DIFFERENT APPROACHES TO VISUALIZING SIMULATIONS OF WELDING PROCESSES

L. KESSELBURG*, S. WARKENTIN*, O. MOKROV*, R. SHARMA*, U. REISGEN*

**RWTH Aachen University, Welding and Joining Institute, 52062 Aachen. Germany*
Kesselburg: https://orcid.org/0009-0008-7556-9644, lukas.kesselburg@isf.rwth-aachen.de;
Warkentin: https://orcid.org/0009-0004-4514-8962, sergej.warkentin@isf.rwth-aachen.de;
Mokrov: https://orcid.org/0000-0002-9380-6905, oleg.mokrov@isf.rwth-aachen.de;
Sharma: https://orcid.org/0000-0002-6976-4530, rahul.sharma@isf.rwth-aachen.de;
Reisgen: https://orcid.org/0000-0003-4920-2351, uwe.reisgen@isf.rwth-aachen.de

DOI 10.3217/978-3-99161-089-2-031, license CC BY 4.0

ABSTRACT

In modern industry and research, vast amounts of data are being generated continuously, both through real-world measurements and numerical simulations. In the case of numerical simulations, particularly transient simulations, this creates significant challenges for human comprehension. Consequently, there is a critical need to create expressive visualizations while keeping computational resource requirements manageable. These visualizations can range from simple images to videos, and ultimately to interactive renderings, depending on the specific use case. Furthermore, a more established simulation system can offer a highly integrated interface, which allows the user to parameterize and schedule new simulations in addition to the interactive visualization of running and archived simulations.

The requirements for effective visualization change depending on the end-user's needs. On one end of the spectrum, developers and researchers involved in the data generation process require full access to all raw data, often coupled with sizable computational resources. This access facilitates early validation, error tracing, exploratory analysis, and further enhancement of the visualizations themselves. However, this configuration often necessitates a more intricate setup process. On the opposite end, reviewers, researchers, or executives generally require straightforward access to a refined version of the data, accompanied by an optimized and stable visualization pipeline. This enables tasks such as reviewing scientific publications or making data-driven decisions in a business context.

In this paper, we explore various visualization strategies tailored to meet the diverse needs of these two user groups, with a particular focus on simulations of welding processes. By demonstrating these strategies, we illustrate the trade-offs between computational resource demands and usability. We show two variations of the Server-Client-Architecture. In the first case, the client is a desktop program, presenting the user with a wide variety of options for analyzing and visualizing data, while utilizing computational resources from the client or the server. In the second case, the client is a web-browser, with the main benefit of removing any setup required. Additionally, we want to show an approach that utilizes WebAssembly to execute the full visualization pipeline in a web-browser. This includes loading the data, filtering, mapping and rendering. While this approach is more constrained in terms of data volume and processing power, it aims to enhance exchange between institutions through its ease of use and ease of deployment.

To further emphasize the interchangeability of simulation results, the open vtk file format is used, as it is widely utilized in academia. This allows us to leverage several KitWare software programs and libraries, thereby streamlining the development process by providing advanced visualization tools, reducing the necessity to work directly with raw visualization primitives, thus enabling the reuse of code across different strategies.

Keyword: software architecture, visualization, interactive 3D visualization

PROBLEM

In the context of welding simulation in research and industrial practice, various software solutions generate substantial amounts of data. We have identified four types of interactions when dealing with this data:

1. Online debugging
2. Internal discourse among colleagues
3. Conference presentations or publications
4. Research partners in industry

The exchange of data between these types of interactions poses a significant challenge, particularly in the everyday use of software solutions. Our results, which represent only a small portion of the data generated, are presented in 2 other publications [1][2] at this conference. The simulation software in use, namely SPlisHSPlasH [3], has the capability to export Particle Data into two different formats. One of these formats is the legacy vtk format (version 4.1). The format is relatively straightforward, comprising metadata in text form interspersed with binary data. A notable disadvantage of this file format is its inability to compress data, which can result in inefficient utilization of storage space and bandwidth. This results in a substantial accumulation of raw data, even for rudimentary experiments. The number of particles in our smaller simulations is approximately 150,000. However, for more complex setups, this number can increase to one or two million particles. As previously mentioned, the vtk file format is capable of storing at least the ID and position of each individual particle. The position of each particle is expressed in terms of a x, y, and z coordinate. Additionally, we store supplementary attributes; however, for the sake of simplicity, we will only consider the storage of a single attribute, such as the temperature of each particle. Therefore, it is necessary to allocate storage space for five floating point numbers per particle, with each of them requiring four bytes of storage. This amounts to a total of $150{,}000 \cdot 5 \cdot 4 = 3{,}000{,}000 \approx 3[MB]$. In practice, the amount of storage required for a given file is approximately twice as large, due to the necessity of internal metadata in the file format. It should be noted that these effects are only observed at the level of a single timestep. In this configuration, the export of 400 timesteps per simulation is typical. Consequently, a rudimentary simulation necessitates storage capacity that exceeds 1 GB on disk.

