ISSN 2186-7437

NII Shonan Meeting Report

No. 145

The Moving Target of Visualization Software for an Ever More Complex World

Hank Childs, University of Oregon, USA
Takayuki Itoh, Ochanomizu University, Japan
Michael Krone, University of Tübingen, Germany
Guido Reina, University of Stuttgart, Germany

February 11–15, 2019

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

The Moving Target of Visualization Software for an Ever More Complex World

Organizers:
Hank Childs (University of Oregon, USA)
Takayuki Itoh (Ochanomizu University, Japan)
Michael Krone (University of Tübingen, Germany)
Guido Reina (University of Stuttgart, Germany)

February 11–15, 2019

Abstract

Visualization has not only evolved into a mature field of science, but it also has become widely accepted as a standard approach in diverse fields, ranging from physics to life sciences to business intelligence. Despite this success, there are still many open research questions that require customized implementations, for establishing concepts and for performing experiments and taking measurements. Over the years, a wealth of methods and tools have been developed and published. However, most of these are stand-alone and never reach a mature state where they can be reliably used by a domain scientist. Furthermore, these prototypes are neither sustainable nor extensible for subsequent research projects. On the other hand, the few existing community visualization software systems are fairly complex. Their development teams, sometimes consciously and sometimes unconsciously, make decisions that affect their overall performance, extensibility, interoperability, etc. The purpose of this workshop was to discuss challenges, solutions, and open research questions surrounding developing sophisticated and novel visualization solutions with minimum overhead.

Executive Summary

The workshop identified nine topics for visualization software that require significant attention. These challenges were identified by spending the first day of the workshop with participants giving short presentations on their experiences with visualization software, with a special focus on research challenges. The participant perspectives were then organized into the nine topics. Over the following days, sub-groups discussed each of the nine topics and then presented the key points of their discussions to the group and received feedback. Shortly after the workshop, members of each sub-group wrote summaries for its associated topics; these summaries are the basis of this report. The nine topics identified by the participants were:

- How should visualization software designs be adapted to incorporate development activities from outside our community?
- Why is visualization software so hard to use? Are we good at developing effective software for users? Can we do better?
- How to lower the barrier to entry for developing visualization software?
- Should there be one community-wide visualization system for joint development?
- Can we increase reuse via building block-based visualization systems designs?
- What are the benefits and drawbacks of using a game engine for visualization software development?
- How to get funding for visualization software development and sustainability?
- How do we evaluate visualization software? What makes software successful or unsuccessful?
- Can we define a taxonomy of visualization software?

Topic Discussion Summaries

We will now summarize the findings of the respective group discussions. Each of the challenges lists all participants of the respective session(s) in alphabetical order.

How should visualization software designs be adapted to incorporate development activities from outside our community?

Katja Bühler, Dave DeMarle, Johannes Günther, Eduard Gröller, Markus Hadwiger, Yun Jang, Barbora Kozlíková, Michael Krone, Patric Ljung, Xavier Martinez, Valerio Pascucci, Guido Reina, Ivan Viola, Xiaoru Yuan

Background and Motivation: Visualization is a methodology for comprehending complex information from data visually, to support a hypothesis, to extract a model of a particular phenomenon, or to communicate knowledge across a spectrum of audiences. Such methodology is applied through usage of visualization software that has been designed for a particular purpose or for generic more workflows. Visualization software is typically developed over several years, or in several cases even decades of intensive development. In the initial phase, the architecture of visualization software is designed to fit a purpose with specific technologies that are modern and well-supported at that time. Sometimes, even certain risks need to be taken when the bleeding-edge technology is utilized. Those developers, who decide on a prospective technology

of the future, gain a substantial advantage, as their software will age more slowly than if using well-established, but possibly (out)dated software and hardware frameworks. This interacts with the fact that we live in a rapidly evolving world where new technologies frequently offer new opportunities for tackling a particular problem. As such, the life cycle of visualization software typically witnesses the introduction of several new and disruptive technologies that can potentially be beneficial for the visualization software. Whether, why, and how such an evolving ecosystem should be incorporated into existing visualization software is a nontrivial question to ask.

Which are the technological aspects that might influence how the software is shaped over time? One source of new technology that can expand the visualization software is the technological and scientific outcome of the visualization research community itself. Visualization research has become a very active area, each year attracting more researchers to address the open research challenges. Another set of new technologies might come from outside, a trend that is manifesting itself in user requirements, for example. A typical case can be web-based technologies that avoid multiple complications in user adoption. Web-based deployment allows software to be used right away, without tedious installation requiring elevated user rights. It can be used on any platform and allows to distribute the intended computation and data management between user's client and (at least) a remote server. Today, in the times of blooming sharing economy philosophy, instead of having the computational and storage resources locally in the client, software design has the opportunity to take advantage of externally offered services in the cloud. This might offer the benefit of lowering the costs for the hardware and the software. Furthermore, if the cloud-based components are maintained by other development teams, it avoids the burden of implementing everything with just one team. Visualization software is naturally affected by the advances in graphics hardware and the evolving concepts on how to harness these in the visualization pipeline workflows. Graphics processing units (GPUs) have drastically evolved in the last decades, offering incredible speedup opportunities if utilized efficiently. The respective programming models are also frequently innovated, e. g. introducing new shader types, or even entirely new graphics hardware APIs, such as Vulkan or Metal, to reflect the new hardware capabilities through modern ways of programming. In addition, CPU parallelism has evolved significantly, and together with dedicated, efficient libraries that can take advantage of modern hardware architectures for rendering three-dimensional scenes (e.g. OSPRay), offers a huge advantage to the developers (and to users, who do not have to rely on the availability of GPUs). One important aspect is the avoidance of the so-called 'not invented here' syndrome and consequent offloading of maintenance of a particular functionality to a dedicated development team outside. This comes with its own advantages (potentially better code maturity, saved time, etc.) and disadvantages (low influence on API and features), but, except for specific cases, the advantages dominate.

In the last years, the visualization community started to recognize the significance of machine learning and artificial intelligence (AI) for automating increasingly complex tasks. Already today, deep network architectures are omnipresent in science, technology, and medicine. AI is often considered a silver bullet for many workflows, at least as long as large amounts of training data are available. Classical visualization software is algorithm-centric. Certainly, some of the visualization workflows would significantly benefit from adopting AI

methodology. However, integrating such disruptive technology, which induces a shift of paradigm, into an algorithm-driven design is highly non-trivial.

A less disruptive change, but still a major challenge to tackle from the programmer's perspective in practice is the adoption of new programming languages. In the past, many developers heavily invested into java as a new, modern programming language. Today, java is well-established, but other concepts that were in the past much less promising are taking its place in the hype cycle, such as JavaScript. Meanwhile, C++ is still often utilized, especially in high-performance computing tasks where computational efficiency is a strict requirement. However, even C++ has significantly evolved over the last years. Shading languages are another example, where many new opportunities are offered. General-purpose GPU computing APIs, such as CUDA, are yet another example where adoption might be rewarded by significant benefits. Scripting or domain specific languages are another very popular form of user interface that offers flexibility and programming capabilities that has not been foreseen twenty years ago. Adopting these concepts can turn a user with limited computer literacy into a programmer able to solve her/his tasks without the need of a dedicated developer. One more step in freeing up developers from graphics programming are various game engines, such as Unreal, Godot, and Unity, or graphics engines, such as Cinder. These offer the opportunity to developers to leave maintenance of basic functionality to an external development team, and contribute only high level functionality.

When emerging technologies surface, for developers of software systems it is a tough decision whether the new technology should be adopted and when the right time is. When a software system has grown together with a particular API, such as OpenGL for example, it is very difficult to replace it with another API. It is also an opportunity to rejuvenate the software system, e.g. to build an abstraction on the top of the graphics API, however it can also easily mean multiple man-months of investment. As an example, the well-known visualization toolkit VTK has recently undergone an upgrade of OpenGL from 1.1 to version 4.2+. While several visualization algorithms have increased in performance by several orders of magnitude on modern graphics hardware, the costs of one man-month were substantial. Recently, when Intel decided to support integration of OSPRay into existing visualization systems, VTK again as well as MegaMol have both been extended to make use of OSPRay functionality. Several visualization systems are instead migrating the code towards web-based implementations. Brain* and Mol* are examples where this has been successfully accomplished. User base growth potential has increased significantly through this decision. Nanographics' whole-cell visualization framework marion now supports remote rendering that is interactively manageable from the client side. In this case, not only the significant additional implementation effort was necessary, but also abandoning the Unity game engine, due to licensing restrictions regarding broadcasting the content via remote rendering.

