Computers & Graphics 87 (2020) 12–29

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Survey Paper

The moving target of visualization software for an increasingly complex world

Guido Reina[a,*], Hank Childs[b], Krešimir Matković[c], Katja Bühler[c], Manuela Waldner[d], David Pugmire[e], Barbora Kozlíková[f], Timo Ropinski[g], Patric Ljung[h], Takayuki Itoh[i], Eduard Gröller[d,c], Michael Krone[j,*]

[a] University of Stuttgart, Germany
[b] University of Oregon, Eugene, OR, USA
[c] VRVis Research Center, Vienna, Austria
[d] TU Wien, Vienna, Austria
[e] Oak Ridge National Laboratory, Oak Ridge, TN, USA
[f] Masaryk University, Brno, Czech Republic
[g] Ulm University, Ulm, Germany
[h] Linköping University, Norrköping, Sweden
[i] Ochanomizu University, Tokyo, Japan
[j] University of Tübingen, Germany

ARTICLE INFO

Article history:
Received 26 September 2019
Revised 16 January 2020
Accepted 20 January 2020
Available online 25 January 2020

Keywords:
Software engineering
Visualization
Visualization community
Visualization research
Visualization software

ABSTRACT

Visualization has evolved into a mature scientific field and it has also become widely accepted as a standard approach in diverse fields, including physics, life sciences, and business intelligence. However, despite its successful development, there are still many open research questions that require customized implementations in order to explore and establish concepts, and to perform experiments and take measurements. Many methods and tools have been developed and published but most are stand-alone prototypes and have not reached a mature state that can be used in a reliable manner by collaborating domain scientists or a wider audience. In this study, we discuss the challenges, solutions, and open research questions that affect the development of sophisticated, relevant, and novel scientific visualization solutions with minimum overheads. We summarize and discuss the results of a recent National Institute of Informatics Shonan seminar on these topics.

1. Introduction

Visualization is a sub-field of computer science based on methods for conveying complex information by data visualization to support a hypothesis, to extract a model of a particular phenomenon, or to communicate knowledge across a spectrum of audiences. These methods are applied by implementing and using visualization software designed for a particular purpose or for more generic workflows. Visualization has matured considerably as a field and a community, and it is a widely accepted technique for analyzing various data types and data sources. The maturity of visualization is also reflected by the way we try to formalize and understand the basic processes and workflows that comprise successful visualization approaches in the context of specific application scenarios and science domains. For example, Munzner [1] formalized the requirements analysis and design of effective visualization methods, as well as associating common data types and user requirements with well-known visualization methods. Chen et al. [2] considered visualization from the standpoint of information theory by analyzing the information loss and communication channels. They then developed an ontological framework that allows experts to analyze the technical shortcomings in visual analytics systems [3]. Van Wijk proposed a method for modeling the *value* of visualization [4], which includes the cost factors for developers and users as well as the knowledge that can be generated by utilizing a system.

Despite all these modeling efforts, the production of novel visualization approaches and implementing known methods are hindered by major practical hurdles in terms of the actual

* Corresponding author.
E-mail addresses: guido.reina@visus.uni-stuttgart.de (G. Reina), michael.krone@uni-tuebingen.de (M. Krone).

https://doi.org/10.1016/j.cag.2020.01.005
0097-8493/

implementation of a working software prototype that can provide performance insights and/or withstand the tests conducted by domain users in order to properly evaluate a proposed approach.

This issue is easy to dismiss given that our community has implemented, relies on, and is still constantly producing an excessive number of (open source) frameworks, libraries, engines, and domain-specific languages. In addition, the challenges posed by realistic data sets [5,6] and the approaches required (and thus the sheer quantity of implemented code) are becoming increasingly complex. These problems considerably increase the ramp-up time for every new graduate student and given the ever-changing requirements of diverse domains, the entry barrier for valuable contributions in our field is indeed a moving target. In this study, the term *visualization software* encompasses the whole range from one-shot software prototypes to feature-rich software platforms that are under continuous development and extension over a long period of time.

In order to address these challenges, we met for a National Institute of Informatics Shonan Seminar [7] to discuss and identify the demands that our community faces when working with the diverse range of existing software and planning new applications from a research and engineering perspective. The engineering side is still often overlooked despite its considerable impact on the effort required for all of our research. In this study, we present a summary of our findings, where we summarize the pitfalls, challenges, and opportunities that may arise. We aim to increase awareness in our community of the need to take opportunities and agree on viable solutions, common benchmarks, and efforts to reduce the overheads we must constantly overcome. New members of our community should be allowed to stand on the "shoulders of giants" [8] from theoretical and methodological viewpoints, but especially in practical terms when engineering an actual approach. Given the composition of the discussion group at the Shonan seminar, some of the results presented in this study are biased toward scientific visualization, but most of the challenges identified are generally applicable to the entire visualization field. Information visualization and visual analytics have a wide range of applications and the large potential user base has led to stable and commercially successful solutions, such as Tableau or Spotfire. The tasks and solutions in scientific visualization are usually very narrow in scope, which often leads to many small, specialized solutions.

The remainder of this paper is organized according to the different challenges and opportunities that we identified as most important, which we discussed in greater depth during the one-week seminar. We introduce the topics in the same order that they were discussed at the seminar, and we aim to make each topic as self-contained as possible. In Section 2, we discuss how the **ecosystem** encompassing developments and novel approaches from other communities might influence visualization software. In Section 3, we consider the **user experience** of visualization software from the perspective of developers and end users (domain scientists). In Section 4, we analyze the **barriers to entry** from a developer's (and thus a visualization researcher's) viewpoint. In Section 5, we discuss the appropriate modularization of visualization approaches as **building blocks**, especially the choice of an appropriate abstraction level. We then address the question of whether establishing a **single visualization system** would solve many of the problems that we face (Section 6), or if employing available **game engines** might be the most appropriate solution (Section 7). In Section 8, we expand on the **sustainability** of existing solutions and the influence of community activities and funding. In Section 9, we consider the **evaluation** of complex software systems and the quantification of success for visualization software. Finally, we propose an initial **typology** in Section 10 to provide a basis for further elaborate and systematic investigations of the current state of the available visualization software. The remaining challenges may be discussed in a future manuscript.

2. The ecosystem beyond visualization

Background and Motivation. In 2004, in his study "On the Death of Visualization" [9], William Lorensen highlighted the need to "form alliances with other fields" to ensure the survival of visualization research. Visualization research survived and it is now common practice for visualization software to incorporate technology from other communities. A good example from scientific visualization is the need to deliver graphics capabilities by incorporating rendering algorithms from computer graphics. Further examples are present at all stages of the classical visualization pipeline [10], as shown in Fig. 1, and visualization itself has always bridged a number of disciplines. We need to consider when and how to incorporate external technologies for our community to efficiently utilize our collective resources. Ideally, adopting external technologies will lead to improved visualization software, reduced development costs, or both.

The need to incorporate external technologies has become more urgent in recent years because the deployment environment for visualization software has become increasingly complex. In the past, visualization development frequently involved producing a stand-alone binary that could access data on a local file system and display it to a user's monitor. The current requirements for visualization software are more challenging, where they may include delivery on the web as a service, accessing data in heterogeneous remote storage systems such as the cloud, working with data sets that exceed the RAM or even HDD capacity of a desktop computer [11,12], operating within large software ecosystems where visualization is just one aspect of the larger overall mission, or displaying to output systems ranging from mobile and virtual reality devices to power walls. The solutions to these challenges often come from other communities and it is often cost efficient for our community to employ their solutions by borrowing rather than building.

Challenges. Incorporating external technology presents many challenges. In particular, we focus on the life span, future-proofing, interoperability, and costs. These challenges have been well understood by the software engineering community for several decades [13,14], but we think it necessary to reiterate and discuss the aspects that are most relevant for visualization software.

Life span is an important consideration for both the visualization software and the external technology that may be incorporated. The life span can vary greatly for visualization software: many projects might last for only a few years, whereas others continue for much longer. For example, VMD [15] was developed in the mid-1990s and it is still being extended [16]. VTK [17] was initiated in the mid-1990s and two popular end-user applications were introduced in the early 2000s, i.e., ParaView [18] and VisIt [19]. All three products are still undergoing active development. Similarly, external technology may have different life spans and various releases. OpenGL [20] has been used extensively by our community, but it may be supplanted by Vulkan [21] or Metal [22] in the near future. In addition, many visualization software packages currently utilize Python V2.7, but this version will not be maintained after January 1, 2020. The basic concerns for visualization software in terms of the life span include: (1) incorporating an external technology that ends support, (2) making assumptions about external technology that might make change difficult in the future, and (3) being forced to allocate developer time updating to new versions of libraries.

Future-proofing a software design is closely connected to a software's life span. Many visualization programs have simple

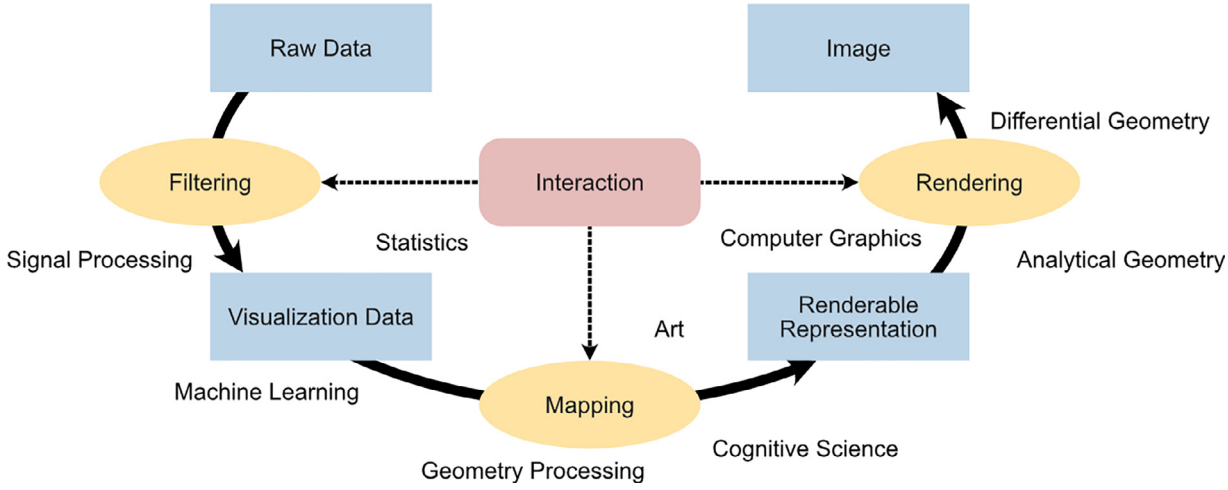

Fig. 1. Classical visualization pipeline annotated with examples of influences from other disciplines in different stages.

extensibility paths for new algorithms but some new paradigms are sufficiently different to break these extensibility paths. For example, a scientific visualization program may be well designed to incorporate new flow visualization algorithms, but unsuitable for incorporating new user interface patterns. Another consideration is the hardware employed. Visualization software is naturally affected by advances in hardware and the evolution of concepts about how to harness these in the visualization pipeline. Recent examples for our community include GPUs, multi-core CPUs, and distributed-memory parallelism, as well as upcoming examples including FPGA and Tensor Processing Units. The choice of programming language also impacts future-proofing. Contributions in common or popular languages are more likely to develop a significant user base, whereas unpopular languages are likely to hinder adoption. As a once popular language choice, Java has now been superseded by alternatives such as JavaScript and Python [23]. Many systems such as VTK-m [24] are based on C++, which makes intensive use of template metaprogramming for efficiency. Shader languages have also evolved tremendously over the last decade and a half. Furthermore, the choice of language influences the impact of the code produced. In terms of best practice, we consider that it is critical to understand the purpose of the developed software (e.g., performance and accessibility to large groups) and then match with a programming language that is aligned to the purpose. Many scientific visualization applications value performance very highly. We note that C++ has evolved its standard to adapt to the pervasiveness of parallelism, thereby allowing it to play a significant role in the future for applications that require high performance.

Interoperability is another important challenge. In particular, a significant issue is the data formats that are often introduced by domain scientists without considering interoperability or visualization-friendly properties, such as their suitability for stream processing or parallelization. Another significant issue is the programming language employed, where some languages facilitate interoperability, such as Python and C, whereas others do not. Choosing a language naturally enables interoperability with some technologies, but makes adoption more difficult for others.

Cost, specifically minimizing the cost of adopting other technologies, is the final challenge that we consider. Some technologies are at the forefront of technological development, thereby making their adoption a significant effort with uncertain prospects. In the situation where new technologies do not prevail, early adopters are specifically affected and must search for alternatives. Given the generally rapid evolution of computer science and related technology, any software that persists for at least five years has a strong likelihood of witnessing the emergence of a disruptive technology (e.g., see Fig. 2), which is usually difficult to adopt and initially unstable. A smaller challenge is the steep learning curve because studying how to effectively exploit new technology takes time. These points align with the motivations behind lean software development [25], which considers the tradeoffs between costs (development, adoption, and deployment) and benefits.

Conclusion. We will now discuss the aforementioned challenges using the specific example of applying machine learning—particularly deep learning—in visualization research. The motivation for considering machine learning is the potential for this technology to either overcome open challenges, or at least solving these challenges with less developer time.

Here, **future-proofing** is crucial because machine learning software will probably be very different in 10 years' time. The visualization developer must design software that meets the needs of visualization but it must also be **interoperable** with machine learning technology. If the software has a **life span** of a decade or more, it is likely that the code will have to be adapted to multiple machine-learning development cycles. Finally, the adoption **cost** is high because visualization researchers frequently have no familiarity with machine learning software.

The overall costs and the benefits of incorporating external technology can be high. A project that decides against incorporating external technology may keep the costs low, but it cannot harness potential benefits. Conversely, a project that overly embraces external technology runs the risk of high integration costs and limited benefits. New technologies can open up promising research avenues and inspire novel approaches to old problems. In particular, Ph.D. students will have the opportunity to prototype new ideas with the latest technologies, study new problems, and extend their area of expertise.