The interpretation of raw binary data interleaved with text is not a viable option due to the lack of comprehensibility. The objective of this study is to facilitate the comprehension of the contained information through a more effective visual presentation. In this particular context,

the term “useful” can be defined in a number of different ways. It must be acknowledged that not all observers of the generated visualization are seeking the same insights. Depending on the role of the viewer, there are different requirements. We have identified 4 different roles and corresponding perspectives for looking at a visual representation of the data.

Online debugging

The first role is that of an individual software developer or researcher who is attempting to enhance the overall simulation, either by investigating the algorithms used for the simulation or the parameter set in use. This use case necessitates a high resolution in both the spatial as well as the temporal dimension in the case of transient simulations. It is often advantageous to have this information at one’s disposal during the course of the simulation process, to decide early on whether a certain simulation should be allowed to continue, subject to minor changes or terminated, due to useless results. Moreover, it is helpful to have access to as much of the simulation state as possible. This encompasses not only definitive outcomes but also intermediate results.

Internal discourse among colleagues

The second use case concerns the internal discourse among colleagues, facilitating the comparison of results and the planning of forthcoming simulation runs and the enhancement of previously implemented algorithms. In this case the fundamental prerequisite is the capacity to quickly gain a comprehensive understanding of the entire simulation, which means we still need a high spatial and temporal resolution as before. A particular emphasis is placed on the capacity to effortlessly navigate through the various time frames of a transient simulation. Conversely, intermediate results or values of a single simulation step can often be disregarded. A comprehensive evaluation of central properties often suffices to determine the success of a single simulation. This evaluation can be informed by the insights of senior researchers or by reference experiments conducted in the real world.

Conference presentations or publications

The third use case is encountered when the conducted research is finally about to be published to a broader scientific audience. In the context of concluding a research project, it is often well understood which properties of the simulation are supposed to be presented. Consequently, it is straightforward to decide which properties to visualize. A further complication arises when one attempts to preserve the complete spatial and temporal resolution of the simulation. At this point in the process, traditional paper-based publications are proving to be a somewhat limiting factor. The limitation to static images poses a significant challenge in showing the simulation state changing over time or from different angles. This can be mitigated by employing multiple images instead of just a single one. However, the number of discrete

points in time or angles from which the simulation is displayed remains fairly limited. Moreover, this frequently comes at the cost of making the individual images smaller, thereby reducing the spatial resolution. In such circumstances, presenting the same results at a conference can be of significant benefit, as this allows for showing video footage in addition to still images. This is a substantial enhancement for transient simulations. However, it does not allow for complete interactivity because the speaker decides what is shown. This requires an understanding of the audience's exact level of knowledge.

RESEARCH PARTNERS IN INDUSTRY

The last use case arises when working with industry research partners. Similar to the previous case, the goal is to demonstrate the highest possible spatial and temporal resolution, albeit often only for a few core properties such as temperature, enthalpy, and stress. The main difference is that, unlike the previous case, the visualizations are required on an ongoing basis for newer and older simulation runs to facilitate a richer exchange between research partners, rather than at one distinct point towards the end of the research process to show final results. These requirements have proven especially difficult in the past.

As we can see the requirements vary based on the purpose of the simulation. This makes finding an ideal way to visualize the results difficult. Therefore, we will present a few different visualization tools that have proven useful in our work. Most of these tools are built around software provided by Kitware, particularly the “visual toolkit” library. This library is very versatile and offers a wide range of possible visualizations.

SIMULATOR

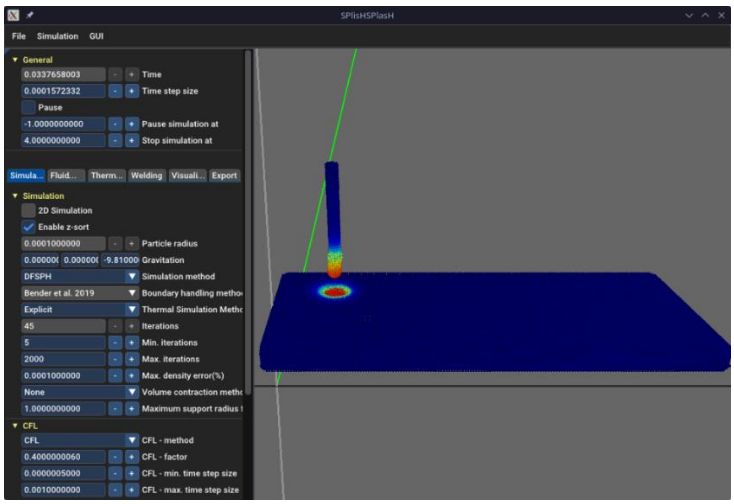

Fig. 1 Screenshot of the SPlisHSPlasH simulator with parameter menu on the left

The first option a user of the SPlisHSPlasH software will probably encounter to visualize the simulation state is built into the program. The included graphical user interface (GUI) can be seen in Fi. 1. It is enabled by default when the program is running. Although it can be disabled to allow running simulations on systems that do not have the ability to provide graphical output. The biggest benefit of this visualization is that users have access to all particle attributes. This offers the most possible insight into the current simulation state. However, users can only inspect the simulation state of the current timestep. This is because the state of previous time steps is stored but cannot be reread by the simulation. This visualization method is most useful for the first use case. An individual software developer or researcher can quickly iterate on the algorithms and their corresponding parameters and view the results as they are computed. However, this is not a good fit for the other three use cases. This is because only the current state is available, which is not saved permanently by default.