Challenges: Visualization software builds and depends on the surrounding hardware and software technology. However, this ecosystem is constantly changing and evolving. Adapting to those changes is necessary or desired for various reasons. One of the causes that drive change is actually of legal nature: the licensing terms of used software packages or technologies may change in a way

that prevents to continue the operation of visualization software, or may incur unplanned costs. Another instance of legal issues is the extension of visualization tools to usage scenarios that are not covered by the current licensing terms, examples are remote rendering, multi-user capabilities, or multi-node rendering.

However, the main motivation for adopting new technologies for visualization is rooted in its high requirements: large and complex data sets need to be processed and visualized with high frame rates, resulting in high computational load and requiring more efficient technology. Operating at this forefront of technology has inherent challenges. First, there is uncertainty: it is often difficult to predict which new technology will prove to be valuable, or which one turns out be a (short-lived) hype. Second, technologies may be (partly) discontinued. For example, the Mantle project was deprecated, OpenGL was effectively removed from the Mac platform, and the number of supported platforms for the Oculus Rift was reduced. Lastly, the newest technology is often fragile and unreliable in the beginning, showing unspecified behavior or performance deficits. Dependent software then has to wait for updates in which these issues are fixed. Sometimes profiting from new technology requires little effort, e.g., by using newer and faster hardware. But more often than not, switching technology is expensive: it requires significant time to investigate and evaluate new technologies, an investment which may not pay off if a wrong technology was chosen (the definition of 'wrong' very strongly depending on the context). Then, learning the proper usage of new technology and getting acquainted with it to beneficially exploit it takes time as well. Especially disruptive technology represents challenges, as significant effort is needed to break out of the current mindset, because fundamentally different concepts of usage may be required. Furthermore, it may be necessary to maintain backward compatibility with current technology, for example to still support users with older hardware, which generates additional overhead.

Another area where the environment of visualization software creates challenges revolves around interoperability. In particular, many different and specialized data formats are an issue: data formats of domain-specific simulation codes are developed to serve the requirements of the simulation, which may be ill-suited for the task of visualization.

Based on discussion, we extracted several questions for further research. In general, a good software architecture and abstractions that support flexibility and future-proofness are the means to meet the above-mentioned challenges. Such abstraction can be implemented on different levels. First, how to best exchange data between different software packages from different scientific domains. We live in an increasingly complex world, thus interdisciplinary work is necessary. Can we conceive common file formats and APIs to interchange data – and what will those look like? Second, there are already established abstractions useful for visualization: commercial and open source game engines hide the different flavors of low-level graphics APIs, offer easy access to various platforms (e.g., mobile, desktop, consoles, VR/AR), and can provide excellent performance if used properly. There are several potential game engines to choose from, among others Unity, Unreal, and Godot. These systems have different capabilities, strengths and weaknesses. Thus, an in-depth evaluation and comparison of major game engines including thorough performance benchmarks will be a valuable research contribution. Third, domain specific languages (DSLs) can provide abstraction beyond APIs while offering conciseness and expressiveness. While there are already proposals for DSLs targeting specific and narrow visualization

sub-domains, can we conceive one DSL for general visualization?

Conclusion: In conclusion, there are two main strategies for dealing with a changing environment for visualization, with distinct advantages and benefits. One direction is to avoid disruptive changes and to wait until the, e. g., low-level graphics ecosystem stabilizes. This can be achieved by using modern, high-level abstractions, e. g., game engines. Besides gaining better capabilities, such systems open up new possibilities and, most of all, increase productivity, if they fit the requirements. Thus, instead of dealing with time-consuming fixes in low-level implementations, the saved time can be invested into improvements of visualization and exploration of new ideas.

The other main direction is to embrace change and, as an early adopter, gain the first-mover advantage. For example, this could mean getting rid of legacy code (for example OpenGL code before version 4.6) and to start using SPIR-V for a gradual transition. This transition can be eased by starting with wrappers, to finally arrive at bare-metal implementations for maximum flexibility, control and performance. It may be an option to hire experts to help with this transition.

Either way, change also opens up new opportunities: Promising research avenues are more likely and new approaches to old problems may present themselves. Additionally, students benefit when they gain experience within the new environment and with new techniques.

In case of either of those strategies, academic community, students, as well as industry engineers, gain the opportunity of prototyping new ideas with the latest technologies to study new grounds. Prototyping typically gives a good indication whether a particular approach is worth a more thorough effort. Prototyping however cannot replace stable and sustainable software development, code quality quickly degrading during and after the prototyping phase. Although it usually takes only a short amount of time, even prototyping can be an expense that is not always affordable. The conclusion here is that creating a stronger social community that can share the pains and gains of experimentation could be the way to move forward.

Why is visualization software so hard to use? Are we good at developing effective software for users? Can we do better?

All attendees discussed this topic

This topic was discussed in the first breakout session by all participants (three groups in parallel). This topic was revisited later in the week. The summary incorporates aspects from all sessions, but is specifically derived from the last discussion, which involved Hank Childs, Dave DeMarle, Eduard Gröller, Shintaro Kawahara, Guido Reina, and Xiaoru Yuan.

Background and Motivation: This discussion was centered around why much of the software we develop is difficult to use, preventing adoption by domain scientists. One characterization was that the software we develop requires someone with Ph.D.-level expertise in visualization to use. Of course, this concern does not apply uniformly, as some software from our community is

easy to use, for example Tableau. Other products, like FieldView and EnSight, are able to collect licensing fees, demonstrating that user experience is, at the very least, acceptable. Further, open source tools like ParaView and VisIt have been able to garner large user communities, again speaking for usability.

Challenges: A central challenge centers around the goals for the software we are trying to create. While perhaps as an oversimplification, these goals can be thought of as covering a spectrum, with an individual Ph.D. student developing software to further their research on one end (research software), and a commercial company trying to make a profitable product on the other end (commercial software). When research software is delivered to domain scientists, there can be a conflict in desired outcomes, as often the domain scientist wants to solve a problem, but the visualization scientist wants to develop a novel technique. Further, visualization scientists often considers what they develop as a prototype, but the end user is expecting a product, comparing against industrial-grade software like iPhones, PowerPoint, etc. In these cases, it is possible that the allocation of resources to developing software is still sensible (i.e., the visualization scientist should not spend months of development time to save the domain scientist minutes while using their software), but the expectations are mismatched. Regardless, in many cases, visualization scientists can do more to close this gap, including being more knowledgeable about human-computer interaction, being more committed to writing documentation, and having increased knowledge of visual literacy.

There are also open research questions surrounding visualization software usability with respect to user interface (UI). Two high level questions emerged: (1) How can we automatically generate simplified versions of UIs?, and (2) Do we have the wrong UI paradigm altogether? For the first question, we believe an important sub-question is: what are the building blocks that would assist this automatic generation? For the second question, we believe there are many directions: Should our UI consist of asking users what they want to do, like a conversation? Should the program suggest examples of successful visualizations as a starting template? Should machine learning be involved in either guiding the user or in filtering what they see?

Conclusion: Making more usable visualization software would have benefits for both visualization scientists and domain scientists. For domain scientists, outcomes range from solving problems in a short period of time (a time savings) to the ability to discover more in the same period of time (increased knowledge). For visualization scientists, there are additional benefits. First, user adoption leads to multiple, positive outcomes, including increased funding, an increased reputation in the community, and (for corporations) increased competitive advantage. Further, easier to use software requires less user support, which saves visualization scientists time. If the user community becomes large enough, then this community can start to support itself (blogs, mailing lists, wikis, etc.), which can further save time for visualization scientists. Finally, having software that is more usable may increase the chances of commercialization, either by adoption from a company or by industrious students taking their research and attempting to commercialize it.