3. Effectiveness and user experience

Background and Motivation. In this section, we discuss why many visualization packages are difficult to use, which impedes their adoption by domain scientists. During the workshop, one participant stated that using the software we develop requires domain scientists to have PhD level expertise in visualization. This concern does not apply in general because some of the software produced by our community is easy to use, such as Tableau (formerly Polaris [26]). Other products such as FieldView [27] and EnSight [28] are generating licensing fees, thereby indicating that

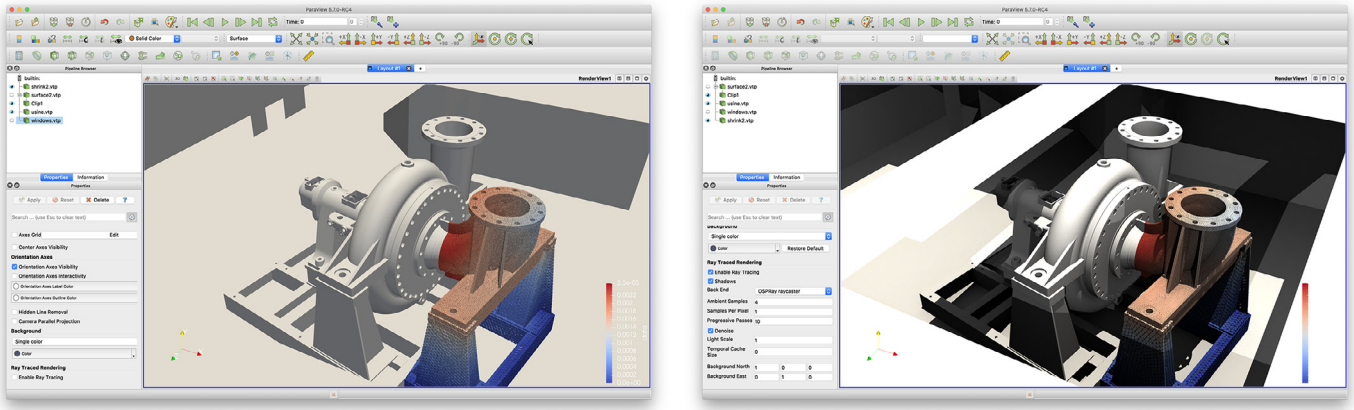

Fig. 2. Visualizations of the EDF power plant data set: left, using the conventional OpenGL rendering in ParaView; right, using ParaView's OSPRay support that adds shadows and ambient occlusion for improved structure perception. The CPU-based OSPRay ray tracer can be considered disruptive because it demonstrates the advantages of relying on CPUs, whereas most visualization research implicitly assumes that scalable approaches require a GPU implementation. In addition, since the introduction of OSPRay, the adoption of ray tracing has increased considerably in visualization. Images courtesy of Aaron Knoll. Data set provided by Kitware. Ownership of the data is attributed to Électricité De France.

the user experience (UX) is at least acceptable. Further, open-source tools such as ParaView [18] and VisIt [19] have large user communities, which again implies their usability. However, the UX is a significant issue, where ParaView and VisIt have vast GUIs that require a significant amount of time to understand.

Challenges. A major issue is the goals we try to satisfy with the software we create. These goals cover a wide spectrum, ranging from an individual PhD student developing software to advance their research (research software) to a commercial company trying to make a profitable product (commercial software). Thus, conflicting goals might emerge for research software where domain scientists want to solve problems in their application areas, whereas visualization scientists are more interested in developing novel visualization techniques. Furthermore, visualization scientists often provide research prototypes, whereas the end user is expecting production quality by using industrial-grade software such as iOS and PowerPoint as a frame of reference. In these cases, the resources needed to develop software must be carefully managed. Visualization scientists cannot afford to invest months of development time in order to save the domain scientist only minutes when using their software. Thus, there is often a mismatch between the expectations of the visualization researcher (perceived as a developer) who wants to spend the least possible time polishing a prototype and the domain scientist (perceived as user) who wants the best possible UX. In many cases, visualization scientists can do more to close this gap, such as by increasing their knowledge of human–computer interactions, exhibiting greater commitment to writing documentation, and enhancing their visual literacy [29,30]. An alternative strategy involves avoiding writing user interface code (UI) altogether and deploying new approaches within existing frameworks instead. For example, ParaView and VisIt both employ building block-based approaches that allow developers to focus only on building their new capability and then fitting that capability into the existing interface.

There are also several open research questions regarding the usability of visualization software with respect to UI design. The concept of "user guidance" has been studied by the visual analytics community [31]. Thus, there is an opportunity to collaborate with this community by focusing on UX and helping to define the application of these principles to visualization software. The application of these concepts by the visualization community could include providing hints and tool tips in the UI, and "'wizard" dialog boxes that automatically construct common workflows. Furthermore, restrictions based on the type of data loaded could be enforced, such as by disabling operations that are not possible on the data or in the current workflow. Two high-level questions concern the automatic generation of simplified versions of UIs, such as dashboards, and the appropriate UI model to use in the first place. In the first cases, an important sub-topic involves identifying the building blocks for automatic UI simplification. In the second case, there are many possible directions, where the UI could ask users what they want to do in a similar manner to a conversation. In addition, the program could suggest examples of successful visualizations as a starting template (cf. Liu et al. [32] or Alexa et al. [33]). Machine learning could be employed to guide the user or filter what they see (e.g., see Gotz et al. [34] or Keck and Groh [35]).

Conclusion. Making visualization software more usable would have benefits for both visualization and domain scientists. For domain scientists, the improved outcomes may include solving problems more rapidly (time savings) to discover more in the same period of time (increased knowledge). There are also additional benefits for visualization scientists. In particular, wider user adoption will lead to multiple positive outcomes, including increased funding, an increased reputation in the community, and an increased competitive advantage (for corporations). Moreover, making software easier to use requires less user support, which saves time and effort for visualization scientists. A sufficiently large user community can start to support itself (with Blogs, mailing lists, Wikis, etc.), which can save further time for visualization scientists. More usable software increases the likelihood of commercialization by a company or by industrious students who might attempt to bring their research to the marketplace. Finally, thorough evaluation of software by actual users is significantly simpler if the software offers a reasonable UX.

4. Entry barriers to existing solutions

Background and Motivation. Software products require active and vibrant communities of users and developers in order to thrive in the present and adapt and change to prepare for the future. The long-term success of any software project depends on the ability of developers to maintain, modify, and integrate it with other related software components, including the ease with which new developers can join a project and contribute rapidly. At present,

the entry cost for productive development of existing visualization software solutions is high, and this is even true for experienced graduate students with domain expertise. The task is equally challenging for the users and developers of visualization software, mainly because of the need to have expertise in both visualization techniques and software engineering.

Some of the common barriers include a lack of comprehensive documentation. This documentation should extend beyond reference manuals and it must include the design rationale and motivation (for users), as well as design documents describing the overall architecture of the software and how specific features fit together (for developers). It should also include examples of how to add additional functionality, the coding style and practice, and the software engineering process. It is also important to document how the compatibility between software modules is maintained, and the intended interfaces for inter-operability with other software tools. The vibrancy of both the user and developer communities will determine the long-term success of community visualization tools, where this relies on low entry barriers.

Challenges. High entry barriers exist for both the user and developer communities. For developers, the primary concerns are that the documentation is often outdated, non-existent, or incorrect. In an ongoing development, internal changes can affect the system design or programming interfaces even if they do not change the user experience. In a research environment, time and motivation constraints often result in outdated documentation, which makes it difficult for developers to understand the architecture of tools and learn how to modify, enhance, or fix issues.

Another challenge is that not all developers know the same programming languages. This is especially important for PhD students because it will result in delays as they may need to learn a new programming language or search for another tool.

Another challenge for developers is the potentially complicated build process for complex visualization software. Visualization tools often have external dependencies on various libraries. Managing (especially backward) the compatibilities of a large number of libraries is a maintenance problem. It is quite difficult to build large visualization tools even if stable third party libraries are well supported and available. The diversity of operating systems and compilers is growing and the maintenance of stable build processes is demanding.

If any of these challenges are not met, developers tend to move on search for other solutions that will work on their particular platform with the minimal amount of effort. Developers will favor software that is highly trusted with characteristics such as longevity, stability, bug tracking and monitoring, platform support, and appropriate coding styles.

For the user community, documentation about how to install and use the software is of key importance, including user manuals, feature lists, tutorials, and examples. In particular, user-based tutorials including videos and blogs are very widely used for game engines. In general, occasional users have different expectations and requirements compared with intensive users.

Users typically do not want to build the software tools, which places the burden of providing installable solutions on the development team. For users, one-click solutions such as Docker, Singularity, or web-based visualization tools have lower entry barriers.

Good user interfaces for visualization software are an ongoing research topic and they evolve as newer technology becomes available. It is difficult to design a user interface that is effective for both occasional and intensive users. In addition, not all potential software usage scenarios can be anticipated. Providing an adequate interface for all use cases is intrinsically impossible (see Fig. 3). Therefore, re-configurable UI and UX for existing software has been studied as a middle ground [36].

Conclusion. The visualization community has tried a number of approaches for overcoming entry barriers. In particular, visualization software has been wrapped into interpreted languages such as Python and Lua to aid developers. Meta-build tools can simplify the build process, such as *build_visit* for VisIt and *superbuild* for ParaView. Package management simplifies the installation of visualization software (e.g., Nuget, npm, macport, conda, and spack). Visual programming has been included in many visualization systems and pipeline designers have been implemented (e.g., all systems in Fig. 3 use this concept in different flavors). This paradigm also extends to game engines, where one example is Blueprints in Unreal Engine. Repeated tasks are supported by macros and compound modules in some visualization software.

Lowering the entry barriers involves a number of open research topics, including finding the best way to mix and match features from different systems to create custom solutions, determining the appropriate designs for better re-usability in visualization software, measuring the entry barriers for developers and users alike, and generating and updating documentation for both developers and users.

We recommend the following solutions to lower the entry barriers for visualization software. The first recommendation is targeted at improving the user experience, which means that this topic is tightly linked to the challenges discussed in Section 3, e.g., wizards that construct a suitable workflow for a specific data type and task. Another effective training aid involves cooperation between domain scientists and the visualization community. Strategies such as “tiger teams” or “liaisons” pair domain scientists with visualization experts for targeted visualization projects to allow the free flow of both domain and visualization knowledge between people. These strategies can help domain scientists to understand the most efficient ways to use tools, as well as informing visualization developers about how the tools are being used and how they could be improved. Finally, we note the need for domain-specific interfaces for visualization software. Domain-specific visualization software benefits from a narrow scope that allows the UI to focus on a particular type of visualization and adjust to the mindset of the users. General purpose visualization software benefits from a rich set of capabilities, but usually at the expense of a higher entry barrier.

Documentation aimed at both developers and users is critical for the longevity and success of all projects. Explanations of the design and functions of the software, as well as the intended use, need to be kept up to date. Manuals, tutorials, examples, and videos play a key role in providing accessible documentation to both communities. Incompatibilities between software libraries cause compile time and run time problems, so lists of dependencies must be maintained, including the version numbers of libraries. These challenges have persisted over the years, mainly as a function of the size and disjointed nature of the visualization software community. Due to the scope, nature, and complexity of visualization research and visualization software development, most efforts are devoted toward the functionality, whereas documentation and training are often given a lower priority. Thus, an opportunity exists to identify and define common designs and abstractions, software stacks, development, and deployment mechanisms to simplify the software for both the developers and users. Finally, it would be helpful to quantify the entry barriers for developers and users in order to determine whether suitable measures have been implemented. However, in our opinion, it is difficult to determine such a quantity in advance because it is highly dependent on the specific problem as well as the personal experience and pain threshold of the individual user. Thus, we envision a structured evaluation for quantifying the individual entry barrier in terms of a specific problem in a similar manner to the McGill Pain Questionnaire [37]. This questionnaire would

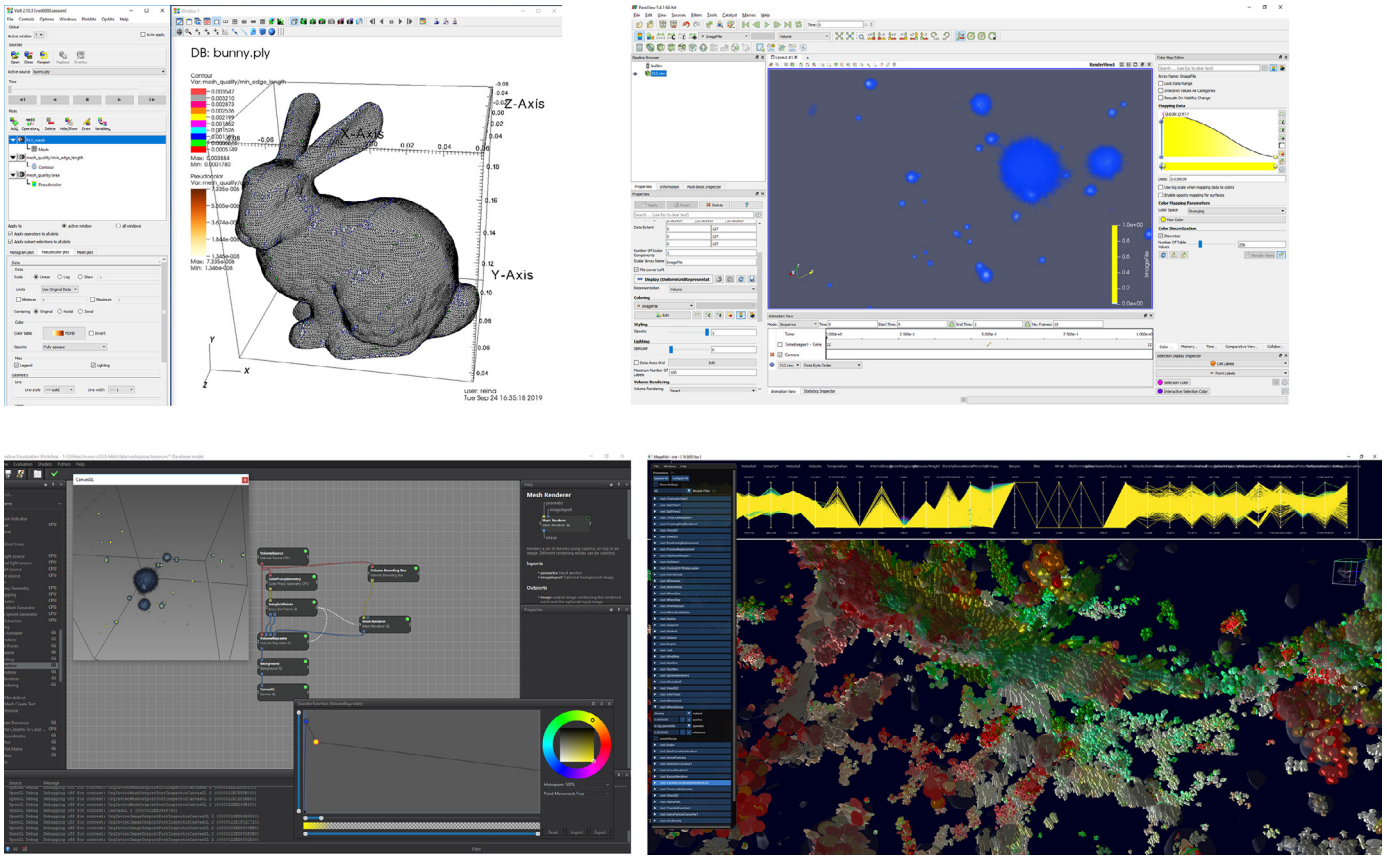

Fig. 3. Selection of different visualization systems, from left to right: VisIt, ParaView, Inviwo, and MegaMol. All have a complex interface and radically different interaction paradigms, so users must prepare their visualization in different ways.