The SPlisHSPlasH simulation software can be configured to regularly export particle attributes of interest. The particle IDs and positions are always exported. From the remaining set of particle attributes, the user can select an arbitrary subset. Each chosen attribute is exported for each particle. This feature is independent of the visualization provided in the GUI. The provided visualization cannot be used to visualize the exported results afterwards. The exported data serves as the basis for all other visualizations.

PARAVIEW

We predominantly use ParaView internally for loading, analyzing, and visualizing exported vtk files. ParaView is KitWare's "[…] open source, multi-platform data analysis and visualization engine." [4]. It provides a simple yet flexible interface to the Visualization Toolkit (VTK), a library developed by KitWare as well. The VTK library offers a wide variety of file readers, filters, and visualization methods. In addition to ParaView's interactive capabilities, ParaView enables us to export images of static simulation states or videos to visualize transient results.

One feature that we use extensively internally is the macro system, which is conceptually simple yet powerful. Macros reduce the setup time for visualizations used repeatedly with different sets of input data. When recording a macro, ParaView tracks all user interactions and translates them into calls to its built-in Python application programming interface (API), which are collected and result in a file containing Python code. This is useful not only because we can record and replay any interaction with the program, but also because recorded interactions are easy to share between colleagues. By sharing the macro files using a version control system (VCS), we can also track changes made to the macros.

As a first step, users usually use ParaView on their local systems to visualize data generated from previously run simulations. We have done this in the past as well. First, we would run a SPlisHSPlasH simulation with particle export enabled. Then, we would use ParaView to visualize the results while the simulation is still running or after it has finished. However, this became problematic when we started running the simulation on remote systems. We did this to utilize stronger computational resources and run multiple simulations simultaneously or to run simulations while still having our personal systems fully available.

Some systems used for remote simulations lack a GUI. It is not possible to use ParaView as a standalone application on those systems. One option is to copy the generated files from the remote system where it was generated. Then, copy it to a local user system where ParaView can be used. However, this process is not suitable for long-term usage and should be avoided for two main reasons. First, copying the simulation results between systems adds an extra step to the development process, which degrades the overall development speed. Second, it avoids duplication of simulation results across systems. Over time, duplication of files can cause confusion. This occurs when certain files are deleted, renamed, or moved on some systems but not on others. Furthermore, duplicating identical files across systems consumes additional disk space.

Fortunately, this problem is often encountered in many different settings, so it is well-known at KitWare. ParaView's implementation solves this problem by following a client-server architecture. This is usually hidden from the user. If no extra steps are taken and ParaView is used as any other desktop application would be, both the client and the server start on the local system and connect automatically. In settings where the data to be visualized is located on a remote system without a GUI, the user must first start the ParaView server on the remote system in addition to running the normal desktop application on their local system before connecting the client to the server. Once this is done, the ParaView instance on the local system can access the remote system. The user then has access to the remote machine's file system. Data loading and analysis are executed on the remote machine, and the final image can be rendered on either the client or the server. In Fig. 2 the parts of the ParaView GUI that change after connecting to a remote system are marked in red. This includes the server address, the file path on the remote system, and the resource utilization of the remote system.

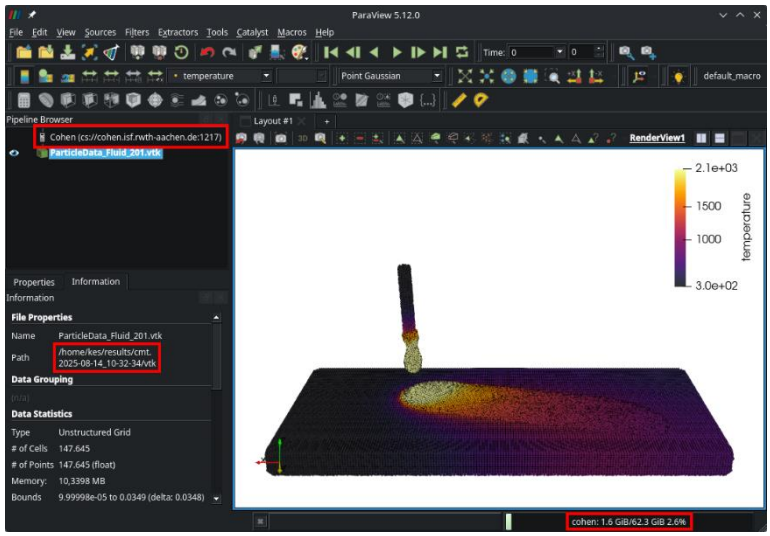

Fig. 2 Screenshot of ParaView. Sections marked with borders change when connected to a remote server. Upper left: Connection information, Left: File path, Lower Right: Remote Host and its resource utilization