How to lower the barrier to entry for developing visualization software?

Katja Bühler, Dave DeMarle, Takayuki Itoh, Yun Jang, Michael Krone, Patric Ljung, Xavier Martinez, Kenneth Moreland, Allen Sanderson

Background and Motivation: Visualization software often has a high barrier to entry. Even for Ph.D. students, it is often too hard to run and extend the visualization software. Also, it is not a straightforward task to teach the usage and development of visualization tools because it requires both expert knowledge of visualization techniques and experience in software engineering. That is one of the main reasons why we need to lower the barrier of entry of visualization software for users and developers. Here, we remark that requirements of end users are different from those of developers.

Challenges: There are many challenges to lower the barrier to entry. We have discussed this topic mainly on two sides including developers and users. On the developer side, the primary concern is that there is a lack of developer-centric documentation. Often, documents for the developers are missing, stale, and even wrong. Another challenge is that not all developers know multiple programming languages, which causes delays in development since the developers must either learn new programming languages or search for other libraries. Also, APIs for interfacing with other programming languages are not well supported. More challenges include configuration management, maintainability, and backward incompatibilities. Many development kits require complicated building and compilation procedures with many external dependencies. Besides, many libraries are not well maintained, i.e., they are not updated to the state-of-the-art techniques or might not even be compatible with the latest technology anymore. Therefore, many developers tend not to adopt new libraries unless they trust them with respect to the aforementioned issues. More things that the developers are checking include how long the product will survive, whether bugs will be fixed soon, what new platforms will be supported, and whether coding styles are appropriate. On the user side, it is challenging for developers to produce a comprehensive user manual including a list of features, tutorials, and examples. Often, there are conflicting requirements for basic users and power users. Simplified interfaces desired by basic users necessitate hiding or removing advanced controls desired by power users. Also, end users in general do not like to compile software. Another main concern is that the developers sometimes do not know the mindset of their users and what they need. This can decrease the trust into a specific visualization software.

To tackle the challenges, our community has tried many things. For the developers, wrapping into interpreted languages, such as Python and Lua, and meta-build tools, such as *build_visit* and *superbuild*, have been developed. Also, package management systens like Nuget (MS Visual Studio), npm, or macport have been released. Visual programming including Blueprints in Unreal Engine and pipeline designers have been proposed, but with mixed success. To help the developers, many communities have built forums and mailing lists and rewarded helpers, via kudos and StackOverflow scoring and achievements to

make communities more active. For the users, one-click solutions including Docker, Singularity, and web-based visualization tools have been examined to lower the barrier to entry. To support repeated tasks, macro and compound modules are supported in some visualization software. For more intuitively usable software, reconfigurable UI and UX as a whole has been studied for existing software. Moreover, user-contributed tutorials including videos and blogs are very common for game engine platforms.

Conclusion: We recommend the following measures to lower the barrier of entry for visualization software:

- Documentation is one of the most important issues. A sufficient amount of documentation (including manuals) will make software accessible. We also recommend providing a sufficient number of examples and informative comments. Here, comments describing how to use the software are more informative rather than explaining the code itself. Incompatibility of software libraries often causes runtime problems. Lists of dependencies, including appropriate version numbers of libraries, should be published in the documentation. Consistency among tools is also an important issue. Software providers should carefully keep consistency of interfaces across the visualization tools. Or, they should describe any inconsistency in the documentation.

- We will need to be conscious about abstraction levels of interfaces sometimes. It is desirable to provide different levels of abstraction for different classes of users.

- From a developer's perspective, it is desirable to keep your own code independent from the underlying framework, so that it can be re-purposed or re-used when needed.

- Users and developers may contribute software by reporting failures. We recommend to reward such helpers. Software providers need to make connections to know users of the software and understand their application context. Communication should therefor not be limited to failure reports.

Meanwhile, we still have some open questions to lower the barrier. First of all, we were not yet able to come up with a measure for the barrier to entry for visualization software. Such a measure quantifying the barrier would make it easier to recognize said barrier and to discuss concrete solutions. Also, it would be convenient to find methodologies to efficiently develop low-barrier software. For example, we expect the latest artificial intelligence technologies will realize automatic intelligent documentation and adherence to coding conventions. More realistically, it would be convenient if we had automatic alerting for developers of incomplete or inconsistent documentation. We also discussed further issues, including how to mix and match to get features from different visualization systems, and how to establish best practices for reusable visualization software design.

Should there be one community-wide visualization system for joint development?

Johannes Günther, Markus Hadwiger, Kenneth Moreland, Timo Ropinski, Allen Sanderson

Background and Motivation: A very important question in visualization is the basic software infrastructure that should be used for research and development. One model that is being used successfully is to leverage a common monolithic framework such as VTK, Amira, Voreen, VolumeShop, MevisLab, Inviwo, MegaMol or others. However, not every group working on visualization research or software development is building on the same monolithic framework.

Often, there are differences between the research performed, for example, in national labs versus the research done in individual research groups in universities. The former often build on VTK, whereas the latter often either develop their own monolithic software infrastructure for their own group (and maybe collaborators) or develop a new software infrastructure for the research of individual PhD students or sometimes even for individual papers. An important influence on these choices is whether it is important to support a large user base (including domain scientists that are neither visualization researchers nor developers) or whether the primary goal is producing a novel algorithm for a research paper.

However, in order to be able to perform novel research in an efficient way we would like the visualization software developed by different groups to work well together, to able to leverage each other's work and especially not reinvent the wheel every time a new research idea should be turned into a software prototype. Moreover, this would also make the transition from research prototype to professional/commercial software easier. That is, should all visualization researchers and developers use one big sandbox, e. g., VTK, for all projects? Why is not everyone using VTK?

As illustrated by the many successes of using VTK, agreeing on a powerful monolithic framework has many benefits. However, it is also limiting, because sometimes new research ideas, hardware developments, or specifics of new data modalities or even just much larger data sizes do not fit well into the existing codebase. Moreover, in order to use a single framework, everyone must agree on which framework to use and whether it is the best choice for the problem at hand. Sometimes, the benefits of using existing software can be outweighed by limitations on freedom or creativity due to the inherent constraints of an existing system that might have been designed with different goals (or hardware architectures, etc.) in mind. It is unlikely that a huge monolithic system can ever have full acceptance by an entire community.

So if different research groups use different software frameworks, how can they be made to work well together? Efficient interoperability is required in order to be able to leverage and combine existing software.

The first important question for interoperability is the data model. After getting access to software of a colleague or collaborator, even assuming that it compiles and links properly with the existing software, it is rare that data can be shared or exchanged directly. There are too many different formats and data structures in which data can be represented.

Some examples for existing data structures/formats are NumPy (multi-dimensional arrays), file format containers such as Hdf5, netCDF, ADIOS, file

formats such as NRRD and VTK. Domain-specific file formats include PDB (for molecular dynamics data) and DICOM (for medical imaging, including many container types).

Some examples for existing software frameworks and libraries that are used successfully in visualization are VTK, ITK, TTK, Kokkos, OSPRay, Inviwo, Qt, libraries for reading Hdf5, netCDF, ASSIMP, Glm, etc.

Another crucial question is the programming language and corresponding infrastructure used. C++-based frameworks have a noticably harder job when integrating different frameworks, libraries, and components than frameworks based on Python (e.g., NumPy, SciPy) or Javascript (e.g., D3). This leads to more people integrating and using frameworks based on Python, Javascript, or similar languages.

Having a central repository for libraries, especially larger libraries that can be broken down into smaller parts would be very helpful. Examples are boost (to some extent), Anaconda, and other package managers.

Challenges: To support community-wide efforts, basically two approaches are conceivable. There could be a single visualization software system, which follows a standard to be agreed upon by the visualization community. Alternatively, an eco-system of multiple visualization systems could be maintained, while interoperability between these systems would have to be established.