allow us to relate the problem complexity, the time spent using software and reading documentation, and the results obtained (i.e., but using van Wijk's increase in knowledge ΔK [4]).

5. Building blocks and abstraction

Background and Motivation. Our community has generated many visualization software systems but no two systems are the same, although they often contain common parts. In particular, they typically share components related to user interaction, as well as specific visualization algorithms and rendering and windowing APIs. In some cases, the novel elements require new implementations. In other cases, the elements developed for one tool can be reused in another. We assume that this reuse would involve community software comprising building blocks. These building blocks may be developed by the first person who requires a capability and they would then be reused (and possibly improved) by others who need the same capability.

Our community has produced several successful types of visualization building blocks. These building blocks have been designed at different granularities, and they have been implemented as visualization toolkits with composable modules in some cases. Examples include AVS [38], D3 [39], ITK [40], SciRUN [41], VTK [17], and VTK-m [24]. In other cases, the building blocks comprise modules that solve one piece of a problem very well. These examples of toolkits are mostly for environments employed for rendering, e.g., OSPRay [42], or for graphics and interactions, e.g., Unity and Unreal Engine [43,44]. Furthermore, some environments provide mechanisms for combining modules, such as Jupyter Notebook [45], SciKit Learn [46], and Matlab [47]. Finally, our community has developed building block systems where the modules can be plugged together using domain-specific languages, such as Scout [48], Diderot [49], Vislang [50], and Vega Lite [51]. In addition, many applications provide building block systems via a GUI (or scripting), including examples developed by participants at our workshop such as FieldView [27], MegaMol [52], Inviwo [55], ParaView [53], and VisIt [19]. Despite these successes, it not clear how to design building block systems that satisfy the needs of most of our community.

Challenges. Our discussion of the specific challenges is organized into technical aspects related to system development, technical aspects related to system maintenance, and non-technical aspects.

Multiple technical barriers affect system development. First, the diverse set of factors involved includes many that may be in conflict, including the platform (Windows, Mac, and Linux), language (e.g., C, C++, Fortran, Python, Java, and Javascript), science domain that needs to be supported (e.g., biology, medicine, and physics), and data model. For example, a virtual reality application for molecular visualization requires a rendering infrastructure that can deliver a very high frame rate (e.g., 90 fps), whereas applications for handling very large data sets from physics simulations can often tolerate much lower frame rates (e.g., 10 fps). These different requirements explain why the respective communities have pursued different rendering strategies. However, common building blocks may exist that benefit both. A second related problem involves possible trade-offs. Principles such as flexibility and generality often increase reuse, which reduces the overall developer time (at least for the next developer), but these principles are often in conflict with performance goals. In communities where users expect a comprehensive functionality, sacrificing performance may be the only way to deliver a product

with an acceptable feature range, but this is not the case in other communities. The granularity of the building blocks is another characteristic related to trade-offs and performance. Building blocks can be designed along a granularity spectrum from fine to coarse. Thus, building blocks may perform very small tasks at a fine granularity, which favors reuse, or they may perform lengthy complicated tasks at a coarse granularity, which favors performance. A third technical consideration when developing a system is interoperability. The building blocks will need to work together to maximize the reuse potential of a system, which requires addressing difficult issues related to shared data models.

Technical barriers also affect system maintenance, and one specific concern is sufficient testing. This issue is of increasing importance if a developer community is growing and each developer writes only a small part of the code. Furthermore, the large variety of available hardware makes this more difficult because developers with a large software base may only have access to a subset of the hardware that needs to be tested, e.g., GPUs, distributed memory parallel systems, PowerWalls, and virtual reality devices. Thus, there is a possible risk that code will become fragile if many hardware features need to be supported. Another concern is the management of a large code base. For example, if all developers contribute their self-designed volume rendering code, then many modules choices will be available, with only marginal differences in some cases. Thus, a vibrant project may risk being damaged by its own success. A final concern is the future-proofing of projects, where systems must be able to evolve as requirements change over time or become obsolete.

Finally, we consider barriers that are not technical in nature. The first problem involves obtaining funding for initial development and for maintenance. Another issue is related to the creation of a community. A successful building block project must be equally attractive to developers and users. The building blocks should be easy to handle for users and also easy to incorporate into their workflows. Several concerns affect the developer side. One issue involves encouraging the participation of developers, i.e., there should be a low entry barrier to writing modules and contributing them. In particular, an appropriate reward model should incentivize individual developers. Participation will be discouraged if a developer contributes code, but only somebody else benefits. Furthermore, the software development process must be coordinated such as by moderation of the community, and developers must be encouraged to provide contributions that are crucial for the project's success, e.g., creating documentation.

Conclusion. Solving the challenges identified above would have many benefits. We discuss these benefits in the following in term of their costs, community value, and advancing science.

In terms of costs, the primary benefit is minimizing the time required to build new software for developers. A successful building blocks project will have additional cost-related benefits. For example, low cost software development may promote new approaches, such as the development of rapid prototypes or software intended for very small user groups. Furthermore, this model may enable access to better solutions (faster, with fewer errors, and more features) compared with the completely self-written solution, which is also more expensive (see Fig. 4 for an example of how the MegaMol framework has changed over a decade to leverage the improved open-source community). Finally, there is a cost savings benefit of developing expertise. Programmers can develop modules according to their own expertise with a building block model, which can then be used many times by non-experts. Subsequently, novices can exploit the building blocks made by experts to rapidly develop their own solutions.

There are also many community benefits, such as economy of scale, which is a different perspective to that discussed regarding costs. Another community benefit involves increased impact. If many people work together on a project, they often have contacts with a larger number of end users. Thus, the modules developed by one person are more likely to be used by many others. A third benefit is validation because the software will be used in more contexts, while the results will be viewed by more people, and the lifetime will probably be longer. A fourth benefit is the increased credibility of a larger software project, including the benefits for potential stakeholders because results are produced over a longer period of time. Finally, if a project is maintained by many people, the impact to the project caused by a community member leaving due to retirement, graduation, or a change of interests is reduced.

Two additional benefits are related to the advancement of science. The first is reproducibility because if algorithms are contributed to a public system, the results can be reproduced more easily. Reproducibility is currently a weak point in our community because many research results are difficult to replicate. The second benefit is improved utility where the building block design naturally leads to increasingly feature-rich software, which can then be applied to solve more complex problems. Furthermore, the usage of building blocks decreases the implementation effort and this can also lead to more sophisticated software because more time can be spent improving the user experience and accessibility for less experienced users.

6. One visualization system to fit all needs

Background and Motivation. An essential issue in visualization involves the basic software infrastructure that should be used for research and development. An approach with widespread success is employing a readily available framework such as Amira [54], Inviwo [55], MegaMol [52], MevisLab [56], ParaView [18], SCIRun [41], VolumeShop [57], Voreen [58], or VisIt [19].

Alternatively, researchers may develop a software infrastructure for their own group (and possibly collaborators) or only implement prototypes for individual PhD theses or even single research projects or publications. The successful use of VTK [17] demonstrates that agreeing on a powerful framework can have many benefits [59]. However, it can also be limiting because new research ideas, novel hardware, the specific details of new data modalities, or data sizes might not fit well into the existing code-base, structures, and approaches.

The primary goal of the project significantly determines whether to use an existing system or to develop a new one. This can involve either supporting a large user base or producing a novel algorithm for a publication. Using an existing system might impose technical restrictions or limit the developer's creativity. It is unlikely that a huge system will ever be fully accepted by the entire community. As a consequence, both developing new software and extending an existing system are used in practice.

Ideally, the visualization systems developed by different groups should work together in order to exploit the advantages of each instead of repeatedly implementing existing techniques. This would also facilitate the transition from a research prototype to a stable visualization production system. We focus on these interoperability aspects in the following.

Challenges. The interoperability of software systems is strongly influenced by the choice of programming language and corresponding infrastructure. In particular, it is more difficult to integrate different libraries, and components into C++-based frameworks compared to frameworks based on Python or JavaScript, which have widely distributed package managers and linking issues are avoided by design.

Direct interoperability between systems implemented using different programming languages is quite complex [60] if it is

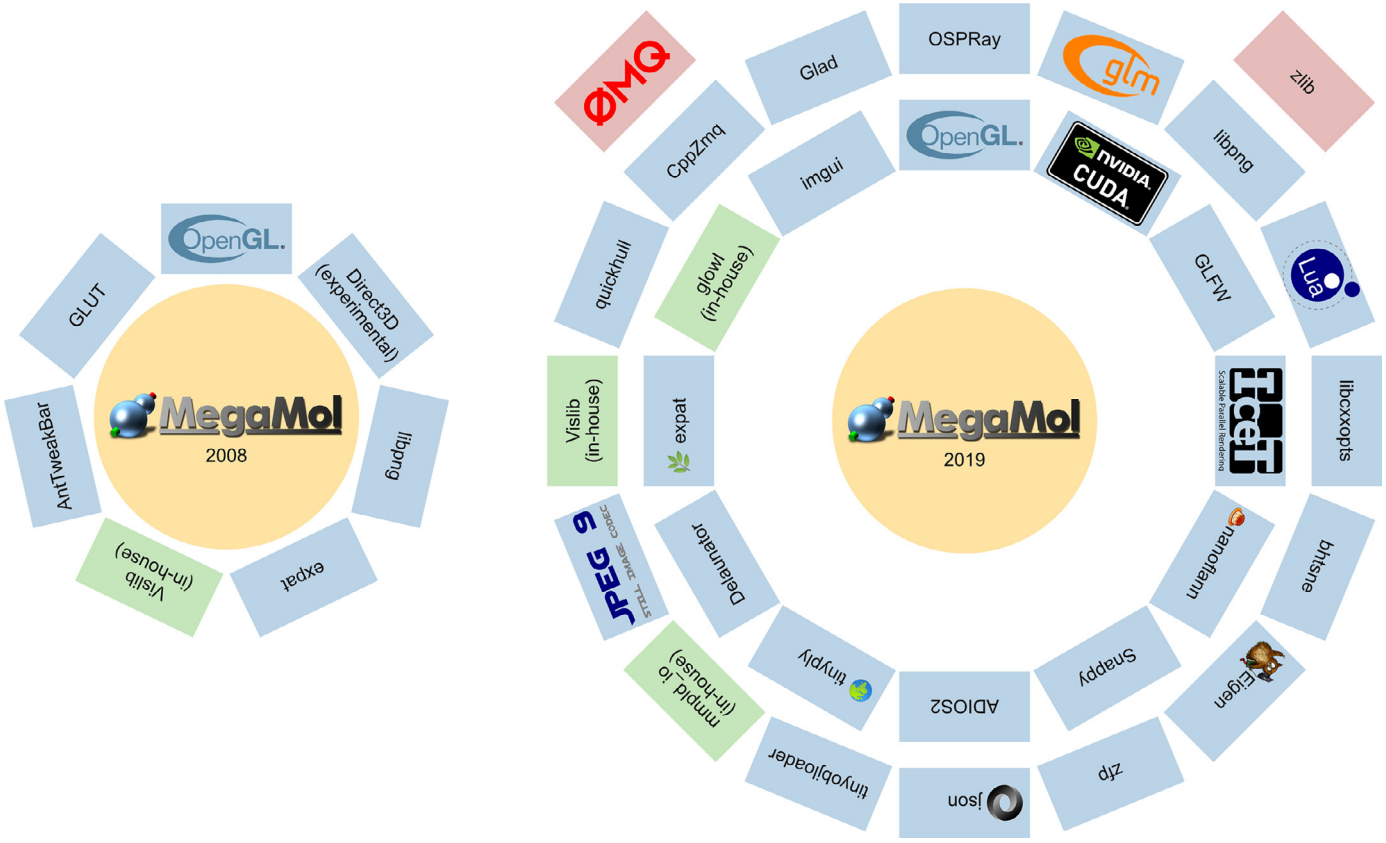

Fig. 4. Comparison of the external dependencies in MegaMol for its initial design in 2008 (left) and that in 2019 (right). Third-party projects are shown in blue, in-house projects in green, and dependencies of dependencies in red. It should be noted that the C++ cross-platform open source ecosystem did not provide many high quality building blocks a decade ago and much was developed in-house. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

possible at all. In these cases, the easiest way to achieve interoperability is to consider the underlying data model. Nevertheless, data can rarely be shared or exchanged directly between visualization systems. Some examples of existing data structures or formats are NumPy (multi-dimensional arrays) [61] and file formats such as NRRD [62] and the VTK [17] file format. File format containers such as Hdf5 [63], netCDF [64], and ADIOS [65] can be quite problematic because there are no semantic definitions of their contents and the code will rely on assumptions (or user interaction) to interpret the data.