The server component can also be used across multiple remote systems. This can be done to either process and visualize data distributed across different systems or to use different systems for initial data processing before rendering on a separate system with specialized hardware. So far, we have not needed to use either of these options. Additionally, it should be noted that KitWare discourages the latter use case. It was developed at a time when rendering hardware was harder to find and more specialized. Nowadays, the overhead of sending intermediate data is usually greater than the benefit gained from using specialized rendering hardware. This makes it difficult to justify using distinct systems for analysis and rendering tasks. [5][6]

Overall, ParaView is probably the most powerful visualization tool that we use. It is suitable for almost every visualization use case introduced previously, though not equally. The first use case is covered when ParaView is used to visualize states that have already been exported from a running simulation. Compared to using the simulator's visualization, it is not possible to access the fully updated state of the simulation at all times particle properties that were not marked for export. However, users have the ability to step through the exported timesteps at their desired speed. This makes it well-suited for the second use case, in which colleagues can inspect and compare the results of different simulations, either individually or as a team, to gain more insight. The third and fourth use cases are covered by the image and video export feature. It is difficult to offer ParaView's interactive capabilities to external users for several reasons, which are described in the next chapter.

TRAME

While the simulator in combination with ParaView is a sufficient solution for the first two use cases, ParaView is only a partial solution for the last two. ParaView can be used to create images or videos of the simulation that can be used in publications or correspondence with research partners. Second, it would be possible to exchange raw simulation results via a research data management platform for the third use case, or a conventional cloud storage platform for the fourth use case. Then, one could ask readers of our publication and research partners to download, install, and use ParaView with the provided data themselves. This approach seems feasible at first due to its simplicity. However, when examined more closely, these benefits quickly disappear. When moving files containing simulation results to external services, large amounts of data are duplicated again. Additionally, only some of our audience may have ParaView available, as most people do not have the option to install new programs in a corporate environment. Those who have ParaView available may consider using their ParaView installation as a client to a ParaView server running on our simulation server. However, this would require opening our firewall, which is problematic from an IT security perspective. Additionally, we would need a more elaborate ParaView server setup because, currently, each user needs to open their own session on the server. While this is not a significant effort it is an additional step requiring Secure Shell (SSH) access to the server.

In an earlier effort to address this challenge, it was decided that a more phased-out solution following the client-server architecture should be implemented. In search of a solution requiring minimal additional development effort, Trame was identified. Trame is also

developed by KitWare. According to their documentation, “Trame is an open-source platform for creating interactive and powerful visual analytics applications. Based on Python, and leveraging platforms such as VTK, ParaView, and Vega, it is possible to create web-based applications in minutes.” [7] While creating a software solution in “minutes” is an impressive claim, the use of Python, VTK, and ParaView seems appropriate, because Python is a familiar programming language for us, and we can leverage our existing experience with ParaView.

The core benefit of Trame is that it enables developers to implement all the necessary components of a visual analytics application from a single Python code base or even a single file, if desired. This is achieved by leveraging Python’s ability to seamlessly integrate code written in different programming languages. [8]

Trame follows a reactive state model. This means that changes to the application state on the server are automatically reflected in the user interface, and changes made through the user interface are automatically reflected in the application state on the server. Therefore, a developer using Trame as an application framework does not need to worry about synchronizing the state between the user interface and the application logic. Additionally, Trame provides simple mechanisms for reacting to events in the user interface. On the other hand, Trame provides a simple way to design the user interface. For more traditional components such as buttons, text inputs, and selection menus as well as for composing the layout, Vuetify and, by extension, Vue are used. In addition to the components one would expect from a UI library, Trame provides widgets to display visualizations from various widely used tools for 2D and 3D visualizations. The most relevant ones are the widgets that show views from VTK or ParaView. There are also widgets for Python libraries, such as Matplotlib, Plotly, Leaflet and a few others.

Unlike traditional web servers, Trame uses a long-running server process rather than spawning a new process for each incoming request. This allows Trame to maintain the state of the application and the different visualizations. As can be seen in Fig. 3 we kept the complexity of our user interface low. When starting up, the Trame application searches the server for available simulation results. These results can be selected from the list at the top of the drawer on the left. The other relevant controls are for playing back the timesteps of the simulation. in addition to a pause/play button, it is possible to jump to a specific timestep by entering its number in the text field, or to use the progress bar at the top to scroll through the available timesteps of the selected simulation.