When investigating the idea of a single visualization software system, again two alternative ways of realizing this are possible: the system could be centralized or decentralized. VTK for instance follows the centralized approach, i.e., it is managed by a single entity and all contributions have to go into a single centralized repository. Furthermore, it is difficult for users to focus on a subset only, which would require to configure arbitrary subsets of VTK. ImageJ on the other hand follows a decentralized approach. While the ImageJ application only provides an application frame with the most essential functionality, e. g., to open and view data, external users can easily provide plugins to extend the functionality of this application frame. Due to the decentralized nature, these plugins can be published as JAR files on any website, from where they can be downloaded and copied into the ImageJ folder to extend ImageJ's functionality. This approach has the benefit that it allows contributions without the need to "ask for permission", and that on the other hand the application frame developers do not need to support others people's software. We believe that this lowers the barrier to entry, and scales better with respect to extending the system at hand. Of course, this approach also has downsides to it. It needs to be clarified how to manage the large numbers of dependencies, and how to deal with software blocks which have been released at different times. Furthermore, there exists a varying level of support, and users need to trust the developers of extensions not to include malicious pieces of source code.

Although we believe that it is very difficult—and maybe even counter-productive—that the visualization community adopts a single visualization software system, we would also like to elaborate on the eco-system of multiple visualization systems. The big challenge in this setup would be the interoperability of these different systems. We see two principle approaches to how such an interoperability could be achieved: either through a standardized programming interface, or on the data model level. Although a standardized programming

model would be more flexible and would support developing interactive visualization applications combining different visualization systems, it is considerably harder to realize, especially when multiple programming languages are involved. On the other side, using a common data model also has several challenges attached to it. For instance, once the community agrees on such a data model, they would be "locked in", which could hinder creativity, and flexibility with respect to to new data modalities. Nevertheless, the biggest challenge would be to convert all already existing data sets to allow them to be visualized.

All of these challenges aside, the questions needs to be asked whether it would be a valid approach to solely provide data models without any functionality. The workshop members considered whether more people, including themselves, would use the VTK data models if they did not have to bring in the bulk of VTK. The general consensus was, yes, if there was a lightweight library implementation more developers and researchers would be inclined to use it. So while we think that having a single data model would be hard, everyone having their own data model gives rise to an n^2 problem with respect to data conversion across systems. Alternatively, it could be investigated whether it is helpful to have a small set of data models for different classes of problems or communities. For this, we would need to determine the different requirement of the respective communities—for instance, simulation analysis, topology analysis, molecular graphics, or medical visualization. Maybe a taxonomy of communities could help to make these grow systematically.

Independent of which way is chosen to support community-wide visualization efforts, there are some challenges that are inherent to all approaches. Any chosen approach requires a critical mass, such that many researchers and students are on board. We should investigate how to convince researchers and students to bring in their ideas and contribute. Also, it needs to be investigated if the time needed to realize the discussed ideas is worth it with respect to the expected benefits or if it is too big an overhead.

Conclusion: We would like to conclude with the following observations. We believe that the challenges developing a single community-agreed system might outweigh the benefits despite the fact that using such a system would allow for fast exploration and replication of published results. With respect to the employed data model it might seem faster at first glance to just use your own data structures, particularly for simple data. However, in the long run it saves time (including student time) to learn and leverage an existing system, as it might often be sufficient to understand the underlying data model. Once this task is completed, it can be leveraged across visualization software systems. Furthermore, if there is no agreed-upon data model, there are usually either performance issues or storage issues, as data needs to be either converted between data models frequently or stored in multiple formats. So in the long run, providing a single, extensible and community-agreed data model might be a successful approach. In the short term, we believe a viable option is to create a bundled development environment, which contains important visualization libraries. Visualization developers could download such an environment, and configure which libraries to use, in a similar fashion as a package manager would work. However, instead of packages, the bundle would provide libraries, such as for instance VTK, Kokkos, Hdf5, netCDF, ASSIMP,GLM, OSPRay, Inviwo, DICOM, ITK, TTK, and Qt.

Based on the selection from these libraries, then a project file for the desired IDE could be generated, whereby all dependencies between the libraries would be considered. Once the project is opened in the IDE, the visualization developer could start coding by using the selected visualization libraries. CMake might have all capabilities to achieve this, as the ExternalProject extension allows to integrate and clone external repositories, which could be done upon request during the CMake configuration stage.

Can we increase reuse via building block-based visualization systems designs?

Hank Childs, David DeMarle, Johannes Günther, Markus Hadwiger, Yun Jang, Xavier Martinez, David Pugmire, Guido Reina, Allen Sanderson, Ivan Viola, Manuela Waldner, Xiaoru Yuan

Background and Motivation: Our community has generated many visualization software systems, and while no two visualization software systems are the same, they often contain common elements. In particular, they typically contain elements around user interaction, specific visualization algorithms, rendering, windowing, etc. In some cases, the elements that are being developed are novel and require new implementations. In other cases, the elements that have been developed in one tool could be reused for another. With this topic, we assume that this reuse would come via community software that is composed of building blocks. These building blocks would be developed by the first person that needs a capability, and then would be reused (and possibly improved) by subsequent people who need the same capability.

Our community has produced some successful efforts to date for visualization building block efforts. These building blocks do not have a common granularity. In some cases, they come as visualization toolkits with composable modules. Examples include AVS, D3, OpenDX, ITK, SciRUN, VTK, and VTK-m. In other cases, the building blocks come as modules/libraries that solve one part of the problem very well. These examples mostly draw from environments that do rendering, e.g., OSPRay, and graphics, windowing, and interactions, e.g., Unity and Unreal. Further, some environments provide the glue to plug together modules, such as Python, Jupyter Notebook, SciKit Learn, and Matlab. Finally, our community has developed other "glue environments" that plug modules together via domain-specific languages, such as Scout, Diderot, Vislang, and Vega Lite. There are also many applications that provide "glue environments" via a GUI (or scripting), examples coming from the participants of our workshop included FieldView, MegaMol, ParaView, and VisIt.

Despite these successes, it not clear how to design building block systems that will meet a larger portion of our community's needs.

Challenges: There are many challenges surrounding the topic of building blocks. We organize the discussion in three parts: technical aspects of developing a system, technical aspects of maintaining a system, and non-technical aspects.

There are multiple barriers around technical aspects of developing a system. First, there is a diverse set of requirements, many of which may be in conflict. These include platforms (Windows, Mac, Linux), languages (C, C++, Fortran,

Python, Java, Javascript, etc.), science domain to be supported (biological, medical, physics, etc.), data model, and many more. For example, a virtual reality application for molecular visualization requires a rendering infrastructure that can deliver a very high frame rate (e.g., 60 FPS), while (desktop) applications centering around very large data sets from physics simulations can often tolerate much lower frame rates (e.g., 10 FPS). This difference in requirements helps explain why these two communities have pursued different strategies when it comes to rendering. Regardless, common building blocks may exist that benefit both. A second, related concern focuses on trade-offs. Principles like flexibility and generality often increase reuse, which reduces overall developer time (at least for the next developer). But these principles are often in conflict with performance goals. In some communities, such as those where users expect many features, sacrificing performance may be the only way to deliver a product with an acceptable richness of features. In other communities, this is not the case. Another aspect of trade-offs and performance is the granularity of the building blocks. Here, we think of building blocks designs as spread along a spectrum from fine to coarse, i.e., meaning building blocks perform very small tasks (fine) or very complicated tasks (coarse). The former design encourages reuse, while the latter encourages performance. A third consideration for this topic is interoperability. To generally maximize reuse potential, multiple building blocks will need to work together. In particular, this requires addressing difficult issues relating to shared data models.

There are also barriers around technical aspects in maintaining a system. One concern is in sufficient testing. This concern partially comes into effect when a developer community becomes large, and each developer writes only a small amount of code. Further, hardware makes this more difficult, since, for a large software base, developers may only have access to a subset of the hardware that may needs to be tested, i. e., GPUs, distributed-memory parallelism, PowerWalls, virtual reality devices, and more. The risk, then, is that code will become fragile when these hardware-dependent features are incorporated. Another concern is about managing a large code base. If every developer were to contribute their homegrown volume rendering code, then there would be many, many modules to choose from, sometimes with only very small differences. In short, a very vibrant project runs the risk of being hurt by its own success. A final concern is around future-proofing the project. As requirements change over time, the system must be able to evolve with these requirements, or it will become obsolete.