Using a common data model also has several challenges. First, this approach could hinder creativity and reduce flexibility. Second, obtaining optimal performance usually requires an internal data model geared toward a specific use case and hardware setup. Thus, a problem arises regarding whether to invest the time for data conversion and how many times to perform this process. This can happen either each time the data set is loaded, which sacrifices the responsiveness, or once as a pre-processing step and storing the converted data to disk, which incurs storage demands.

In addition to these challenges, it is not clear whether only providing data models without any functionality is a viable approach. For instance, we assume that more people would use the VTK data models if they were not tightly bound to other VTK components but present in a self-contained and lightweight library instead. It is difficult for the entire visualization community to adopt a single data model but the availability of an excessive amount of idiosyncratic data models leads to a quadratic conversion problem between systems.

Conclusion. We believe that the costs of developing a single community-agreed system outweigh the benefits. In terms of the data model employed, less time is required to learn and utilize an existing system because it is often sufficient to understand the underlying data model. Furthermore, in the absence of a common data model, performance or storage issues usually occur because data must be frequently converted or stored in multiple formats. Therefore, providing a single, extensible, and community-agreed data model may be a successful approach. Thus, it is necessary to determine the different requirements of various visualization communities and a taxonomy of these communities would be helpful.

A viable short-term option may involve creating a bundled development environment that integrates important visualization libraries. Visualization developers could download this environment and configure the libraries that they wish to use in a similar manner to a package manager. This environment could be implemented in the popular build tool CMake [66] via the external project mechanism. The bundle would provide useful libraries for visualization, such as Hdf5 [63], netCDF [64], OSPRay [42], Inviwo [55], ITK [40], TTK [67], VTK(-m) [17,24], Kokkos [68], or Sensei [69]. This may appear to be a textbook example of containerization, but support for specialized hardware and features such as GPU acceleration or HPC systems using MPI communication is not yet reliable.

7. Game engines as data visualization platforms

Background and Motivation. For the last two decades, computer video games have been a primary driver of the development of GPU technology, which has been highly beneficial for the visualization community. During the same period, game developers have also developed game engines. These frameworks facilitate the development of games by making them significantly more efficient and feature-rich. Recently, some of these engines have been made

publicly available under varying licenses, such as Unity [43], Unreal Engine/Studio [44], and Unigine [70]. Other engines have also been developed as open source, e.g., Godot [71].

Unity and Unreal Engine, which are two of the major game engines, are both cross-platform, covering all platforms that are suitable for data visualization, i.e., Windows, MacOS, Linux, iOS, Android, and tvOS. They support a variety of interface environments (desktop, mobile, AR/VR/MR/XR, head mounted displays, touch, and multi-display clusters) and the necessary graphics APIs (Direct3D [72], OpenGL [20], Vulkan [21], and Metal [22]). Both engines support features such as high dynamic range rendering, asset management, animation systems, particle systems, cinematics, AI controllers, network and multi-player support, and marketplaces for developers and creative artists. Unity and Unreal Engine also differ in some respects. Unity is smaller and has fewer built-in features compared with Unreal Engine, which comes packed with out-of-the-box functionalities. Unreal can also be considered more artist friendly. It does not require traditional programming and offers the Blueprint visual style of programming logic, materials, and many other components such as AI behavior. Unity has a lower entry barrier for programmers through C# and the possibility of directly writing shaders in HLSL. Unreal Engine requires adherence to stricter C++ interfaces and architectural constructs for adding custom code and compute shaders, thereby making it more demanding for the developer.

These game engines are parts of entire ecosystems with tools to support the creative production pipeline. They support the "last mile" by providing an interactive and real-time graphics experience to end users, and they are currently being implemented in film and live broadcast production. The current convergence in production pipelines for real-time three-dimensional (3D) graphics and classical visual effects (VFX) for film is transformational in many ways, as follows.

- **Shared toolset:** Many of the tools for asset creation are suitable for use in both VFX and real-time graphics workflows, including model creation [73,74], material authoring [75,76], animation rigging, and motion capture [77]. Extensibility and interoperability are provided through plugins, which allow live-view feedback, real-time procedural generation, and variations of assets.
- **Real-time film production:** Game engines have been used for pre-visualization in VFX production to provide a preview of planned shots and scenes [78]. The quality of real-time graphics has increased to a level where it can be used for compositing in the final result [79], as well as live broadcasting from virtual studios, e.g., adding advanced visualizations of weather phenomena [80]. Live lighting and background effects can be generated to capture an appropriately illuminated live action character in real time [81,82].
- **Ray tracing:** The recent introductions of real-time ray tracing [83,84] and path tracing [85,86] have reduced the need for preprocessing and pre-baked lighting effects to obtain high-quality end results, thereby further decreasing the differences between VFX and real-time rendering pipelines. High fidelity support has also been incorporated due to demands from the automotive and manufacturing industries [87,88].

Many of the features described above are not typical direct data visualization techniques. However, they represent massive development efforts by a large community who have created a rich ecosystem of tools and platforms that could potentially be applied for exploratory and explanatory data visualization. Epic Games is also making significant efforts in the enterprise sector via Unreal Studio [89], where they are targeting industries beyond games such as architecture, automotive, media, and entertainment. Unreal Studio is essentially Unreal Engine with additional support for data exchange through Datasmith [90]. To further broaden the scope and use cases for Unreal Engine, Epic MegaGrants [91] was announced in March 2019 with funding of $100M. The call for proposals explicitly encourages non-game-related developments and it is even open to individuals.

Game engines have great potential for creating interactive applications with a variety of interface modalities (e.g., desktop, mobile, VR/AR/MR/XR, and touch), as well as high quality data visualization video content in multiple formats (e.g., two-dimensional (2D) screens, large/multiple screens, 360 video, and full dome video). It is an open question whether the data visualization community should join these massive efforts and contribute rather than creating their own visualization software universe.

Challenges. The modern game engine has been developed over the course of two decades, with the aim of delivering the highest performance and best visual quality at run-time. Thus, performance and shortcuts have been favored over readily comprehensible architectures and software concepts, thereby yielding complex and intricate software architectures, which are cognitively demanding and require much effort to understand. Therefore, an approach based on a game engine may take longer to explore compared with setting up a basic OpenGL application and exploring shader techniques. However, this approach can be rewarding for a PhD student in terms of learning outcomes. In addition, the workflow and typical work tasks in game development have received significant attention and they are quite efficient.

Generally speaking, levels and worlds are created by importing and assembling assets into a game engine editor where the application (or game) logic is implemented. Finally, the application is built and packaged for specific target platforms. In this context, a game engine follows a simple and well established process. By contrast, visualization applications do not package the data with the application. The data must be loaded into the system dynamically and via user interactions. In addition, native visualization data often do not exist as meshes and textures, but instead they are either used directly in the rendering process or employed to generate meshes and textures, or their combinations. User interactions typically cause the geometry and textures to be modified or regenerated in order to reflect user choices when exploring the data. Typical visualization applications generate most visual representations of the data in a dynamic and interactive manner. In games, most of the data usually have static representations, which may be animated but their structure or topology do not change entirely.

Thus, implementing and adding necessary functionalities is demanding and it requires a deeper understanding of the multithreaded rendering pipeline in order to efficiently generate dynamic geometry. Less material is available online such as documentation, tutorials, and examples to aid understanding, and to show how to implement appropriate visualization techniques. Nevertheless, efforts have been made at using game engines for data visualization purposes, particularly for VR/AR environments.

Some previous efforts at exploiting game engines were related to generic data and network visualizations. In particular, Kwon *et al.* [92] presented and examined methods for showing graphs in an immersive environment using HMDs and the Unreal Engine. The nodes and edges in the graph represented as points and splines are used in a compute shader to generate the renderable meshes in an interactive manner. The exploration of multivariate data in VR using HMDs was presented by Cordeil *et al.* [93] (a video [94] is available). All dimensions are accessible to the user who can select and connect them either by creating 2D or 3D scatter plots or link axes in a parallel coordinates plot. This allows the user to immersively explore a large number of data visualization configurations in an intuitive manner. The components from the ImAxes paper [93] were generalized and included into

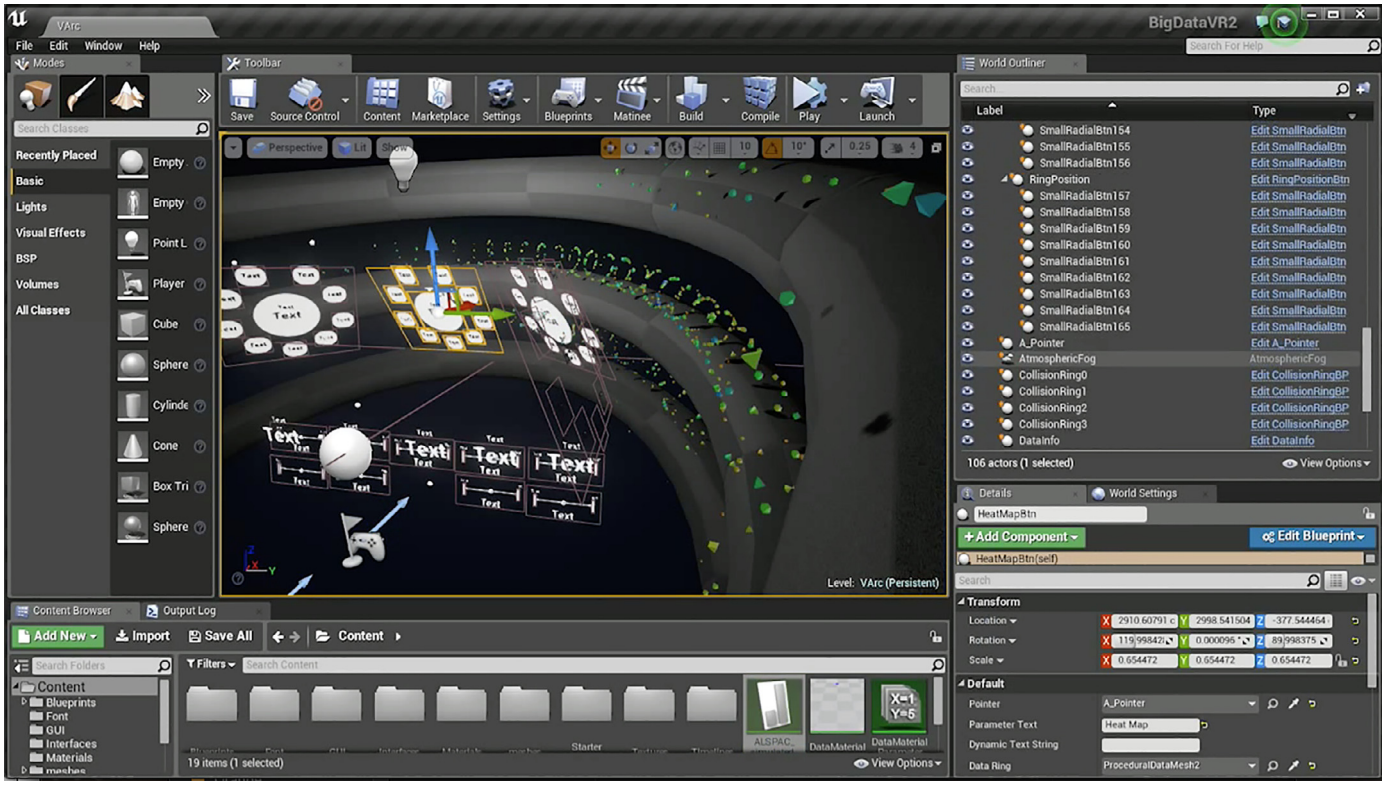

Fig. 5. Unreal Engine Editor showing a helical layout visualization of a large data set for the Big Data VR Challenge.

the Immersive Analytics Toolkit (IATK) [95] available for Unity. Another toolkit for Unity is DXR, which was introduced by Sicat *et al.* [96] and it aims to simplify the creation of data visualization applications for extended reality, including augmented and virtual reality. DXR provides a declarative visualization grammar inspired by Vega-Lite [51], so applications can be defined without dealing with low-level C# programming in Unity.

In 2015, Epic Games and the Wellcome Trust launched a Big Data VR Challenge [97] in order to attract developers to create visualization applications for VR using the Unreal Engine. The team called Masters of Pie was the winner and they showcased how data can be visualized [98]. A view of the project in the UE4 Editor is shown in Fig. 5.

Game engines have been used in other domains such as biochemistry, molecular visualization, and drug discovery. UnityMol [99] is an open source library for use with Unity, which provides molecular viewing and a prototyping platform. Kingsley *et al.* [100] presented a system for collaborative drug discovery based on the Unity platform. Other examples of the use of the Unreal Engine can be seen in YouTube videos [101,102].

Conclusion. Game engines have been used in several scientific domains, but particularly in biochemistry and immersive data visualization. Extending their use in further domains requires additional work in terms of data processing and management as well as the dynamic and interactive creation of visual representations. Game engines provide tools to allow the simple assembly of applications in game development. Their possible benefits for data visualization could be quite transformative. Useful content for public dissemination and science communication could be generated with the data visualization tools available in game engines, including videos and interactive applications, together with compelling storytelling.

The following actions are necessary to promote and facilitate the adoption of game engines for data visualization.