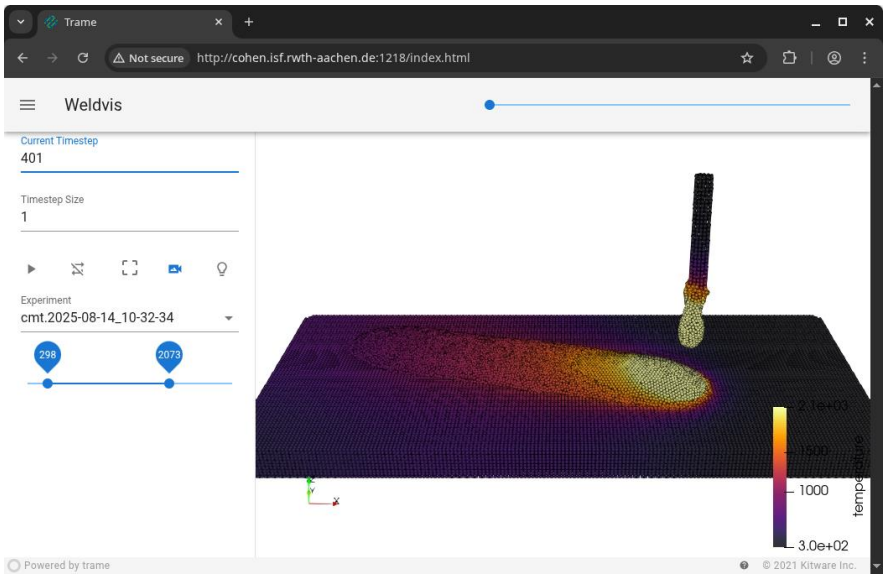

Fig. 3 Screenshot of Trame application

The central component of the user interface is the 3D view of the simulation results. Note that the resulting visualization is essentially the same as what we previously achieved in ParaView. We were able to reuse the macro recorded in ParaView and use its code to configure Trame's ParaView widget, resulting in an almost identical visualization. Although not apparent in Fig. 3, it is important to note that the 3D view is fully interactive. In addition to scrolling along the time axis, users can change their perspective on the data. To accomplish this, Trame employs a communication sequence that differs from that of traditional web servers. Fig. 4 provides a rough outline of this sequence. First, the application is downloaded and initialized. A WebSocket connection is opened, and a permanent server process is started. The client application continuously sends user interactions with the 3D view to the server. The permanent process continuously renders images based on the latest interactions. These images are sent back to the client and finally displayed to the user.

