Inviwo — A Visualization System with Usage Abstraction Levels

Daniel Jönsson[ID], Peter Steneteg[ID], Erik Sundén[ID], Rickard Englund, Sathish Kottravel[ID], Martin Falk[ID], *Member, IEEE*, Anders Ynnerman[ID], Ingrid Hotz[ID], and Timo Ropinski[ID], *Member, IEEE*

Abstract—The complexity of today's visualization applications demands specific visualization systems tailored for the development of these applications. Frequently, such systems utilize levels of abstraction to improve the application development process, for instance by providing a data flow network editor. Unfortunately, these abstractions result in several issues, which need to be circumvented through an abstraction-centered system design. Often, a high level of abstraction hides low level details, which makes it difficult to directly access the underlying computing platform, which would be important to achieve an optimal performance. Therefore, we propose a layer structure developed for modern and sustainable visualization systems allowing developers to interact with all contained abstraction levels. We refer to this interaction capabilities as usage abstraction levels, since we target application developers with various levels of experience. We formulate the requirements for such a system, derive the desired architecture, and present how the concepts have been exemplary realized within the Inviwo visualization system. Furthermore, we address several specific challenges that arise during the realization of such a layered architecture, such as communication between different computing platforms, performance centered encapsulation, as well as layer-independent development by supporting cross layer documentation and debugging capabilities.

Index Terms—Visualization systems, data visualization, visual analytics, data analysis, computer graphics, image processing

✦

1 INTRODUCTION

THE field of visualization is maturing, and a shift can be observed from algorithm-centric research to application-centric research. While in previous years research has focused on novel visualization methods and algorithms for a particular data type, e.g., volume rendering [1], line integral convolution [2], or tensor glyphs [3], today visualization research also puts emphasis on the solution of specific application-oriented visualization problems in a wide range of domains, e.g., in medical visualization [4], engineering sciences [5], biological visualization [6], or astronomy [7]. Visualization researchers are confronted with new challenges, as in most applications the interplay between different visualization algorithms and the integration of multiple data sources must be considered. Furthermore, data is typically large and heterogeneous. All these aspects make it challenging to develop interactive visualization applications that are applicable to real-world problems. Over the years, several efforts have been made to provide systems designed to ease the development of visualization applications. VTK [8] was one of the foundational

frameworks for this purpose, enabling visualization application developers to load the data to be visualized and combine several building blocks into a visualization pipeline. Following the success of VTK, many visualization systems have been released to enable application-centric visualization research, e.g., VisTrails [9], VolumeShop [10], MeVisLab [11], VisIt [12], VAPOR [13], Voreen [14] and Amira [15]. While the capabilities of these systems vary, most of them make use of the popular separation of concerns design principle [16] by employing different layers of abstraction. Since the visualization pipeline can readily be modeled through data flow networks [17], many modern visualization systems expose the highest layer of abstraction as a data flow network editor to the application developer, e.g., VisTrails [9], MeVisLab [11] or Voreen [14]. This layered architecture approach has been proven successful, but it also comes with a few downsides, which could potentially hamper the visualization application development process.

It can first be observed that visualization application developers have varying needs. Exposing the highest layer of abstraction is often not enough to facilitate the development of complex visualization applications, as in many scenarios it becomes necessary to modify functionality rooted at different levels of abstraction. For instance, besides modifying the data flow network a developer might also want to change the C++ or computing platform specific code for a particular building block. As data flow network editors support rapid development, i.e., the influence of made changes can be directly seen in the visualization, such a development paradigm is also desired at the lower levels of

---

• D. Jönsson, P. Steneteg, E. Sundén, R. Englund, S. Kottravel, M. Falk, A. Ynnerman, and I. Hotz are with the Linköing University, Linköing 581 83, Sweden. E-mail: {daniel.jonsson, peter.steneteg, erik.sunden, rickard.englund, sathish.kottravel, martin.falk, anders.ynnerman, ingrid.hotz}@liu.se.
• T. Ropinski is with Ulm University, Ulm 89081, Germany, and Linköping University, Linköing 581 83, Sweden. E-mail: timo.ropinski@uni-ulm.de.

abstraction. Furthermore, as a consequence of such changes, it also becomes necessary to debug and document at these levels. We refer to the possibility to change, debug, test and document functionality at different abstraction layers to *cross-layer development*. We regard our realization of this concept as one of the two main contributions of this system paper.