- Generate a collection of tutorials outlining the principles for data management and the dynamic creation of visual representations within game engines.
- Produce examples that can be shared and used for learning and the creation of new applications and examples.
- Share best practices for typical visualization tasks on online forums.
- Create sets of reusable tools and plugins that can be shared in the available marketplaces.

The effective use of game engines for scientific visualization has been demonstrated by several successful examples of integrating VTK and Unity [103,104] as well as the Unreal Engine [105]. Producing visualizations using a game engine also allows us to exploit the functionalities that the engines already provide in order to create rich and intriguing visual storytelling around data visualization for experts or the general public.

8. Medium-term funding and sustainability

Background and Motivation. The development, extension, and maintenance of sustainable state-of-the-art visualization software requires skilled personnel. Substantial financial resources and long-term commitments are required to achieve the level of stability and quality that distinguishes usable software from research prototypes.

These efforts are of great interest to both the visualization community and domain users, where the former require re-usable visualization libraries that enable fast prototyping to accelerate innovation and the latter need novel visualization solutions that solve actual problems or accelerate insight processes. Depending on the type of software and intended application domain, different strategies can be employed to convert an initial prototype into a valuable and long-lasting tool with high impact, and acquire sufficient funding for its development. Reniers et al. [106] presented

an iterative process that evolves a research prototype to an actual product and demonstrated a case study based on the visual analytics tool Solid*. The lessons that they learned can provide valuable guidance throughout the process from an engineering viewpoint. However, substantial financial resources are still required.

In most public basic research funding schemes, particularly ICT research, sustainable software development is not considered research, and thus it is not eligible for funding. However, we are currently witnessing an increase in the awareness of the value of research software and it is affecting the funding landscape. The National Science Foundation (NSF) in the USA is becoming more supportive of community software. The National Institutes of Health (NIH) invest even more resources into software development than the NSF. In Europe, several initiatives have focused on funding software sustainability, including the Wellcome Trust and Software Sustainability Institute in the UK, DFG in Germany, and H2020 projects in the EU.

Another opportunity arises when domain users identify software as new enabling technology for their research and start to invest into its development. An example is the inclusion of third-party contracts for software development in the funding applications of domain experts.

Another possible avenue involves founding a company dedicated to sustaining a research software package, which can be facilitated by seed funding opportunities similar to those offered by the US Small Business Seed Funding and the SME Instrument of H2020 in the EU, or by venture capital firms. These can be employed to transform a prototype into usable software that is ready for commercialization. Examples of previous successful startups include Trifacta, Tableau Software, and Amira. Conversely, Kitware has structured their business model around supporting open source software (ParaView, CMake, and others) with a clear research focus. However, for most types of visualization research software with very specific functionalities and a narrow focus, the expected customer base and thus the revenue is often not sufficiently large to attempt this step.

In rare cases, a company might be interested in pushing their technology into the research community and allocate funding. An example is Intel supporting the integration of their open-source ray tracer OSPRay into visualization frameworks such as MegaMol, VisIt, and ParaView.

Challenges. Most research funding sources typically do not make funding decisions or evaluate projects based on the quality of the software produced, which poses a problem when trying to finance software derived from research projects. The funding agencies are primarily interested in the advancement of science, which is usually measured in terms of publications. However, experts from the application domain expect a usable tool as an intrinsic part of a collaboration. Dedicating a sizeable amount of resources to produce a reliable tool may reduce the quality and quantity of the scientific output. Reaching a balance between meeting the needs of collaborators and the requirements of funding agencies is a challenge. PhD students are generally more inclined to maximize their scientific results to further their career, which is also in the interest of the funding agencies. However, this attitude can be detrimental to a project because the resulting software will be far from useful and the collaborators might lose interest.

Intellectual property is another challenge related to funding. In particular, intellectual property can become an issue when software starts to be useful to a broader audience. Disagreements concerning utilization and/or commercialization among the involved parties can prevent the dissemination of software. Thus, all rights and responsibilities among the parties involved must be clarified in detail before an investment is made. In the best case, the legal set-up is defined and agreed upon at the start of a software development project. Conversely, funding agencies, such as the German Research Foundation, often strongly encourage or even require open-source software to facilitate the re-usage of publicly funded software by other researchers. The choice of the licensing scheme under which software is published should be aligned with potential commercialization plans or targeted user groups. Using permissive software licenses, such as MIT, BSD or Apache, or combining a dual licensing scheme such as a copyleft license with a commercial license can definitely ease the commercialization process. However, if software is compiled using other software libraries published under a mixture of licenses, then this model might not be applicable without violating one of the licenses. Providing software under an open source license and patenting the invention at the same time is possible and this is a viable approach for protecting the usage of the invention independent of the published code depending on the license selected.

Conclusion. Software development requires time and this time incurs costs. Thus, the visualization community cannot grow without financial support to transform research ideas into usable software. The funding agencies and stakeholders that invest in research and innovation should become more aware that it is cheaper to support quality software development than to pay for repeatedly replicated programming in the long run. Visualization is predominantly an enabling technology for scientific discovery. Funding sources should realize that including support for software development can have far-reaching impacts on many scientific domains.

Given the lack of sufficient consideration at present from funding agencies, scientists need to find alternative sources for financially supporting software development and maintenance. Researchers need to factor these costs into their budget estimations in project proposals, but this will make it hard to compete with proposals that do not. A viable option is to involve undergraduate students in the software development process, although their limited experience can lead to lower quality software. Here, another obstacle to effective sustainability is that students usually participate in a project only for a short time.

At present, software that is adopted by a community is considered to have a good chance of becoming sustainable. The community can loosely collaborate and collectively contribute to software development. The development cost is then distributed among the members, which can make the support requirements more manageable. A vibrant community can induce reinforcing circular effects, where the software makes the research more productive, research adds functionalities to the software, the added functionalities increase the impact of the software, and the added impact helps those involved with securing new funding for the next research project. Building up a community is a difficult task, which requires a lot of time and commitment, and it is still not guaranteed to succeed. Prior research is necessary to estimate the size of the potential community. A community will typically only work for software that is either open source or otherwise extensible, or that offers free academic developer licenses.

9. Evaluation of software and quantification of success

Background and Motivation. At present, it is difficult to convince reviewers and readers of a scientific paper that a novel visualization technique or tool is efficient and effective without some type of evaluation. Various types of evaluation can be conducted depending on the object evaluated. Evaluating a visualization technique is quite different from evaluating a visualization tool or a visualization software library. We have considerable experience of evaluating techniques and visualization tools, but less experience of evaluating visualization development libraries.

Evaluation practices in our field have been reviewed in at least three surveys. Lam et al. [107] introduce seven evaluation scenarios used in information visualization derived from more than 800 papers. Isenberg et al. [108] built on these scenarios and described the current status and historic development of evaluation practices based on 581 papers published at the IEEE Conference on Scientific Visualization (IEEE VIS SciVis). The vast majority of the papers considered in both surveys dealt with evaluations of visualization techniques. Preim et al. provided a critical discussion of the evaluation practices employed for medical visualization [109], which are also useful for the whole visualization community. Finally, a group of visualization researchers with an interest in evaluation organizes the Biennial Workshop on Evaluation in Visualization (BELIV) at the IEEE VIS conference.

Challenges. The majority of user studies in visualization, especially evaluations of new visualization techniques, were performed in laboratory settings based on strictly defined questions and in a limited period of time. This type of set-up is certainly necessary but it is not always favorable for an exploratory evaluation, which is the typical use case for visualization software. In some cases, more formal evaluations were conducted outside predefined and strictly controlled laboratory settings. In particular, Seo et al. [110] reported the development, refinement, and evaluation of a tool called Hierarchical Clustering Explorer. By February 2005, this tool had been downloaded about 2500 times. The authors identified six publications by other scientists who regarded this tool helpful for their further research. The authors then conducted a long-term evaluation based on a six-week study with six participants from different domains. They also conducted an email survey of more than 50 software users. Undertaking such an evaluation is certainly a great effort, but the feedback obtained is extremely valuable. A similar long-term study is not possible for every visualization tool, but it should be considered for software tools with many users. Shneiderman and Plaisant identified long-term case studies as promising tools for empirical evaluation because control studies tend to distance themselves from practical problems and real-world problems [111].

The evaluation of visualization software or algorithms generally involves many issues regarding what to test and how to do it. For example, (formalized) interviews are a simple approach for evaluating user satisfaction. Support requests from users also provide additional feedback.

For open source solutions, the number of downloads and the size of the community can be employed as representative metrics. In 2017, the IEEE VIS conference introduced the *Test of Time* award to recognize "articles published at previous conferences whose contents are still vibrant and useful today and have had a major impact and influence within and beyond the visualization community" (papers that were published 10, 15, 20, and 25 years ago are considered for the award). Most of the Test of Time award papers described visualization techniques. The Jigsaw system [112] received the award in 2017. This software has been used in many fields (e.g., visualization, text analysis, journalism, law enforcement, and finance) and it was awarded because of its high impact. The ManyEyes system [113] was awarded because of its visionary role in bringing visualization to common users.

All of the measures mentioned above (satisfaction, user community size, awards, etc.) are indicators of the significance and importance of visualization software. In particular, we want to estimate the value of visualization research based on an evaluation. Van Wijk [4] defined a model of a visualization process by quantifying its value from technological and economic viewpoints. The technological evaluation measures the visualization based on its effectiveness and efficiency. The economic evaluation focuses on profitability starting from the initial knowledge of the user and then measuring how much effort (and thus cost) is needed to gain additional knowledge (profit). He also discussed Line Integral Convolution (LIC) for vector fields as an interesting example that was well received by the visualization community but not used frequently in application domains. New visualization methods such as LIC do contribute to a researchers' output (papers, citations, etc.) but computational fluid dynamics engineers may consider these methods as excessively complex, i.e., the investment is excessively high given the perceived increase in knowledge. However, an increase in knowledge is only one measure of success and it is highly dependent on the user group and the development goal of the software.

The cost model can also be used to justify development decisions regarding the features and visualization methods included in a specific tool. The selection of the basic underlying framework directly influences the cost of implementing the features.

A PhD student or scientist starting a new research project must consider the costs of either implementing a new prototype or building upon an existing visualization framework. The first choice incurs direct costs and additional costs for maintaining or rebuilding the software during the project. The second choice requires investments in becoming familiar with existing code, and potentially multiple times if the decision was sub-optimal and further options must be considered. These costs can be estimated using models from software engineering but this is a non-trivial, data-intensive [114], and error-prone [115] process that requires significant experience.

PhD students comprise a large proportion of the readership of visualization articles and they are directly interested in visualization software that they might use in their research. In the various subfields of visualization, students may be familiar with popular choices such as VTK [17] or D3 [39], but many more visualization frameworks and libraries are available. However, to the best of our knowledge, a broad and comprehensive evaluation and comparison of visualization development frameworks has not been conducted. This problem is not unique to the visualization community. Powell et al. [116] described a practical strategy for evaluating software tools in industry. Based on their own experience of selecting software tools, they identified eight relevant issues that should be considered when choosing a software library.

Conclusion. After more than half a century of visualization research [117], we seem to know how to evaluate individual visualization techniques as well as more complex exploratory tools. We have learned that the execution of user studies is not trivial, and thus we need to team up with experts. The success of commercially and freely available visualization tools based on visualization research (see Section 6 for many examples) indicates that our research is relevant to industry and the user community. We also have good guidelines about how to write visualization papers [118] or how to perform a design study in close collaboration with domain experts [119,120]. However, it seems that we need similar, systematic guidelines for evaluating visualization development tools. Clearly, one of the first steps required to address this issue is conducting a qualitative classification of existing software, as discussed in the next section.

10. Building a typology of visualization software

Background and Motivation. According to Forward and Lethbridge [121], a taxonomy of software types provides a context for empirical results, facilitates the reuse of artifacts, increases the use of models and frameworks, and helps educators to build courses. In the visualization field, we envisage similar benefits such as improved communication between researchers, developers, students, and end users due to the availability of a formal description of the

problem space. This formal description can be employed to make informed decisions regarding the software to use for a particular problem and how to build a database of visualization software, which could serve as the basis for developing a comprehensive visualization software online browser. Online browsers are already used for surveys of research prototypes in specific visualization sub-fields [122]. A visualization software database would represent the space of existing solutions but also the space of open problems, i.e., "white spots" for new research.

There are many state-of-the-art reports that focused on specific visualization topics, such as text visualization techniques [123], visualization of high-dimensional data [124], molecular visualization [125], and dynamic graph visualization [126]. In the information visualization field, recent surveys have identified the design spaces of existing visualization construction tools [127,128]. An overview of surveys was presented by McNabb and Laramee [122].

However, we lack a formal description of the design space of existing visualization software across research fields and application domains. Hierarchical software-type taxonomies have also been proposed [121]. In addition, the software engineering community has assembled a body of knowledge (SWEBOK [129]) based on the ISO/IEC norm for software life cycle processes [130], which can serve as a common basis for any taxonomy of more specific software. To facilitate the development of rich descriptions of visualization software from multiple perspectives, we consider that it is more suitable to create a multidimensional typology [131] that allows categories not to be mutually exclusive [132]. Such a typology would provide a helpful basis for communication across a broad range of stakeholders, including visualization researchers, designers, software developers, computer science students, and domain experts from a wide range of scientific and nonscientific domains. The requisite typology also needs to encompass visualization software at all maturity levels ranging from highly specialized research prototypes to powerful multi-purpose frameworks.