The final set of barriers we considered were those not of a technical nature. The first topic is obtaining funding, both to do initial development and to perform maintenance, both of which are discussed as its own topic in this report. Another topic centers around community. For a building block project to be successful, it must attract both developers and users. On the user side, this means that the building blocks should be easy to use, and also (in some cases) easy to incorporate into user workflows. On the developer side, there are several concerns. One is to encourage developer participation, i.e., a low barrier to entry to writing modules and contributing them back. An important additional aspect surrounding encouraging developer participation is a reward model that incentivizes individual developers — if a developer contributes code and another entity benefits, then it would discourage participation. Additional important concerns are to coordinate development (how is the community moderated?) and ensure that developers do the activities to enable the project to succeed

(e.g., documentation).

Conclusion: The benefits of solving the challenges described above are many. We organize the discussion of these benefits around costs, community, and advancing science.

In terms of cost, the primary benefit is to minimize developer time to build new software. That said, a successful building blocks effort would lead to additional cost-related benefits. For one, when the cost of developing new software is low, it enables new paradigms, for example developing rapid prototypes or software that is intended for very small user groups. Further, this model enables potential access to better solutions (faster, error-free, more features) than a funding source may have paid for otherwise. Finally, there is a cost savings benefit around developing expertise. With a building block model, a programmer can develop a module within their own expertise which in turn can be used many times by non-experts. Taken to the extreme, novices can leverage work by many experts to develop solutions quickly.

In terms of community, there are also many benefits. One community benefit is an economy of scale, which is another perspective on the cost discussion just presented. Another community benefit involves increased impact. When many people work together on a project, the modules developed by one person are likely to be used by more. A third benefit is validation: as the software will be used in more contexts, its results will be (re-)viewed by more people, which can lead to improvements in the software. This can ultimately result in a more stable software that has a longer lifetime. A fourth benefit is the increased clout that a larger software project garners, which includes (but is not limited to) a benefit for all stakeholders because this will likely lead to a longer lifetime. Finally, as a fifth benefit, when a project is maintained by more and more people, it lessens the blow to the project when any of those people leave due to retirement, graduation, changes in interests, etc.

Finally, two additional benefits were identified with respect to advancing science. The first is reproducibility. If algorithms are contributed to a public system, then their results can be reproduced. This topic is currently a sore point in our community, as many research results are not easily reproduced. The second benefit is improved education. As software becomes increasingly sophisticated, an inherent property of the building block design is that it creates a level of abstraction for the encapsulated functionality. Consequently, even complex, feature-rich software can still be used by users with less expertise.

What are the benefits and drawbacks of using a game engine for visualization software development?

David DeMarle, Yun Jang, Michael Krone, Patric Ljung, Xavier Martinez

Background and Motivation: Game engines have now been around since the late 90's and many have matured into very large and feature-rich software platforms for developing highly advanced user experiences. There are some engines that have been made available to a general audience of developers, such as Unity3D and Unreal Engine 4. Godot is an Open Source (MIT License) game

engine that was first released publicly in 2014. Unity and Unreal Engine are by far the most used game engines freely available. Although Unreal Engine is under a commercial license the source code is available to registered developers. Both engines provide marketplaces for developers and artists to sell and buy assets, plugins, and tools, including free content.

The structure of a game engine is that it provides a generic platform with a rendering pipeline, application logic, and asset management. Virtual worlds are created in a loosely coupled manner, in contrast to a scene graph, and each object has a set of components attached to it, that provides various behaviours or specific features. This allows for reuse of such features without typical object-oriented class inheritance. As such, a game engine allows for the developer to configure and decide exactly how the system should operate, in order to provide the best performance.

Unreal Engine, with over 20 years in production and made open source in 2015, and Unity 3D, initially released in 2005, have state-of-the-art graphics rendering support but have also matured in many other aspects. Many features may, however, be of little use in typical visualization applications and specific features useful for visualization are not present in game engines.

The motivation for using game engines is to exploit the rich set of features and high graphics rendering quality, the ecosystem for sharing or selling assets, and so forth, for visualization purposes. Regarding the "missing" features desired in visualization applications these can in many cases be added on top of the game engines as plugins and tools that can be made available on the marketplaces.

Another motivation is that OpenGL will likely become obsolete[1] and replaced with Vulkan, Metal, DirectX, and WebGL, as the markets appear today. An alternative for reaching cross-platform support is to use game engines that already support all relevant platforms[2] and APIs [3], thus building upon a higher abstraction layer, although low-level access is still allowed but with more constraints.

Current issues slowing down adoption of game engines in scientific visualization: The main reasons why the visualization community is not massively using game engines might come from the absence of data processing pipelines and no high-level access for dynamic mesh and primitive generation and alteration.

Also, game engines mostly take full control over the whole rendering pipeline that the visualization community sometimes needs access to. Indeed, one of the most commonly used game engine to date is not fully open-source (Unity3D only provides paid access to the source code) and therefore does not offer the flexibility needed. Access to writing custom shaders is provided but they still have to obey the rules of the engine rendering pipeline. The Unreal Engine source code is available to developers and it is possible to rewrite parts of the core engine rendering pipeline, which, however, may be a massive undertaking.

Another point would be that using game engines feels like hacking a tool that was not made for scientific visualization[4]. For example, the video game

[1]The most recent version, OpenGL 4.6, was released on July 31st, 2017.

[2]Windows, macOS, iOS, Linux, Android, PlayStation, Xbox, Nintendo Switch.

[3]DirectX, Vulkan, Metal, OpenGL, GL|ES, WebGL/HTML5, PS4 GNM/GNMX, NVN (Nintendo Switch).

[4]This, however, is not very different from the way early GPU generations were re-purposed for what was called *GPGPU* back then and today is catered for using compute functionality

developers tend to use statically-generated objects and meshes, with baked lighting, whereas in the scientific visualization community, most of the meshes are generated at run-time based on the data to visualize. Tools to manage these objects are usually not accessible at run-time. Also, game engines keep large portions of the data resident in GPU memory for performance reasons, whereas visualization might require falling back to streaming data, especially when there is not enough available memory.

As famous and high-quality game engines are usually created and maintained by large companies (Epic Games for Unreal Engine and Unity technologies for Unity3D), users have no leverage over their general orientation and frequent changes that the company will enforce. Game engine users still have the choice to keep a certain version but without new features and bug corrections, or re-engineer their code base to adapt to the changes. A similar behaviour could be observed when using a third party library but large game engines tend to have breaking changes more often.

As game engines are tools to create commercial games, the license imposed by the software can also be restrictive and/or expensive. In a similar way, support can also be an extra cost to pay even if the quality of this support can be high.

Game engines tend to provide a wide range of features, the complexity of these tools imply that users will invest a lot of time in one platform. For example, Unreal Engine is today a huge software with many features, a high barrier of entry and a steep learning curve that could discourage people to use it.

Also, the information visualization community needs to display non-spatial data, for example 2D plots, and process data on the fly. That would require to develop respective tools in game engines. Note that this has been done many times during the last years [2].

Several popular scientific visualization projects based on game engines can be cited. One current active project developed in Marc Baaden's lab is UnityMol, a molecular viewer that uses Unity3D game engine to render molecules in VR/AR/Desktop environments. Another active visualization software initially based on Unity3D named CellView is developed by Ivan Viola's group and provides a multiscale view, from a whole cell to atomistic details of proteins inside the cell. The Immersive Analytics Toolkit (IATK) is a visual analytics software tool, also based on Unity. NVIDIA has demonstrated a tile-based remote rendering feature using GPU-based video encoders/decoders in which ParaView was used together with Unreal Engine on the server side for rendering, where meshes are passed to Unreal from ParaView[5].