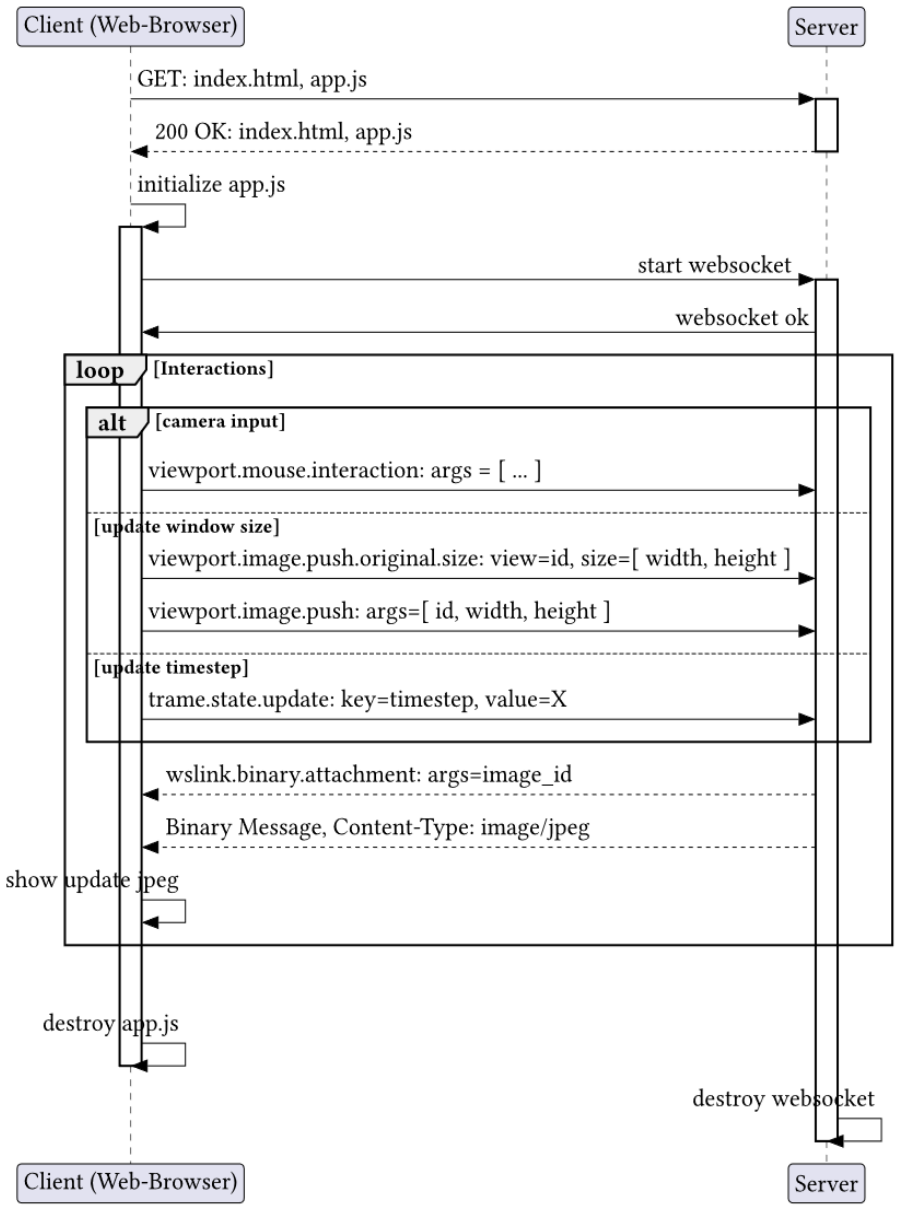

Fig. 4 Diagram of the Trame Communication between Client and Server

The resulting application seems like a good fit for the third and fourth use cases. The Trame application can be used to view all timesteps for different experiments. Additionally, each

state can be explored from different angles and perspectives. However, after using the application internally for some time, it became apparent that it would not be feasible to offer it to a broader audience on an ongoing basis. This is due to the administrative work that would be required. First, the Trame application would require opening ports in our firewall. Another problem is that the long-running server process is susceptible to extended unavailability caused by short-lived errors. For example, when trying to load simulation results that did not conform to the expected format, the server process would crash and become unavailable for requests of valid simulation results until the Trame application was restarted. This problem could be partially solved by using one of the more advanced Trame deployment options. The available options are:

- Using the application inside Jupyter.
- Building a desktop application with Tauri.
- Building a container image for the cloud.

However, all these options require additional software and increase administrative effort and reliance on external software. Another issue that the Trame application and ParaView in Client Server mode have in common is that the server hardware used by the visualization applications is the same hardware used to run the simulations, which leads to a conflict of interest. Viewing simulation results decreases the amount of computing power available for running simulations.

WEBASSEMBLY

After acknowledging that the Trame application was far from ideal, we searched for another solution that would require less administrative effort to support over time. The final solution we will discuss in this paper is based on a technology called WebAssembly. According to the official specification, "WebAssembly (abbreviated Wasm) is a *safe, portable, low-level code format* designed for efficient execution and compact representation. Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well." [9]

WebAssembly is a standard maintained by the World Wide Web Consortium (W3C). It was invented and standardized due to the desire to utilize more of the computing power available to JavaScript applications running in browsers than is usually possible. In addition to the core specification, many extensions have been standardized to increase the capabilities of a WebAssembly environment. The WebAssembly design process focused on the following properties:

- Efficient and Fast
- Safe
- Modular
- Support for a wide variety of programming languages

These days, it is possible to compile almost any programming language into WebAssembly. For systems programming languages such as C, C++, and Rust, this is achieved by using WebAssembly as the compilation target instead of traditional assembly languages such as x86_64.

As with Trame before, we are fortunate that we are not the only ones with a "[…] demand for powerful, interactive data visualizations directly within web browsers." [10] KitWare has also identified this demand. To meet this demand, KitWare has extended VTK to create versions that run in a WebAssembly environment. The resulting product "VTK.wasm", can be integrated into custom applications in various ways. For brevity's sake, we will only look at how we used it. To use VTK.wasm, we wrote a small C++ program that downloads a VTK file generated by the simulation. Then, it uses the traditional C++ VTK API to build the visualization pipeline. We can reuse the code from our previous attempt with Trame here because the interface provided by VTK is very similar to the one provided by ParaView. As Fig. 5 shows, the user interface is equally minimalistic compared to the Trame application.