Challenges. A major challenge when constructing a practical visualization software typology is selecting appropriate details and abstraction levels. SWEBOK is highly accurate but its application to actual software yields a table that is overwhelming to most given the hundreds of topics it contains. A visualization software typology should allow the reader to capture all of the distinctive features that are specific and appropriate for visualization software, but it should still be sufficiently compact to be usable and comprehensible in practice. These requirements raise several questions, as follows. Should it only describe multi-purpose visualization software or should it also include highly specialized prototypes that might even build upon other visualization frameworks? Should it only include systems for scientific visualization or be more inclusive? What perspectives should be considered for users of the typology? An end user needs different information compared with a software developer. In order to build a successful and frequently used typology, the level should be not too shallow, i.e., excessively trivial, or deep. An excessive amount of detail will prevent simple comparisons and users would be reluctant to classify their own framework because of the effort required. Ideally, a visualization software typology should focus on specific details of the visualization field, as well as covering general software development aspects but only to a degree that is relevant to the users. These requirements are different for actual end users and developers (visualization researchers). Another challenge is the typology development process comprising multiple steps, such as the development of relevant dimensions and grouping of cases [133]. This construction process requires in-depth theoretical knowledge about the field from multiple perspectives. Therefore, inputs are required from multiple experts as well as iterative reviews and edits to select meaningful types [121]. Finally, a typology must be empirically tested to confirm its validity [134].

Conclusion. Constructing a visualization software typology that is useful to a variety of stakeholders is not a trivial task, and it requires much knowledge and experience. An open seminar or workshop such as Shonan or Dagstuhl provides a valuable setting for conducting this task. Therefore, at our Shonan meeting, we conducted the initial stages of a visualization software typology development process. In a brainstorming session with 12 participants, we first established a list of 90 aspects for describing visualization software. In smaller groups, we then obtained parallel first classifications of these aspects, which were then merged and refined into six high-level categories and their sub-categories. The overall consensus is illustrated in Fig. 6. The resulting shallow categorization represents a foundation for developing a more detailed typology for expressively characterizing a wide range of visualization software.

After constructing a typology, the next step involves verification and demonstrating usefulness based on its successful application to a set of real-world examples. However, the major challenge is communicating the typology to the target audience to ensure that it is actually used. With strong support from the community, it will be possible to build a substantial database of existing software systems, which can have a considerable impact by serving as a standard repository of available visualization software systems. This would be especially useful as a guide for new PhD students who need to decide the system for building their research upon.

11. Discussion and conclusion

We identified a number of difficult challenges that the visualization community must address in terms of the development and maintenance of software as well as the significance of these challenges. Some of these challenges can be considered opportunities whereas others seem to have solutions in an idealized case but they will be difficult to address under realistic conditions, at least at present. In the following, we provide some general advice for the visualization community.

- Investing in user experience reduces user support and helps with the adoption of the software. This is the first step toward developing sustainable software through an active community. A good user experience also facilitates the collection of results and evaluating software.
- We consider that adopting technological advances and disruptive change early on will provide new research opportunities. Promising research avenues are more likely to be identified and new approaches may emerge to old problems. A stronger social community that shares the pains and gains of experimentation would help to outweigh the risks of investing in new technology, as well as facilitating the integration of promising technology into existing frameworks to make it easier for the overall community to move forward.
- Even in a research project, best practices from software engineering should be followed. Good and up-to-date documentation ensures and improves the usability for users and developers. Modularizing source code into self-contained building blocks facilitates re-use, as well as fostering comparability and reproducibility. Modularization also allows for the non-disruptive integration of disruptive technology. VTK-m is a good example because it supersedes the existing VTK functionalities by providing platform-specific parallelized implementations that can utilize accelerators. Calls to legacy VTK can then be selectively replaced by calls to VTK-m, thereby minimizing the cost of modernizing software based on VTK (such as ParaView).

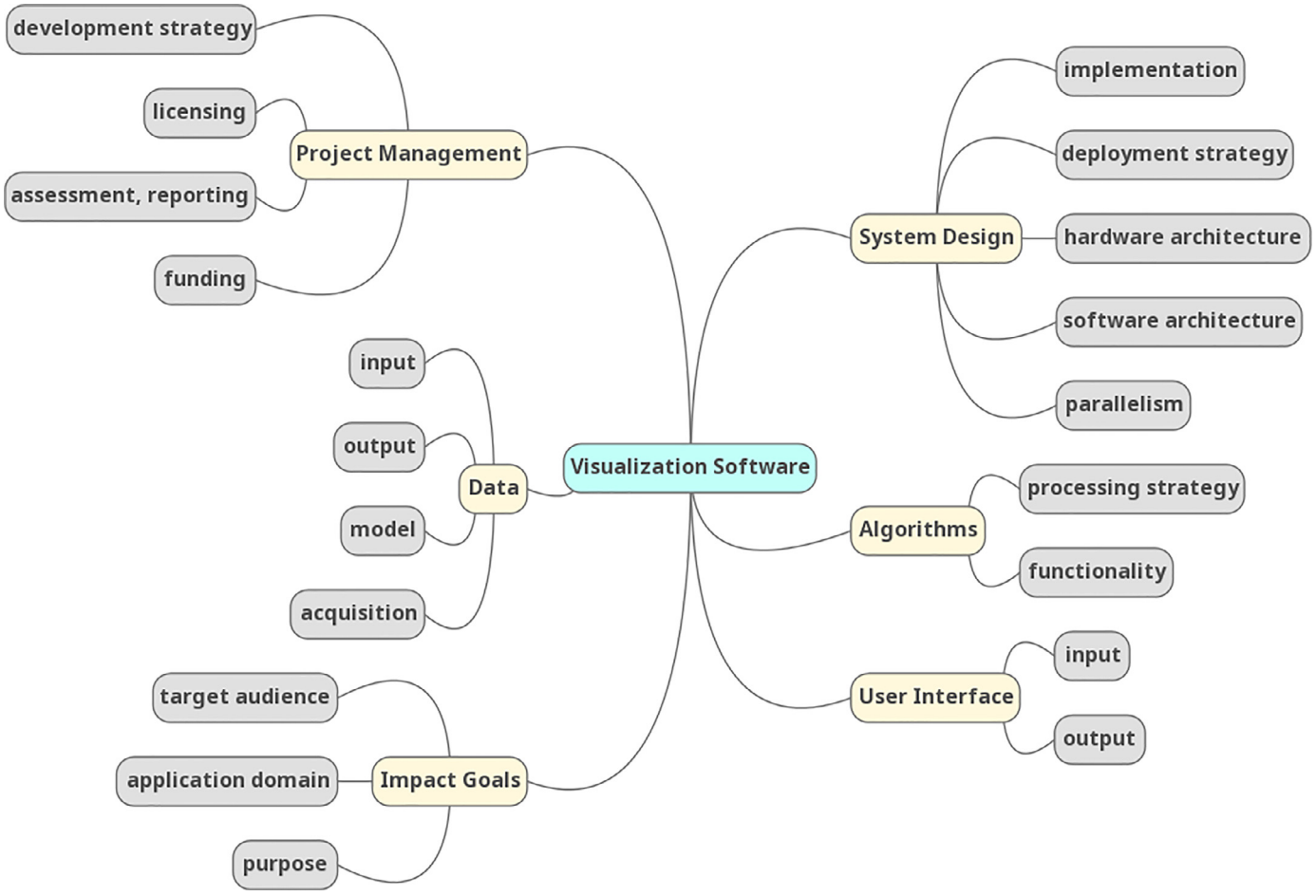

Fig. 6. Topmost levels of the visualization software typology agreed upon during Shonan seminar 145.

- Game Engines can provide a viable basis for visualization development, especially for data set sizes that are GPU-friendly.
- How to fund the development and maintenance of software should be considered early in a research project. The need for and the possible success of a planned software system should be determined based on at least a simplified "market analysis," which will influence the software design and determine whether the system should be build from scratch.
- As mentioned in the introduction, research groups that use a common development platform seem to be more productive. Therefore, based on our own experience, we reckon that one-shot prototypes are wasteful in the long run. However, it is clear that a certain minimum group size is necessary to successfully develop and maintain a visualization framework. Choosing between a self-designed framework and extending an existing framework is also a nontrivial decision, as discussed in Sections 6 and 2.

We also identified the following steps that will make us more productive as a community.

- First, we require more published and open source code. Reproducibility is severely hampered without readily available code. The perceived lower code quality of a research prototype is no excuse for not publishing it. Repeatedly implementing a published technique for comparative purposes is impractical, and it makes objective and faithful replication questionable. An interesting initiative has been the VisImp project with the aim of eventually implementing all basic visualization techniques in VTK [135], but this project appears to have been discontinued (http://visimp.org/ is no longer available). Another helpful activity is the EuroRV[3] workshop (now called TrustVis), which has the aims of ensuring reproducibility, verification, and validation in visualization. Two approaches might improve the situation, as follows. One possibility is introducing artifact tracks at our conferences, as employed in other disciplines and even other areas of computer science. A second possibility is making the submission of code and data a mandatory part of the publication process to ensure full reproducibility, which is also a common practice in other disciplines. Organizations such as IEEE appear to have the required infrastructure in place. In addition, the quality of the code provided will influence citations and the uptake of approaches in the long run, thereby helping to differentiate more valuable contributions.
- We consider visualization as an enabling science because our goal is to help other scientists or even laymen to make discoveries and be more productive. However, application papers are usually considered "weaker" contributions compared with technique papers. We argue that the application of a new visualization tool to solve a significant problem (even if it does not introduce a novel technique) should yield a significant publication. Visualization scientists should not need excuses or have to allocate personal time to producing tools that are useful to other scientists. We cannot wait for people to commercialize every single useful finding or make business plans and judge risks when the time would sometimes be better invested actually implementing the software.
- The previous point is closely linked with the policies of funding agencies. The situation is slowly improving (see Section 8)

but in typical research proposals, we still have to hide the time required to develop and hone software within the time budgeted for research. If software was integral to a publication or we had specific venues for publishing our tools, then the development time would also be integral to research proposals.

Despite our in-depth discussion of sustainable visualization software development, it is important to note that preliminary prototypes are also valuable. As mentioned in Section 9 and Section 10, researchers must work with limited information and foresight, which will sometimes lead to sub-optimal or incorrect choices, e.g., a library might prove unsuitable or short-lived, and a framework might not meet expectations. This risk can be reduced by prototyping. Throwaway Prototyping [136] typically gives a good indication of whether a particular approach is worthy of more thorough and systematic investigation. It can also be applied in the decision process concerning libraries and frameworks. In general, Throwaway Prototyping cannot replace stable and sustainable software development because the quality of the code will degrade rapidly during and after the prototyping phase. However, while Throwaway Prototyping might often seem like an expense that is not affordable, it is worth noting that an appropriately engineered prototype that can be refactored into stable code is even more expensive to develop.

Another central question is the quantification of success relative to the time spent developing software and the "conventional" research output, i.e., the number of publications. We should focus on the development of sustainable software because we consider that this strategy leads to increased research outputs in the long run. In terms of the time spent, the following inequation must be valid in order to justify framework development:

$$t(\theta_\pi) \geq t(\theta_\phi), \quad \text{where} \tag{1}$$
$$t(\theta_\pi) = t(\theta) + t(\pi)$$
$$t(\theta_\phi) = t(\theta) + \frac{t(\mu)}{|\Delta|} + \frac{t(\phi)}{|\Psi|}.$$

Thus, the time t required to implement a technique θ within paper-specific prototype π must be larger than or equal to the time required to extend an existing framework ϕ with the same technique. We hypothesize that implementing a technique has a basic time requirement $t(\theta)$, which is constant. A prototype that includes this technique has some overheads, such as data set loading, which requires additional time $t(\pi)$. When using a framework, we assume that this overhead functionality is already provided. Thus, we factor in the development time of the framework $t(\phi)$ as well as the maintenance time $t(\mu)$. However, $t(\phi)$ can be amortized over the set of all papers Ψ that use the framework, thereby reducing its cost in a proportional manner. Similarly, when a number of people use the framework, the maintenance effort is split among the contributors Δ, which reduces the proportion per paper. Clearly, building a framework has a one-time cost $t(\phi)$ that becomes negligible if many papers use the same framework. The problematic factor is the maintenance cost, which is very difficult to estimate because it depends on the initial design decisions, the state of the ecosystem at the time (see Section 2), and the overall software quality (see Sections 3 and 5). Furthermore, the number of contributors is affected by the framework's funding, sustainability (see Section 8), and community uptake (see Section 4). We have no objective proof but we argue that groups who maintain some sort of common platform or tool for visualization research also have an increased research output. We note that the availability of a common framework within a research group still does not equate to its availability or relevance outside that group. Our own experience suggests that developing and maintaining a framework is worth the effort at least in the medium term: $t(\mu) \ll t(\phi)$. However, if software is used for a decade or more, the changes in the ecosystem alone might require major revisions to the initial design, thereby increasing the maintenance cost at this stage. In addition to this economy of time, using a visualization framework has additional fundamental advantages because it is very difficult to build complex analysis systems for specific use-cases within a prototype. One example is the annual IEEE SciVis contest, which involves tackling complex data and multi-faceted problems. In the last 10 years, only three winning submissions have used from-scratch prototypes (2011, 2013 and 2014), whereas all the others were built on existing visualization frameworks (MegaMol [52]: 2012, 2016, 2019; Voreen [58]: 2010, 2015, 2018; VTK [17]: 2017). Another benefit of using an existing framework is the comparability with previous approaches.

Finally, we must emphasize that visualization software is an important and indispensable pillar of our community. It helps the visualization researcher who is also a developer, and the domain scientist who requires software that is usable in practice. High-quality open-source software accelerates the development of new and improved visualization methods, and it ensures reproducibility and re-usability. It makes evaluating and discussing scientific results much easier and more rapid. At present, not all requirements are satisfied by one approach (see Section 6). Thus, establishing a usable typology (see Section 10) is valuable for addressing a significant research need. This typology will be useful for researchers starting a new visualization project.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Guido Reina: Conceptualization, Investigation, Writing - original draft, Writing - review & editing, Supervision. **Hank Childs:** Conceptualization, Investigation, Writing - original draft, Writing - review & editing. **Krešimir Matković:** Investigation, Writing - original draft, Writing - review & editing. **Katja Bühler:** Investigation, Writing - original draft, Writing - review & editing. **Manuela Waldner:** Investigation, Writing - original draft, Writing - review & editing. **David Pugmire:** Investigation, Writing - original draft, Writing - review & editing. **Barbora Kozlíková:** Investigation, Writing - original draft, Writing - review & editing. **Timo Ropinski:** Investigation, Writing - original draft, Writing - review & editing. **Patric Ljung:** Investigation, Writing - original draft, Writing - review & editing. **Takayuki Itoh:** Conceptualization, Investigation, Writing - review & editing. **Eduard Gröller:** Investigation, Writing - review & editing. **Michael Krone:** Conceptualization, Investigation, Writing - original draft, Writing - review & editing, Supervision.

