1109/TVCG.2018.2865152
Stegmaier S, Magallón M, Ertl T (2002) A generic solution for hardware-accelerated remote visualization. IEEE Vis. https://doi.org/10.2312/VisSym/VisSym02/087-094
Schroeder W, Martin K, Lorensen B (2006) The Visualization Toolkit. Kitware, New York
Speir JA, Munshi S, Wang G, Baker TS, Johnson JE (1995) Structures of the native and swollen forms of cowpea chlorotic mottle virus determined by X-ray crystallography and cryo-electron microscopy. Structure 3(1):63–78. https://doi.org/10.1016/S0969-2126(01)00135-6
Shi S, Nahrstedt K, Campbell R (2012) A real-time remote rendering system for interactive mobile graphics. ACM Trans Multim Comput Commun Appl 8(3s):1–20. https://doi.org/10.1145/2348816.2348825
Stauffert J-P, Niebling F, Latoschik ME (2020) Latency and cybersickness: impact, causes, and measures: a review. Front Virt Real. https://doi.org/10.3389/frvir.2020.582204
Sonntag S, Roth J, Gaehler F, Trebin H-R (2009) Femtosecond laser ablation of aluminium. Appl Surf Sci 255(24):9742–9744. https://doi.org/10.1016/j.apsusc.2009.04.062
Schaufler G, Stürzlinger W (1996) A three dimensional image cache for virtual reality. Comput Graph Forum 15(3):227–235. https://doi.org/10.1111/1467-8659.1530227
Stukowski A (2009) Visualization and analysis of atomistic simulation data with OVITO-the open visualization tool. Model Simul Mater Sci Eng 18(1):015012. https://doi.org/10.1088/0965-0393/18/1/015012
Thompson AP, Aktulga HM, Berger R, Bolintineanu DS, Brown WM, Crozier PS, in 't Veld PJ, Kohlmeyer A, Moore SG, Nguyen TD, Shan R, Stevens MJ, Tranchida J, Trott C, Plimpton SJ, (2022) LAMMPS–a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. Comput Phys Commun 271:108171. https://doi.org/10.1016/j.cpc.2021.108171
Upson C, Faulhaber TA, Kamins D, Laidlaw D, Schlegel D, Vroom J, Gurwitz R, Dam A (1989) The application visualization system: a computational environment for scientific visualization. IEEE Comput Graph Appl 9(4):30–42. https://doi.org/10.1109/38.31462
User "mlavik1" on GitHub: Unity Volume Rendering Plugin. https://github.com/mlavik1/UnityVolumeRendering Accessed 2024-02-28
Verhille V, Bayart B, Piuzzi M, Vartanian A (2014) How to Easily Develop a VR Experience from a 3D Desktop Application Thanks to the TechViz TVZLib API. In: Perret J, Basso V, Ferrise F, Helin K, Lepetit V, Ritchie J, Runde C, Voort M, Zachmann G (eds.) EuroVR. The Eurographics Association, Goslar. https://doi.org/10.2312/eurovr.20141333
Waveren JMP (2016) The asynchronous time warp for virtual reality on consumer hardware. ACM Virt Real Softw Technol. https://doi.org/10.1145/2993369.2993375
Wheeler G, Deng S, Toussaint N, Pushparajah K, Schnabel JA, Simpson JM, Gomez A (2018) Virtual interaction and visualisation of 3D medical imaging data with VTK and Unity. Healthcare Technol Lett 5(5):148–153. https://doi.org/10.1049/htl.2018.5064
Wald I, Johnson G, Amstutz J, Brownlee C, Knoll A, Jeffers J, Günther J, Navratil P (2017) OSPRay—A CPU ray tracing framework for scientific visualization. IEEE Trans Vis Comput Graph 23(1):931–940. https://doi.org/10.1109/TVCG.2016.2599041
Wald I, Knoll A, Johnson GP, Usher W, Pascucci V, Papka ME (2015) CPU ray tracing large particle data with balanced P-k-d trees. IEEE Sci Vis. https://doi.org/10.1109/SciVis.2015.7429492
Wang J, Shi R, Zheng W, Xie W, Kao D, Liang H-N (2023) Effect of frame rate on user experience, performance, and simulator sickness in virtual reality. IEEE Trans Vis Comput Graph 29(5):2478–2488. https://doi.org/10.1109/TVCG.2023.3247057
Ynnerman A, Rydell T, Antoine D, Hughes D, Persson A, Ljung P (2016) Interactive visualization of 3D scanned mummies at public venues. Commun ACM 59(12):72–81. https://doi.org/10.1145/2950040
Zellmann S, Aumüller M, Lang U (2012) Image-based remote real-time volume rendering: decoupling rendering from view point updates. Comput Inf Eng 2:1385–1394. https://doi.org/10.1115/DETC2012-70811
Zielinski DJ, McMahan RP, Lu W, Ferrari S (2013) ML2VR: Providing MATLAB users an easy transition to virtual reality and immersive interactivity. IEEE Virt Real. https://doi.org/10.1109/VR.2013.6549374
Zielinski DJ, McMahan RP, Shokur S, Morya E, Kopper R (2014) Enabling closed-source applications for virtual reality via OpenGL intercept-based techniques. IEEE Workshop Softw Eng Arch Realtime Interact Syst. https://doi.org/10.1109/SEARIS.2014.7152802