Another example of the growing interest for game engines in scientific visualization is an VTK to FBX exporter that enables working with VTK geometry inside game engines to benefit from the various features they offer.

Challenges: Game engines have a different system design compared to many visualization tools, for the game engines the focus is a rich visual environment and persistently high performance. The learning curve can be steep when it comes to doing unusual things in a game engine. This is, however, more often an issue due to lack of examples and tutorials than actually not being possible.

Even if naïve approaches are straightforward to prototype and displaying

[5]Shared in personal conversation between Patric Ljung and Tim Biedert (NVIDIA). A book with a chapter discussing this is forthcoming (VR Developer Gems).

some objects in game engines is easy, advanced rendering and getting efficient representations requires more work and to be fully committed to one specific game engine [1].

On top of that remain some issues about data management functionality and data processing pipelines not natively present in game engines. Using existing libraries for data management or naturalizing this functionality in game engine's programming paradigm could be a way to solve this issue.

An effort from the scientific visualization community could be done to provide tutorials and examples for typical features commonly used in visualization applications, providing insight on dealing with dynamic data, generating variable geometry based on them or implementation details on advanced rendering techniques in commonly used game engines.

Conclusion: The first obvious reason why scientific visualization community would want to use game engines is the ease of development. Setting up a game engine only requires several minutes to have a working solution to work on, and that can be deployed on a large set of systems (all OSes, phones, HMDs, exotic devices...). Game engines provide a framework with a large set of rendering features like high quality shadow mapping or an HDR pipeline that can be enabled or disabled at nearly zero cost of development. They also provide support for recent technologies (various HMD devices, NVIDIA RTX ray tracing technology, as of Unreal Engine 4.22) a short time after the availability of these technologies. It would usually require a significant amount of development time to integrate these features in a custom-made framework. Also, some game-related features like advanced network stacks, video and audio playback streaming, and cutting-edge asset management appear to be really useful and relevant for scientific visualization purposes. Moreover, this large set of features can be combined with plugins developed by the large community that range from paid assets to free, open-source plugins. Even if game engines lack some features for a specific application, they usually provide flexibility and building blocks that can be used to build your own plugin. On top of that, game engines provide a large set of cutting-edge profiling, debugging and testing tools that would also take a significant effort to develop.

In conclusion, several examples of successful visualization tools using game engines can be found. These examples show multiple benefits and can be seen as proof that no technical issue prevents researchers to use such tools for scientific visualization.

How to get funding for visualization software development and sustainability?

Katja Bühler, Eduard Gröller, Shintaro Kawahara, Barbora Kozlíková, Michael Krone, Krešimir Matković, Kenneth Moreland, Valerio Pascucci

Background and Motivation: In this breakout session, we discussed the situation and problems related to funding the development of a sustainable visualization software originating from research projects.

Funding is one of the most painful and sensitive parts as it influences the direction our research and development take substantially. One of the main problems is that software development (the engineering itself) is often not considered part of the funding. In the context of basic research, it is currently very hard to argue a substantial budget for software development. This often leads to rejection of the proposal or budget cuts, and software development is usually one of the first things to be cut. However, software prototypes are usually one of the main outcomes of the project and the outcome of most use to domain users. It is of utmost interest to turn such prototypes into usable software tools. This requires getting through the "valley of death", which is a common problem in software development in general. This step requires more engineering than research skills, but it is necessary if we want to make our scientific outcomes available to a larger community of users.

An intrinsic part of the funding issues is intellectual property, particularly when working with industry. This is very dependent on the local laws and rules of the partner organizations. This problem is even more complicated when the collaborating organizations are coming from different countries. Here arise questions like: "Who is the owner of the software code developed by a student at the university?", "Who checks for copyright and patent infringements?"

There are already some good examples of visualization tools being financially supported from different sources. The ideal case is when a company is interested in pushing the technology into the research community and allocates funding for that. This was the case of MegaMol, VisIt, and ParaView each getting funding from Intel to integrate their OSPRay technology. However, these are rather rare cases. For ParaView, the success is based on the fact that Kitware has structured their business model around supporting open source software with clear research focus. Another example can be the VMD tool, which got basic funding for development of stable and sustainable software from NIH. In US, the current situation is that NSF is becoming more sensitive to community software and NIH invests more resources ito software development than NSF. In Europe, there are already initiatives leading towards funding of software sustainability (Wellcome Trust and Software Sustainability Institute in the UK, DFG in Germany, H2020 projects in the EU). But as these are still first attempts, it is hard to foresee their impact.

But how can we obtain funding for software development when all project partners are coming from the academic environment? In this case, we evidenced examples when the partnering organization was applying for funding for development, seeing the software as the enabling technology and substantial part of their workflow. These resources were then used for the development of the visualization tool by another project partner. Another model could be to use the seed funding to turn the prototype into usable software, which leads to establishing spin-off companies (for example US Small Business Seed Funding and SME Instrument of H2020 in EU).

Challenges: In brief, the problem with funding the development of visualization software produced during research projects is that our funding sources most typically do not make funding decisions or evaluate projects based the quality of software produced. Rather, funding sources are principally interested in the advancement of science, which is usually measured in terms of scientific

publications. The quality of any software produced is considered secondary, if considered at all.

As argued throughout this report, visualization research has a much larger impact when the artifacts of that research include quality software leveraging the (published) innovations. Furthermore, when visualization teams partner with domain scientists, a common and beneficial practice, the domain scientist partners expect some kind of usable product. However, a project that dedicates more resources to producing quality software that ultimately has a larger impact may have less time for scientific innovations. In such a circumstance, a project with a larger impact could be reviewed less favorably. Balancing the needs of core research, which is directly reviewed, and ancillary software development, which is required for the research to be viable but not directly reviewed, is challenging.

Consequently, many universities and laboratories stop development in the prototyping stage. Since research funding does not include software development, software is only constructed to the point to conduct necessary analysis for the research. Further problems might lower the motivation of performing further software development. The researchers might feel that the software being developed is only useful for their small research community or the economy of scale for the software is otherwise limited.

Another practical consideration when acquiring funding for visualization software development is the added complication of intellectual property. Rights to prototype software that lives only as long as a specific research project requires it is of little concern. However, once an investment is made to make the software useful to a broader audience, a market is inherently built and the rights to the intellectual property become important. Several entities — funding sources, universities or laboratories conducting the development, individuals, collaborators, and other stakeholders — may all have valid shared rights to the intellectual property. Disagreement can prevent the dissemination of software regardless of its quality or value. Ultimately, the funding sources are in the best position to dictate the terms of intellectual property, and these terms should be considered at the onset before it is too late. For example, the german research foundation strongly encourages open-sourcing research software resulting from their funded projects.

Research software is never really finished. Good research software will continue to support and grow with subsequent research. This requires software not to be an inert program but rather a continuously evolving software system. As the software grows with new research, it must also simultaneously be managed and supported like any other software project by keeping up to date with technology changes, fixing bugs, supporting users, managing sustainability, and providing documentation. All of this requires considerable commitment from a funding perspective.

Conclusion: To put it succinctly: Software needs development time, and development time is not free. Without the support of funding that can be used for taking research ideas and making usable software, our visualization community cannot truly grow.

Ultimately, making software for visualization research is easier today than it was 20 years ago - in large part because past research from forward-thinking

researchers has produced software that can be leveraged for new visualization research. This is a case to be made to funding sources: It is cheaper in the long run to pay to support quality software development than to pay for research that starts from scratch over and over again. This also extends to discoveries in other scientific domains. The field of visualization is most important as an enabling technology for scientific discovery, so funding sources should be made aware that including funding for software development can have far-reaching impact across many scientific domains.

As researchers, we have limited influence on our funding sources, so we may find it difficult to get direct funding for software development and support. However, we do note some "tricks" that help software development become a symbiotic part of a successful research project. It is possible to get funding in disguise by padding cost estimates for software development. A research proposal starts with a budget from the proposer, and as long as the reviewers agree that funding is in scope then software development can be secured. Of course, such a strategy is at risk of competing with more aggressive budgets. Another possibility is to use students, including undergraduates. This is a cheap way to increase productivity, but students are less experienced programmers and the quality (especially with regard to sustainability) of software can vary greatly. Furthermore, students will not be around to support the software in the future. Finally, if your project is successful enough, it may also be lucky enough to have passionate users that will provide or help get funding.