Acknowledgments

We thank the National Institute of Informatics, Japan for giving us the opportunity to organize a Shonan Seminar. We also thank all the participants at Shonan Meeting No. 145 for contributing the ideas, which we have summarized in this report. In addition to the authors of this manuscript, the participants were: David DeMarle, Kitware; Johannes Günther, Intel Corporation; Markus Hadwiger, King Abdullah University of Science & Technology; Yun Jang, Sejong University; Shintaro Kawahara, JAMSTEC; Steve Legensky, Intelligent Light; Xavier Martinez, IBPC/CNRS; Kenneth Moreland, Sandia National Labs; Valerio Pascucci, University of Utah; Allen Sanderson, SCI/University of Utah; Claudio Silva, New York University; Ivan Viola, King Abdullah University of Science & Technology; Xiaoru Yuan, Peking University. This work was

partially supported by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as well as the Czech Science Foundation (GAČR, Grantová agentura České republiky) within DFG project B05 of the collaborative research center SFB 1244 (project ID 279064222), DFG-GAČR project PROLINT (project IDs 391088465 / GC18-18647J), as well as DFG research software sustainability projects for Inviwo (project ID 391107954) and MegaMol (project ID 391302154). It was additionally supported by Intel®Corporation via the Intel®Graphics and Visualization Institutes of XeLLENCE program (CG #35512501). Michael Krone was funded by the Carl Zeiss Foundation. VRVis is funded by BMVIT, BMDW, Styria, SFG and Vienna Business Agency in the scope of COMET – Competence Centers for Excellent Technologies (854174), which is managed by FFG. Finally, we thank the anonymous reviewers for their very detailed feedback and additional insights.

References

[1] Munznery T. Visualization analysis and design. AK Peters visualization series. CRC Press; 2015. ISBN 9781498759717. https://books.google.de/books?id=NfkYCwAAQBAJ.
[2] Chen M, Jaenicke H. An information-theoretic framework for visualization. IEEE Trans Vis Comput Graph 2010;16(6):1206–15. doi:10.1109/TVCG.2010.132.
[3] Chen M, Ebert DS. An ontological framework for supporting the design and evaluation of visual analytics systems. Comput Graph Forum 2019;38(3):131–44. doi:10.1111/cgf.13677.
[4] van Wijk JJ. The value of visualization. In: IEEE visualization; 2005. p. 79–86. doi:10.1109/VISUAL.2005.1532781.
[5] Moreland K. Oh, $#*@! exascale! the effect of emerging architectures on scientific discovery. In: 2012 SC companion: high performance computing, networking storage and analysis; 2012. p. 224–31. doi:10.1109/SC.Companion.2012.38.
[6] Moreland K, Larsen M, Childs H. Visualization for exascale: portable performance is critical. Supercomput Front Innov 2015;2(3). https://superfri.org/superfri/article/view/77.
[7] Shonan seminar 145: The moving target of visualization software for an ever more complex world. 2019. Accessed: 2019-06-11 https://shonan.nii.ac.jp/seminars/145/.
[8] of Salisbury J.. The metalogicon book 3. 1159.
[9] Lorensen WE. On the death of visualization. In: NIH/NSF Proc. Fall 2004 workshop visualization research challenges; 2004.
[10] Haber RB, McNabb DA. Visualization idioms: a conceptual model for scientific visualization systems. Vis Sci Comput 1990;74:74–93.
[11] Kitchin R, McArdle G. What makes Big Data, Big Data? exploring the ontological characteristics of 26 datasets. Big Data Soc 2016:1–10.
[12] Bauer AC, Abbasi H, Ahrens J, Childs H, Geveci B, Klasky S, et al. In situ methods, infrastructures, and applications on high performance computing platforms. Comput Graphics Forum 2016. doi:10.1111/cgf.12930.
[13] Davis AM. 201 Principles of software development. McGraw-Hill, Inc.; 1995.
[14] Lehman MM, Ramil JF, Wernick PD, Perry DE, Turski WM. Metrics and laws of software evolution-the nineties view. In: Proceedings fourth international software metrics symposium. IEEE; 1997. p. 20–32.
[15] Humphrey W, Dalke A, Schulten K. VMD: visual molecular dynamics. J Mol Graph 1996;14(1):33–8 arXiv:1503.05249v1. doi:10.1016/0263-7855(96)00018-5.
[16] Stone JE, Sener M, Vandivort KL, Barragan A, Singharoy A, Teo I, et al. Atomic detail visualization of photosynthetic membranes with GPU-accelerated ray tracing. Parallel Comput 2016;55:17–27. doi:10.1016/j.parco.2015.10.015.
[17] Schroeder WJ, Martin KM, Lorensen WE. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In: IEEE visualization '96. IEEE; 1996. p. 93–100.
[18] Ahrens J, Geveci B, Law C. Visualization in the paraview framework. In: Hansen C, Johnson C, editors. The visualization handbook; 2005. p. 162–70.
[19] Childs H, et al. VisIt: an end-user tool for visualizing and analyzing very large data. In: High performance visualization—enabling extreme-scale scientific insight. CRC Press/Francis–Taylor Group; 2012. p. 357–72.
[20] Segal M, Akeley K. The OpenGL graphics system - a specification. The Khronos Group inc.; 2019. Tech. rep. https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf.
[21] Vulkan 1.1.127 - a specification. The Khronos Group inc.; 2019. Tech. rep. https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html.
[22] The Metal framework. Apple Inc.; 2014. Tech. rep. https://developer.apple.com/documentation/metal.
[23] Stackoverflow developer survey results 2019. 2019. https://insights.stackoverflow.com/survey/2019#most-popular-technologies; Accessed 2019-07-15.
[24] Moreland K, Sewell C, Usher W, Lo L, Meredith J, Pugmire D, et al. VTK-m: accelerating the visualization toolkit for massively threaded architectures. IEEE Comput Graph Appl 2016;36(3):48–58. doi:10.1109/MCG.2016.48.
[25] Poppendieck M, Poppendieck T. Implementing lean software development: from concept to cash. Pearson Education; 2007.
[26] Stolte C, Tang D, Hanrahan P. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. IEEE Trans Vis Comput Graph 2002;8(1):52–65.
[27] Legensky SM. Interactive investigation of fluid mechanics data sets. In: IEEE visualization'90. IEEE Computer Society Press; 1990. p. 435–9.
[28] Ensight user manual. 2019. Computational Engineering International, Inc.,
[29] Chen M, Hauser H, Rheingans P, Scheuermann G. editors. Foundations of data visualization. Springer; 2019. (expected publication).
[30] Börner K, Bueckle A, Ginda M. Data visualization literacy: definitions, conceptual frameworks, exercises, and assessments. Proceedings of the national academy of sciences 2019;116(6):1857–64. doi:10.1073/pnas.1807180116.
[31] Ceneda D, Gschwandtner T, May T, Miksch S, Schulz HJ, Streit M, et al. Characterizing guidance in visual analytics. IEEE Trans Vis Comput Graph 2017;23(1):111–20.
[32] Liu B, Wünsche B, Ropinski T. Visualization by example - a constructive visual component-based interface for direct volume rendering. In: GRAPP 2010 - Proceedings of the international conference on computer graphics theory and applications; 2010. p. 254–9.
[33] Alexa M, Müller W. Visualization by examples: mapping data to visual representations using few correspondences. In: VisSym99: joint eurographics - IEEE TCVG symposium on visualization. Springer and The Eurographics Association; 1999. p. 23–32. doi:10.2312/vissym19991023. ISBN 978-3-7091-6803-5.
[34] Gotz D, Wen Z. Behavior-driven visualization recommendation. In: Proceedings of the 14th international conference on intelligent user interfaces. IUI '09. New York, NY, USA: ACM; 2009. p. 315–24. doi:10.1145/1502650.1502695. ISBN 978-1-60558-168-2.
[35] Keck M, Groh R. A construction kit for visual exploration interfaces. In: EuroVis 2019 - short papers. The Eurographics Association; 2019. p. 79–83. doi:10.2312/evs.20191174. ISBN 978-3-03868-090-1.
[36] Sarikaya A, Correll M, Bartram L, Tory M, Fisher D. What do we talk about when we talk about dashboards? IEEE Trans Vis Comput Graph 2019;25(1):682–92. doi:10.1109/TVCG.2018.2864903.
[37] Melzack R. The McGill pain questionnaire: major properties and scoring methods. Pain 1975;1(3):277–99.
[38] Advanced visual systems inc. 1989. https://www.avs.com.
[39] Bostock M, Ogievetsky V, Heer J. D3 Data-driven documents. IEEE Trans Vis Comput Graph 2011;17(12):2301–9. doi:10.1109/TVCG.2011.185.
[40] Schroeder W, Ng L, Cates J. The ITK software guide. 2003.
[41] Institute S. 2016. SCIRun: A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI), Download from: http://www.scirun.org.
[42] Wald I, Johnson GP, Amstutz J, Brownlee C, Knoll A, Jeffers J, et al. OSPRay - a CPU ray tracing framework for scientific visualization. IEEE Trans Vis Comput Graph 2016;23(1):931–40.
[43] Unity. Unity technologies. 2019a. https://unity.com; Accessed 2019-08-03.
[44] Unreal Engine 4. Epic Games. 2019a. https://www.unrealengine.com; accessed 2019-08-03.
[45] Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In: Positioning and power in academic publishing: players, agents and agendas. IOS Press; 2016. p. 87–90.
[46] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[47] MATLAB r2019a. 2019. Natick, Massachusetts: The MathWorks Inc.
[48] McCormick P, Sweeney C, Moss N, Prichard D, Gutierrez SK, Davis K, et al. Exploring the construction of a domain-aware toolchain for high-performance computing. In: 2014 fourth international workshop on domain-specific languages and high-level frameworks for high performance computing; 2014. p. 1–10. doi:10.1109/WOLFHPC.2014.9.
[49] Kindlmann G, Chiw C, Seltzer N, Samuels L, Reppy J. Diderot: a domain-specific language for portable parallel scientific visualization and image analysis. IEEE Trans Vis Comput Graph 2016;22(1):867–76. doi:10.1109/TVCG.2015.2467449.
[50] Rautek P, Bruckner S, Gröller ME, Hadwiger M. Vislang: a system for interpreted domain-specific languages for scientific visualization. IEEE Trans Vis Comput Graph 2014;20(12):2388–96. doi:10.1109/TVCG.2014.2346318.
[51] Satyanarayan A, Moritz D, Wongsuphasawat K, Heer J. Vega-lite: a grammar of interactive graphics. IEEE Trans Vis Comput Graph 2017;23(1):341–50. doi:10.1109/TVCG.2016.2599030.
[52] Grottel S, Krone M, Müller C, Reina G, Ertl T. MegaMol - a prototyping framework for particle-based visualization. IEEE Trans Vis Comput Graph 2014;21(2):201–14.
[53] Ahrens J., Geveci B., Law C.. Visualization handbook; ParaView: an end-user tool for large-data visualization. Elsevier; 2005b, p. 717–731. ISBN 9780123875822. doi:10.1016/B978-012387582-2/50038-1.
[54] Stalling D, Westerhoff M, Hege HC, et al. Amira: a highly interactive system for visual data analysis. The visualization handbook 2005;38. 749–67.
[55] Jönsson D, Steneteg P, Sundén E, Englund R, Kottravel S, Falk M, et al. Inviwo - a visualization system with usage abstraction levels. IEEE Trans Vis Comput Graph 2019.