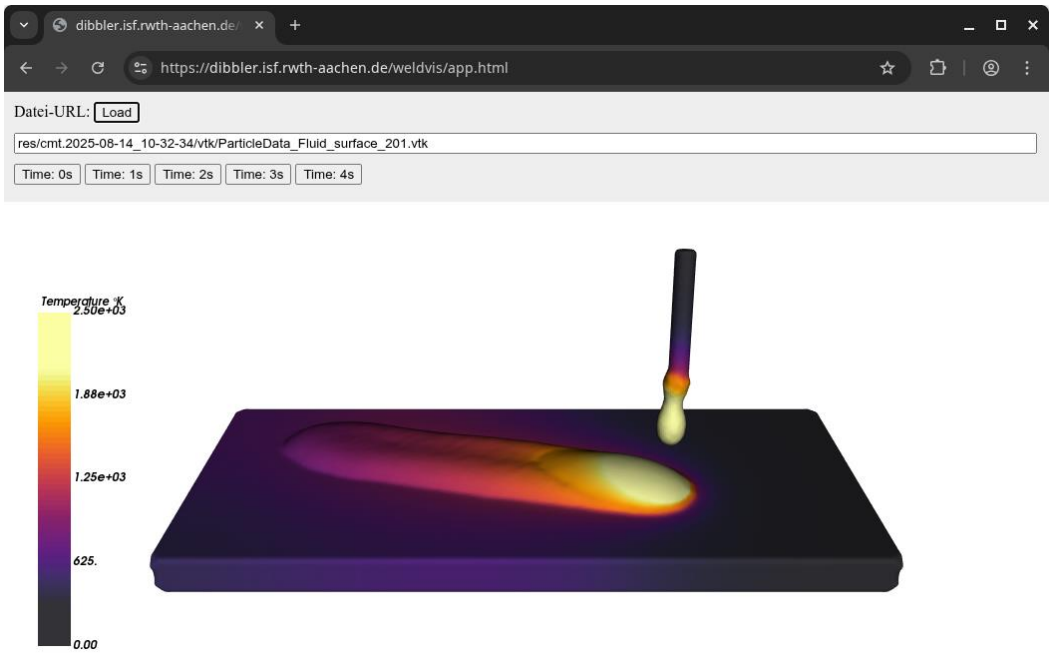

Fig. 5 Screenshot of VTK.wasm application

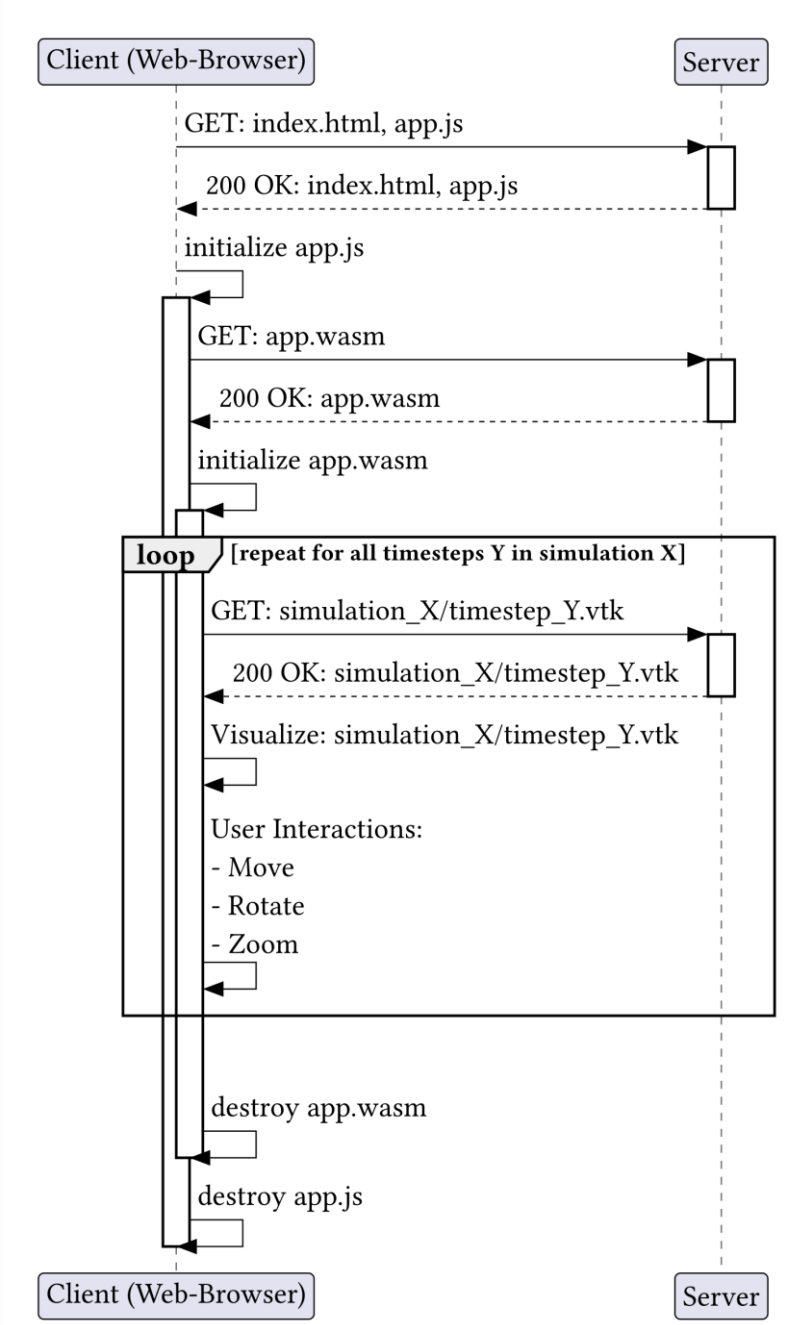

Fig. 6 Diagram of VTK.wasm Communication between client and server

Fig. 6 illustrates the process of downloading the application to the browser and rendering the simulation results. The process begins with the download of the index.html and app.js files (1. GET request). This is necessary because web browsers are not designed to execute WebAssembly on their own. The HTML file declares that the full space should be used as a drawing surface for the WebAssembly application. The JavaScript file is necessary to download (2. GET request) and initialize the WebAssembly program. The WebAssembly program is the actual application. It is responsible for downloading simulation results from the server (3. GET request). Then, the results are rendered locally in the browser instead of on the server. This is possible because the necessary parts of the VTK library have been compiled into WebAssembly instead of traditional assembly and are now running in the browser.