Because software development is always constrained, developers on research teams should take advantage of any available tools to assist in development. There are several automated tools for documentation and formatting. It is an interesting line of research to improve automated tools that could potentially highlight documentation that is incomplete or wrong or, better yet, automate the writing of the documentation.

It is a consensus of the workshop participants that one of the best ways manage visualization software development under limited funding from research is to build a community around the software. A community is a collection of groups with similar research goals (who may or may not have shared funding) that loosely collaborate by collectively contributing to and using a shared software system. The primary advantage of such a community is that the development cost becomes amortized across the groups, which can make the problem of support more manageable. We anecdotally note that visualization groups that have built a community around shared software are often successful in finding funding. We believe these communities to be a virtuous cycle with the software base making the research more productive, the research adding functionality to the software, the added functionality increasing the notoriety of the software, the added notoriety helping those involved securing new funding for the next research project.

We note that building a community is not easy, and much thought should be given to make it successful. Before starting, it is worthwhile to perform informal "market research" to determine the potential community size. Too small of a community will not support the software well. Additionally, these communities work best around software systems with open licenses, which helps prevent issues with intellectual property. Thus, it is best to agree upon open licenses.

How do we evaluate vis software? What makes software successful or unsuccessful?

Hank Childs, Takayuki Itoh, Shintaro Kawahara, Steve Legensky, Krešimir Matković, Kenneth Moreland, David Pugmire, Allen Sanderson, Manuela Waldner

Background and Motivation: The visualization community has provided a large amount of work to the broader scientific community that has impacted the understanding of a variety of scientific and engineering processes. However, there is a sense within the community that our ability to evaluate these visualization software tools and techniques is incomplete. Because of this incompleteness, our ability for introspection, identification of strengths, weaknesses, and gap analysis is lacking.

Sectors of the community have the ability to measure and evaluate the success of visualization software. In the research community metrics like citations and publications are a clear means of evaluation. For commercial software, where costs and benefits are closely monitored and measured, the evaluation of a software is directly related to its ability to provide value to the customer base (and revenue for the developer), and can often be quantified with a specific price. However, in other sectors the ability for evaluation is much less clear. These are often ad hoc, and the result of taking the pulse of the community on what has been adopted as a "standard", or the longevity of a software. Examples of this include the emergence as VTK as a de facto standard in the scientific visualization community. The same applies to the use of a transfer function for volume rendering of particular layers in medical data. Journal covers with visualization, or a scientific impact highlight provided to funding agencies are another means of providing a sense of value. But these metrics are not clearly defined, and very difficult to articulate. These ad hoc methods for evaluation point to a clear lack in community understanding, and could limit our ability to effectively utilize resources for scientific impact.

Community norms on evaluation of visualization software would provide guidance on how to maximize the impact of work in the scientific community and help forge stronger scientific partnerships for increasing the frontiers of science. These norms would also help the community understand what has been successful and should be replicated or explored, as well as understand shortcomings and under-served areas that need to be addressed. This guidance could help funding agencies and industry to more effectively use available resources and as a means for identification of gaps and areas that need further exploration. Researchers addressing pieces of the larger problem would have guidance in understanding how the resulting software artifacts could have lasting impact.

Challenges: This is a fundamentally hard problem primarily because it requires expertise outside of the traditional training for computer scientists. If the ultimate goal of quality software is impact with users, the computer science community is ill equipped to understand, measure and analyze these metrics. Providing impact to users requires an understanding of a wide array of disciplines, including visual perception, human learning, psychology, neurology and cognition. It is a challenge to assemble the right team of people to study and evaluate these

issues. User studies are a means to understand this issue, but good user studies can require a large number of resources, a large amount of planning, and are difficult to carry out. Further, user studies often involve human subjects, which increases the complexity and costs of each research proposal.

In cases where quantitative metrics are available (e.g. citations, adoption rates, inclusion of libraries into user tools), it is still difficult to tie these metrics back to the fundamental criterion: impact on science. Stated another way, it is difficult to correlate frames per second in data exploration to the human cognition processes that leads to understanding of new scientific processes.

Conclusion: The group agrees that establishing norms for software evaluation would provide tremendous value to the visualization community. These benefits would provide a clear indication of what constitutes successful visualization software and would establish a means for identifying and replicating success, as well as the ability to identify and understand areas that need to be addressed. We feel that the establishment of these norms would provide a way for the community to produce software that is has more impact on the scientific community.

Good effective user studies would be a benefit for evaluating new visualization techniques; researchers would not only have a way of evaluating the technique but also have known design goals up front (i.e. evaluate existing techniques before developing new ones). This requires funding and collaboration with "experts" in designing and performing such studies. These benefits would include an increase in user impact, advancing the state of the art, and better investment at multiple levels (funding and research).

Finally, having an environment in which techniques could be evaluated would allow researchers to quickly evaluate new ideas in a formal and informal manner manner which would benefit the development in terms of time, insight, impact, and for pursuing funding.

Can we define a taxonomy of visualization software?

Katja Bühler, Hank Childs, Johannes Günther, Eduard Gröller, Shintaro Kawahara, Barbora Kozlíková, Krešimir Matković, Valerio Pascucci, David Pugmire, Guido Reina, Ivan Viola, Manuela Waldner, Xiaoru Yuan

Background and Motivation: There is a lot of visualization software available on the market and in research. Partially, these software systems are solving similar problems, but practitioners are not always aware of the details in terms of system design, algorithms, or implementation. Building a taxonomy of visualization software has the following benefits for researchers, developers, students, and end users:

- It enables better communication between practitioners as it describes the space of problems we are trying to solve.
- It can serve as scaffold to build a database of visualization software, which can help practitioners find existing visualization software that fulfills their requirements.

- Such a database does not only represent the space of solutions (what has been done) but also a space of problems (what is left open) to identify "white spots" for new research.
- It can serve as blueprint for the design of a new visualization software.

Taxonomies are often used in visualization to classify existing visualization techniques, but also in the field of software engineering. To the best of our knowledge, there is no taxonomy describing visualization software. However, such a taxonomy could have a considerable benefit for the community for the reasons listed above.

Challenges: Building a taxonomy of visualization software is not a trivial task. There are numerous visualization tools that are used in many scientific domains. Further, there are also visualization frameworks or libraries that are used to build sophisticated solutions. There are numerous open questions, such as, for example, should these highly specialized libraries also be included in the taxonomy? How to structure the taxonomy? How to cover different perspectives of the taxonomy users? An end user needs different information than a software developer. How broad should such a taxonomy be? Should it only include systems for scientific visualization or be more inclusive?

The main challenges are how to build a good and useful taxonomy, and once it is done, how to communicate it to the target audience so that it is really used. Having an unused taxonomy would be a failure. With strong support from the community, it will also be possible to build a database of existing software systems that can have a considerable impact by serving as standard repository of available visualization software systems.

While developing the taxonomy we always keep in mind its potential users: visualization software users from various application domains who are looking for a visualization solution, visualization software developers who are supposed to build efficient tools for the first group, the visualization researchers who need to build their research tools in order to verify new methods, teachers facilitating students their first contact with visualization, and, finally, the visualization library developers who support the second and the third group. Each of these taxonomy users has different primary interests. It would certainly be possible to develop four taxonomies, but we think that our community needs a unified one. A unified taxonomy can help the different stakeholders communicate efficiently. Split taxonomies would just further increase cross-domain communication problems.

The level of detail in the new taxonomy represents a further challenge. In order to build a successful and frequently used taxonomy, the level should be not too shallow (the taxonomy would become trivial) or too deep (too many details would easily make any comparison impossible, people would also shy away from classifying their own framework because of the effort). Further, as we deal with a software there are many general software development issues. Which programming language is used, is a form of scripting supported, is it a web-based application, does it run on parallel machines, etc. Of course, all of these aspects are valid for the visualization software (as for many other types of software). Covering them in full detail, however, would make the taxonomy too complex and probably useless. We decided to focus on visualization software specifics and cover general aspects only to a certain degree. The idea is to use available general software taxonomies if an extension is needed.