[56] Heckel F, Schwier M, Peitgen HO. Object-oriented application development with mevislab and python. GI Jahrestagung 2009;154. 1338–51.
[57] Bruckner S, Gröller ME. Volumeshop: an interactive system for direct volume illustration. In: IEEE visualization 2005; 2005. p. 671–8.
[58] Meyer-Spradow J, Ropinski T, Mensmann J, Hinrichs K. Voreen: a rapid-prototyping environment for ray-casting-based volume visualizations. IEEE Comput Graph Appl 2009;29(6):6–13.
[59] Rimensberger N, Gross M, Günther T. Visualization of clouds and atmospheric air flows. IEEE Comput Graph Appl 2019;39(1):12–25. doi:10.1109/MCG.2018.2880821.
[60] Chisnall D. The challenge of cross-language interoperability. Commun ACM 2013;56(12).
[61] Van Der Walt S, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22.
[62] Aja-Fernández S, de Luis GR, Tao D, Li X. Tensors in image processing and computer vision. Springer Science & Business Media; 2009.
[63] Folk M, Heber G, Koziol Q, Pourmal E, Robinson D. An overview of the HDF5 technology suite and its applications. In: Proceedings of the EDBT/ICDT 2011 workshop on array databases. ACM; 2011. p. 36–47.
[64] Rew R, Davis G. NetCDF: an interface for scientific data access. IEEE Comput Graph Appl 1990;10(4):76–82.
[65] Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the adaptable IO system. In: Proceedings of the 6th international workshop on challenges of large applications in distributed environments. ACM; 2008. p. 15–24.
[66] Martin K, Hoffman B. Mastering CMake: a cross-platform build system. Kitware; 2010.
[67] Tierny J, Favelier G, Levine JA, Gueunet C, Michaux M. The topology toolkit. IEEE Trans Vis Comput Graph 2017;24(1):832–42.
[68] Edwards HC, Trott CR, Sunderland D. Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J Parallel Distrib Comput 2014;74(12):3202–16.
[69] Ayachit U, Whitlock B, Wolf M, Loring B, Geveci B, Lonie D, et al. The SENSEI generic in situ interface. In: Proceedings of the 2nd workshop on in situ infrastructures for enabling extreme-scale analysis and visualization. ISAV '16. IEEE Press; 2016. p. 40–4. doi:10.1109/ISAV.2016.13. ISBN 978-1-5090-3872-5.
[70] Unigine 2 SDK - real-time 3D engine. Unigine. 2019b. https://unigine.com; Accessed 2019-08-03.
[71] Godot engine. Linietsky, J, Manzur, A, Godot engine contributors. 2019. https://godotengine.org; Accessed 2019-08-03.
[72] Direct3d. Microsoft Corporation; 2018. Tech. rep. https://docs.microsoft.com/en-us/windows/win32/direct3d.
[73] Houdini Engine plugin for Unreal Engine. SideFX. 2019. https://www.sidefx.com/products/houdini-engine/plug-ins/unreal-plug-in/; Accessed 2019-08-04.
[74] Unreal Studio - Datasmith 3Ds Max exporter. Epic Games. 2019. https://docs.unrealengine.com/en-US/Studio/Datasmith/SoftwareInteropGuides/3dsMax/index.html; Accessed 2019-08-04.
[75] Quixel - Shape the future of 3D art with Megascans, Bridge, and Mixer. Quixel. 2019. https://quixel.com; Accessed 2019-08-04.
[76] Substance - the 3D texturing suite. Adobe. 2019. https://substance3d.com; Accessed 2019-08-04.
[77] Maya Live Link. Autodesk. 2019. https://docs.unrealengine.com/en-US/Engine/Animation/LiveLinkPlugin/ConnectingUnrealEngine4toMayawithLiveLink/index.html; Accessed 2019-08-04.
[78] Robertson B. The fellowship of the ring. Computer Graphics World 2001;24. http://www.cgw.com/Publications/CGW/2001/Volume-24-Issue-12-December-2001-/The-Fellowship-of-the-Ring.aspx; Accessed 2019-08-05.
[79] Real Time Rendering for Feature Film: Rogue One. game developer conference, YouTube. 2017. https://youtu.be/pnigQTOig8k?t=571; accessed 2019-08-05.
[80] The Weather Channel Breaks New Ground with Immersive Mixed Reality. Unreal engine, YouTube. 2018. https://youtu.be/x2aCSV5zYIA; accessed 2019-08-05.
[81] Virtual production: Stargate Studios creates final pixels on set. Unreal engine, YouTube. 2018a. https://youtu.be/n2lp09Euotw; accessed 2019-08-05.
[82] Real-time in-camera VFX for next-gen filmmaking. Unreal engine, YouTube. 2018b. https://youtu.be/bErPsq5kPzE; accessed 2019-08-05.
[83] NVIDIA RTX Ray Tracing. NVIDIA. 2018. https://developer.nvidia.com/rtx/raytracing; Accessed 2019-08-05.
[84] An overview of Ray Tracing in Unreal Engine 4. Epic Games. 2018. https://docs.unrealengine.com/en-US/Engine/Rendering/RayTracing/index.html; Accessed 2019-08-05.
[85] An overview of the Path Tracer in Unreal Engine 4. Epic Games. 2019. https://docs.unrealengine.com/en-US/Engine/Rendering/RayTracing/PathTracer/index.html; Accessed 2019-08-05.
[86] V-Ray for Unreal. chaos group. 2019. https://www.chaosgroup.com/vray/unreal/free-trial; Accessed 2019-08-05.
[87] McLaren car configurator rendering techniques. Unreal engine, YouTube. 2018. https://youtu.be/bc2r8A7hH64?t=840; accessed 2019-08-05.
[88] "The Speed of Light" Porsche 911 Speedster concept + interactive demo. unreal engine, YouTube. 2018. https://youtu.be/QmbvSkBBrmI; accessed 2019-08-05.
[89] Unreal Studio - real-time workflows for enterprise. Epic Games. 2019b. https://www.unrealengine.com/en-US/studio; Accessed 2019-08-04.
[90] Unreal Studio - Datasmith. Epic Games. 2019c. https://www.unrealengine.com/en-US/studio#datasmithOverview; Accessed 2019-08-04.
[91] Epic megagrants.Epic Games. 2019. https://www.unrealengine.com/en-US/megagrants; Accessed 2019-08-06.
[92] Kwon O, Muelder C, Lee K, Ma K. A study of layout, rendering, and interaction methods for immersive graph visualization. IEEE Trans Vis Comput Graph 2016;22(7):1802–15. doi:10.1109/TVCG.2016.2520921.
[93] Cordeil M, Cunningham A, Dwyer T, Thomas BH, Marriott K. Imaxes: immersive axes as embodied affordances for interactive multivariate data visualisation. In: Proceedings of the 30th annual ACM symposium on user interface software and technology. UIST '17. ACM; 2017. p. 71–83. doi:10.1145/3126594.3126613. ISBN 978-1-4503-4981-9.
[94] ImAxes - immersive multivariate data visualisation in virtual reality - ACM UIST 2017. Maxime Cordeil, YouTube. 2017. https://youtu.be/hxqJJ934Reg; accessed 2019-08-20.
[95] Cordeil M, Cunningham A, Bach B, Hurter C, Thomas B, Marriott K, et al. IATK: an immersive analytics toolkit. In: IEEE Conference on Virtual Reality and 3D User Interfaces; 2019. p. 200–9.
[96] Sicat R, Li J, Choi J, Cordeil M, Jeong WK, Bach B, et al. DXR: A toolkit for building immersive data visualizations. IEEE Trans Vis Comput Graph 2019;25:715–25.
[97] Big data VR challenge 2015 (playlist). Unreal engine, YouTube. 2015. https://www.youtube.com/watch?v=EwvuqRkU_Ak&list=PLZIv_N0_O1gYWLIkoIJ6QlKCmKTXfqgbB; Accessed 2019-08-06.
[98] Big data VR breakdown. Masters of pie, YouTube. 2015. https://youtu.be/n7OlHbc0Is0; Accessed 2019-08-06.
[99] Lv Z, Tek A, Da Silva F, Empereur-mot C, Chavent M, Baaden M. Game on, science - how video game technology may help biologists tackle visualization challenges. PLoS ONE 2013;8(3):1–13. doi:10.1371/journal.pone.0057990.
[100] Kingsley LJ, Brunet V, Lelais G, McCloskey S, Milliken K, Leija E, et al. Development of a virtual reality platform for effective communication of structural data in drug discovery. J Mol Graph Modell 2019;89:234–41. doi:10.1016/j.jmgm.2019.03.010.
[101] Taking pharmaceutical discovery to the next level in Unreal Engine. Unreal engine, YouTube. 2018. https://youtu.be/fsHhhDcwGM8; accessed 2019-08-06.
[102] MolVR - basic capabilities. molVR, YouTube. 2016. https://youtu.be/9kf4QL-MqR4; accessed 2019-08-06.
[103] Wheeler G, Deng S, Toussaint N, Pushparajah K, Schnabel JA, Simpson JM, et al. Virtual interaction and visualisation of 3d medical imaging data with VTK and unity. Healthc Technol Lett 2018;5(5):148–53. doi:10.1049/htl.2018.5064.
[104] Johnson S, Samsel F, Abram G, Olson D, Solis AJ, Herman B, et al. Artifact-based rendering: harnessing natural and traditional visual media for more expressive and engaging 3d visualizations. IEEE Trans Vis Comput Graph 2020;26(1):492–502. doi:10.1109/TVCG.2019.2934260.
[105] Elden MK. Implementation and initial assessment of VR for scientific visualisation: Extending unreal engine 4 to visualise scientific data on the HTC vive. University of Oslo; 2017. Master's thesis.
[106] Reniers D, Voinea L, Ersoy O, Telea A. The solid* toolset for software visual analytics of program structure and metrics comprehension: from research prototype to product. Sci Comput Program 2014;79:224–40. doi:10.1016/j.scico.2012.05.002. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques.
[107] Lam H, Bertini E, Isenberg P, Plaisant C, Carpendale S. Empirical studies in information visualization: seven scenarios. IEEE Trans Vis Comput Graph 2012;18(9):1520–36. doi:10.1109/TVCG.2011.279.
[108] Isenberg T, Isenberg P, Chen J, Sedlmair M, Möller T. A systematic review on the practice of evaluating visualization. IEEE Trans Vis Comput Graph 2013;19(12):2818–27. doi:10.1109/TVCG.2013.126.
[109] Preim B, Ropinski T, Isenberg P. A critical analysis of the evaluation practice in medical visualization. In: Proceedings of the eurographics workshop on visual computing for biology and medicine. EG VCBM '18. Eurographics Association; 2018. p. 45–56. doi:10.2312/vcbm.20181228.
[110] Seo J, Shneiderman B. Knowledge discovery in high-dimensional data: case studies and a user survey for the rank-by-feature framework. IEEE Trans Vis Comput Graph 2006;12(3):311–22. doi:10.1109/TVCG.2006.50.
[111] Shneiderman B, Plaisant C. Strategies for evaluating information visualization tools: multi-dimensional in-depth long-term case studies. In: Proceedings of the 2006 AVI workshop on beyond time and errors: novel evaluation methods for information visualization. BELIV '06. ACM; 2006. p. 1–7. doi:10.1145/1168149.1168158. ISBN 1-59593-562-2.
[112] Stasko J, Gorg C, Liu Z, Singhal K. Jigsaw: supporting investigative analysis through interactive visualization. In: 2007 IEEE symposium on visual analytics science and technology; 2007. p. 131–8. doi:10.1109/VAST.2007.4389006.
[113] Viegas FB, Wattenberg M, van Ham F, Kriss J, McKeon M. Manyeyes: a site for visualization at internet scale. IEEE Trans Vis Comput Graph 2007;13(6):1121–8. doi:10.1109/TVCG.2007.70577.
[114] Boehm BW, Abts C, Brown AW, Chulani S, Clark BK, Horowitz E, et al. Software cost estimation with COCOMO II. 1st. Prentice Hall Press; 2009. ISBN 0137025769, 9780137025763.
[115] Symons CR. Function point analysis: difficulties and improvements. IEEE Trans Software Eng 1988;14(1):2–11. doi:10.1109/32.4618.
[116] Powell A, Vickers A, Williams E, Cooke B. Method engineering: principles of method construction and tool support. In: A practical strategy for the

evaluation of software tools. Boston, MA: Springer US; 1996. p. 165–85. doi:10.1007/978-0-387-35080-6_11. ISBN 978-0-387-35080-6.
[117] Levinthal C. Molecular model-building by computer. Sci Am 1966;214(6):42–52.
[118] Laramee RS. How to write a visualization research paper: a starting point. Comput Graphics Forum 2010;29(8):2363–71. doi:10.1111/j.1467-8659.2010.01748.x.
[119] Munzner T. A nested model for visualization design and validation. IEEE Trans Vis Comput Graph 2009;15(6):921–8. doi:10.1109/TVCG.2009.111.
[120] Sedlmair M, Meyer M, Munzner T. Design study methodology: reflections from the trenches and the stacks. IEEE Trans Vis Comput Graph 2012;18(12):2431–40.
[121] Forward A, Lethbridge TC. A taxonomy of software types to facilitate search and evidence-based software engineering. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds. ACM; 2008. p. 14.
[122] McNabb L, Laramee RS. Survey of surveys (SoS) - mapping the landscape of survey papers in information visualization. Comput Graphics Forum 2017;36(3):589–617.
[123] Kucher K, Kerren A. Text visualization techniques: taxonomy, visual survey, and community insights. In: 2015 IEEE pacific visualization symposium (PacificVis). IEEE; 2015. p. 117–21.
[124] Liu S, Maljovec D, Wang B, Bremer PT, Pascucci V. Visualizing high-dimensional data: advances in the past decade. IEEE Trans Vis Comput Graph 2016;23(3):1249–68.
[125] Kozlíková B, Krone M, Falk M, Lindow N, Baaden M, Baum D, et al. Visualization of biomolecular structures: state of the art revisited. Comput Graphics Forum 2017;36(8):178–204. doi:10.1111/cgf.13072.
[126] Beck F, Burch M, Diehl S, Weiskopf D. A taxonomy and survey of dynamic graph visualization. Comput Graphics Forum 2017;36(1):133–59.
[127] Grammel L, Bennett C, Tory M, Storey MAD. A survey of visualization construction user interfaces. In: EuroVis (Short Papers). Citeseer; 2013. p. 19–23.
[128] Mei H, Ma Y, Wei Y, Chen W. The design space of construction tools for information visualization: a survey. J Vis Lang Comput 2018;44:120–32.
[129] Bourque P, Fairley RE. Guide to the software engineering body of knowledge (SWEBOK(r)): version 3.0. 3rd. IEEE Computer Society Press; 2014. ISBN 0769551661, 9780769551661.
[130] ISO/IEC/IEEE international standard - systems and software engineering – software life cycle processes. 2017. Tech. Rep.; Working Group for Systems and Software Engineering - Software Life Cycle Processes. doi:10.1109/IEEESTD.2017.8100771.
[131] Collier D, Laporte J, Seawright J. Typologies: forming concepts and creating categorical variables. In: Oxford handbook of political methodology. Oxford University Press; 2008. p. 152–73.
[132] Bailey K.D.. Typologies and taxonomies: An introduction to classification techniques. 1996. ISBN 9780803952591. doi:10.1002/(SICI)1097-4571(199604)47:4(328::AID-ASI10)3.0.CO;2-Y.
[133] Kluge S. Empirically grounded construction of types and typologies in qualitative social research. Forum Qualitative Sozialforschung/Forum: Qualitative Social Research 2000;1(1).
[134] Doty DH, Glick WH. Typologies as a unique form of theory building: toward improved understanding and modeling. Acad Manage Rev 1994;19(2):230–51.
[135] Visweek 2010 contests. 2010. http://vis.computer.org/VisWeek2010/cfp/visweek_contests.html; Accessed 2019-09-20.
[136] McConnell S. Rapid development: taming wild software schedules. 1st. Microsoft Press; 1996. ISBN 1556159005.