This has multiple benefits. For example, user interactions with the 3D view can be handled locally. No communication with the server is necessary. This is a significant architectural difference from the Trame application. We do not need a continuously running process on the server that tracks the application's state. All server interactions are simple file downloads. First, the application files are downloaded, followed by the data file. Therefore, to offer this application to users, we only need a web server that can serve static files. Since serving static files is the traditional task of web servers, a broader selection of ready-to-use web servers is available. The setup will also be much more familiar to most system administrators. Furthermore, this opens up the opportunity to utilize other features of traditional web servers, such as access control and caching.

We have not yet thoroughly tested the WebAssembly application, but it appears to be a good fit for the third and fourth use cases. Although it requires more development effort initially due to parts of the code being written in C++ and the immaturity of some WebAssembly-related tools, the resulting application is much easier for system administrators to manage since they only need a traditional web server to offer the application.

CONCLUSION

We have demonstrated how we use various components of the KitWare visualization stack to address different challenges that arise from the various ways and reasons for visualizing our simulation results. These components are designed to work together, or even on top of each other, making it easy to reuse code between applications. The biggest challenge when working with KitWare software is the documentation. Not because there is no documentation or too little of it, but because there is a lot of documentation that is fragmented. Every program or library has at least one dedicated piece of documentation and often more. For example, Trame has different sections for examples, guides, and the API itself. Additionally, some information can only be found in blog posts.

We will continue to use ParaView internally for the foreseeable future. However, we are interested in exploring WebAssembly further to share our results with a broader audience.

OUTLOOK

When developing the WebAssembly version into a more capable application, there are two possible directions: The first direction would be a fully integrated application that communicates with a central coordination server. This server knows about a full collection of completed simulations and the storage location corresponding to the results of these simulations. The client can then request the URLs for the files that constitute the result of a full simulation and download them on demand, before showing the visualization to the user. In this case the application server would follow a more traditional architecture where no state is kept between requests and the URLs for different results are stored in a database.

The other direction is simpler. In this direction, the application is designed to consist only of visualization logic and the capability to provide a URL in the user interface. The user can then download a file with simulation results to be visualized using the provided URL. This version would work well with a research data management system. First, the researcher would perform a simulation. Then, they would upload the results to the research data management system. Afterwards, a link to the simulation results, probably in the form of a DOI, could be included in a publication or correspondence with a research partner. The link can then be inserted into the WebAssembly application, which downloads and visualizes the simulation results.

References

[1] O. MOKROV, S. WARKENTIN, L. WESTHOFEN, J. BENDER, R. SHARMA, U. REISGEN: 'Enhancing Gmaw Simulations Through A Hybrid Eulerian and Lagrangian Method Considering an Inclined Welding Torch', *The 14th International Seminar "Numerical Analysis of Weldability*, Graz, Austria, 21 - 24 September 2025.

[2] O. MOKROV, S. WARKENTIN, L. WESTHOFEN, J. ANTONISSEN, J. BENDER, R. SHARMA, U. REISGEN: 'Solution Approaches for the Efficient Modeling of the Layer Build-Up in the Waam Manufacturing Process Using Smoothed Particle Hydrodynamics', *The 14th International Seminar "Numerical Analysis of Weldability*", Graz, Austria, 21 - 24 September 2025.

[3] J. BENDER, ET AL: *SPlisHSPlasH Library*, https://github.com/InteractiveComputerGraphics/SPlisHSPlasH, MIT license.

[4] AYACHIT, UTKARSH: *The ParaView Guide: A Parallel Visualization Application*, ISBN: 1930934300, Kitware Inc., 2015.

[5] AYACHIT, UTKARSH: *The ParaView Guide: A Parallel Visualization Application*, SECTION 8.1, ISBN: 1930934300, Kitware Inc., 2015.

[6] AYACHIT, UTKARSH: *The ParaView Guide: A Parallel Visualization Application*, SECTION 8.8, ISBN: 1930934300, Kitware Inc., 2015.

[7] S. JOURDAIN ET AL: *Trame, Trame Documentation – Guide - Overview*, doi.org/10.5281/zenodo.16898084, 2025.

[8] G. V. ROSSUM: *Glue It All Together With Python*, OMG-DARPA-MCC Workshop on Compositional Software Architecture, Monterey, California, January 6-8, 1998.

[9] A. ROSSBERG: *WebAssembly Core Specification*, Version 2, W3C, 2022.

[10] J. PANCHUMARTI, V. BOLEA, Y. HAN, S. JHAVERI, B. MAJOR, P. O'LEARY and W. SCHROEDER: *Introducing WebAssembly support in VTK*, KitWare Blog, August 15, 2025, https://www.kitware.com/introducing-webassembly-support-in-vtk/.