Method: In the course of this workshop, we established a draft taxonomy describing the design space of visualization software. We split this taxonomy development process into several steps:

- Stage 1 (day 1): Establishing a list of key words describing visualization software in a brainstorming session with 12 participants. The final list contained 90 key words.
- Stage 2 (day 1): In three four-person sub-groups, we performed a first classification of these key words in parallel. Additional key words were added if considered necessary.
- Stage 3 (day 2): In the large group, we merged and refined the initial classifications into two hierarchy levels. This preliminary classification is described below.
- Stage 4 (day 2): Smaller sub-groups created fine-grained sub-categories for the top-level categories.
- Stage 5 (future): Existing software visualization will be classified using the taxonomy. The classification of existing software will probably result in changes or a refinement of the taxonomy.

Preliminary Results: Our draft taxonomy contains the following six high-level categories with their sub-categories in brackets:

Impact Goals (target audience, application domain, purpose): captures the goal of the visualization software that is defined a-priori, such as whether it is an expert system or rather targeted towards a general audience, what is the anticipated application domain, or whether the purpose is to support exploratory analysis, presentation, or other high-level goals.

Data (input, output, model, acquisition): describes the type of the data the software is handling, such as spatial data or time-varying data, which kind of output data is generated, such as images or movies, which data model it uses internally, and whether the data is acquired in situ or post-hoc.

Algorithms (processing strategy, functionality): describes which algorithms the system employs to process the data, such as streaming, single-pass, out-of-core, and which visualization pipeline algorithms are applied to the data.

System design (hardware architecture, deployment strategy, software architecture, implementation): captures how the system is designed in terms of hardware (e.g., whether it is a distributed system or GPU-based system), how it is deployed (as full system or library-based), in terms of software architecture (e.g., client-server), and specific implementation including the platform, languages, and graphics API.

Project management (development strategy, licensing, funding, assessment and reporting, requirement gathering strategy): This category includes project management tasks of software development in general, which are assumed to have a strong influence on visualization software in particular,

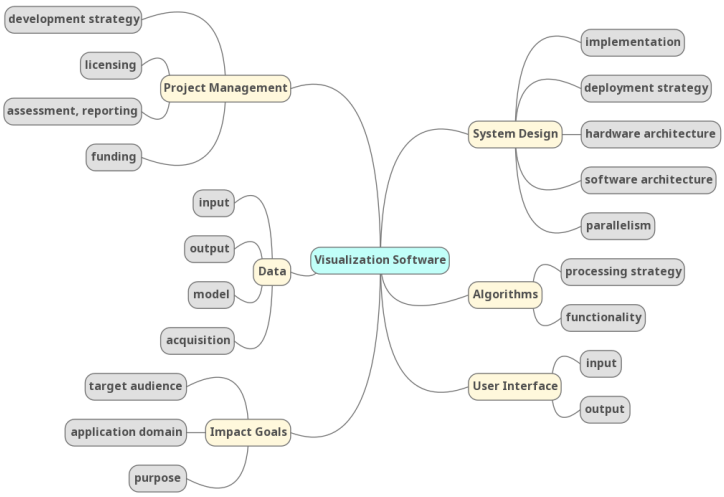

Figure 1: Top-level taxonomy categories

such as whether the system is open source, whether development is crowd sourced, and which kind of documentation is available.

User interface (input, output): describes how the user interacts with the system, i.e. how the user can provide input, such as using conventional mouse and keyboard, command line interfaces, or multi-modal interfaces combining gesture and speech input, and how the output is presented to the user, such as on a conventional screen in contrast to immersive environments like VR or AR.

The first top-level taxonomy categories are visualized in Figure 1.

Conclusion and further steps: The group of visualization experts agreed that there is a definite need for a comprehensive visualization software taxonomy and associated database of visualization software systems. They also agreed that the currently identified topics serve as a valuable basis for finalizing the taxonomy.

The further steps include finishing the taxonomy, and then using it to classify existing visualization software. We will start with the software that the present group is using and developing. After this initial step, the taxonomy will probably be refined. We plan to invite the whole visualization research community to classify their software then. The available mailing lists, such as IEEE_vis, or vis-master, will be used to reach the visualization researchers. The taxonomy might further evolve during this step. Finally, we plan to publish a paper on the final taxonomy with a relatively large base of software tools classified. The taxonomy will then be advertised to a broader community through the mailing lists and through our collaborators in various domains.

We are sure that the new taxonomy will become a valuable resource for the visualization community. We also envision that we will keep further developing it as the software development and the IT technology in general changes. It is clear that no software taxonomy in today's world can last forever, but it is also

clear that our community needs a taxonomy, and we are positive that we are on the right track to develop one.

List of Participants

- Xiaoru Yuan, Peking University
- Kenneth Moreland, Sandia National Labs
- Claudio Silva, New York University
- David Pugmire, Oak Ridge National Laboratory
- David DeMarle, Kitware
- Steve Legensky, Intelligent Light
- Allen Sanderson, SCI / University of Utah
- Barbora Kozlikova, Masaryk University
- Ivan Viola, King Abdullah University of Science & Technology
- Timo Ropinski, Ulm University
- Markus Hadwiger, King Abdullah University of Science & Technology
- Eduard Gröller, TU Wien
- Katja Bühler, VRVIS Research Center
- Johannes Günther, Intel Corporation
- Valerio Pascucci, University of Utah
- Krešimir Matković, VRVIS Research Center
- Patric Ljung, Linkping University
- Manuela Waldner, TU Wien
- Yun Jang, Sejong University
- Shintaro Kawahara, JAMSTEC
- Xavier Martinez, IBPC / CNRS

Meeting Schedule

	Monday	Tuesday	Wednesday	Thursday	Friday
7:00:00 AM					Check-out
7:30:00 AM	Breakfast	Breakfast	Breakfast	Breakfast	Breakfast
8:00:00 AM	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"
8:30:00 AM					
9:00:00 AM	Opening	Organization	Organization	Organization	Summary
9:30:00 AM	Lightning Talks	Breakout#1	Breakout#4	Breakout#7	
10:00:00 AM					
10:30:00 AM	Break	Break	Break	Break	Break
11:00:00 AM	Lightning Talks				Future Work
11:30:00 AM					Wrap-Up
12:00:00 PM	Lunch	Lunch	Lunch	Lunch	Lunch
12:30:00 PM	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"
1:00:00 PM					
1:30:00 PM	Lightning Talks	Group Photo	Breakout#5	Excursion	Seminar End
2:00:00 PM		Breakout#2			
2:30:00 PM					
3:00:00 PM					
3:30:00 PM	Break	Break	Break		
4:00:00 PM	Topic Refinement	Breakout#3	Breakout#6		
4:30:00 PM					
5:00:00 PM					
5:30:00 PM					
6:00:00 PM	Dinner	Dinner	Dinner		
6:30:00 PM	Cafeteria "Oak"	Cafeteria "Oak"	Cafeteria "Oak"	Main Banquet	
7:00:00 PM					
7:30:00 PM	Socializing	Socializing	Socializing		
8:00:00 PM					
8:30:00 PM					
9:00:00 PM				Socializing	
9:30:00 PM	(open end)	(open end)	(open end)	(open end)	

References

[1] K.-I. Friese, M. Herrlich, and F.-E. Wolter. Using Game Engines for Visualization in Scientific Applications. In P. Ciancarini, R. Nakatsu, M. Rauterberg, and M. Roccetti, editors, *New Frontiers for Entertainment Computing*, pages 11–22, Boston, MA, 2008. Springer US.

[2] M. Wagner, K. Blumenstein, A. Rind, M. Seidl, G. Schmiedl, T. Lammarsch, and W. Aigner. Native Cross-Platform Visualization: A Proof of Concept Based on the Unity3d Game Engine. In *2016 20th International Conference Information Visualisation (IV)*, pages 39–44, July 2016.