A fundamental problem in the layer abstraction used by modern visualization systems is related to the underlying hardware used. While it is undisputed that modern visualization applications should be interactive [19], the size and complexity of data often makes it challenging to maintain interactive frame rates. To deal with this challenge, the ever increasing capabilities of graphics processing units (GPUs) is exploited [20], and consequently most current visualization systems, e.g., [9], [10], [11], [13], [14] and [15], rely on the GPU for both data processing and rendering. This is often done by interfacing with computing platforms, such as OpenCL, CUDA or OpenGL, from within the lowest level of abstraction. While this allows for leveraging the power of modern GPUs, it also comes with the drawback that algorithms cannot be combined with new computing platforms, e.g., an algorithm developed for OpenCL cannot directly be used with an algorithm developed for OpenGL. This is especially problematic when considering the fact that new computing platforms are released rather frequently (2004: OpenGL 2.0, 2006:DirectX 10, 2007:CUDA, 2008:OpenGL 3.0, 2009:OpenCL/DirectX 11, 2015:DirectX 12, 2016:Vulkan). In this paper, we propose concepts enabling algorithms implemented for different computing platforms to be used together seamlessly. For example, the image resulting from a rendering using OpenGL can be piped into an image processing algorithm developed using OpenCL. Neither of the two algorithms need to be aware of each other and both algorithms can be programmed using their specific computing platform, which is not possible in other visualization frameworks. This computing platform interoperability is the second main contribution of this system paper.

In this paper we propose design principles, and their technical realizations, which support cross-layer development and computing platform interoperability in modern visualization systems. The proposed design principles have been developed and integrated into the Inviwo visualization system, which has been used to implement a range of visualization applications, e.g., volume data exploration [21], molecular visualization [22], and material science [23]. The proposed concepts are in themselves a contribution beyond the Inviwo system. As visualization systems with cross-layer development and integrated computing platform interoperability will make it possible to use the same system during the entire visualization application development process and gracefully adapt to the introduction of new computing platforms, we foresee that other system developers will find our principles and realization approaches useful.

2   RELATED WORK

Here, we provide a comprehensive comparison between visualization systems and game engines with respect to their computing platform interoperability capabilities and usage abstraction layers for visualization application development.

Visualization Systems. The Visualization Toolkit (VTK) by Schroeder et al. [8] is a C++ framework for creating visualizations and therefore requires several usage abstraction levels to be built on top of the framework for it to be accessible to visualization practitioners. More recently, VTK-m [24] has been developed to better support parallel architectures. Unlike the approach taken in this work, VTK-m provide an abstraction preventing access to the underlying computing platforms. Since VTK and VTK-m are lower abstraction level frameworks, they can be integrated into Inviwo and thereby benefit from its usage abstraction levels.

Examples of systems providing usage abstraction levels are ParaView [18], VisTrails [9], VisIt [12] and tomviz [25]. Since VTK does not have integrated computing platform interoperability, the systems building on this API do not have this property either. Similar to Inviwo, ParaView and VisTrails are general platforms that do not target a specific domain or type of data. Out of these, ParaView uses a treeview to represent the visualization pipeline while Inviwo and many others, e.g., MevisLab [11], Amira [15], VisTrails [9], and Voreen [14], uses an acyclic graph representation. VisIt primarily targets parallel systems and streaming of data defined on 2D and 3D meshes, while tomviz focuses on transmission electron microscope data.

There are also many general systems that are not based on VTK. Amira and MeVisLab are commercial systems targeting life sciences. Another commercial system is AVS Express [26], but it does not target a specific domain. Voreen was designed for volume rendering and has OpenGL built into its core, which prevents it from integrating computing platform interoperability. Forbes et al. [27] introduced a Java-based framework designed for dynamic data by explicitly separating processing into three interconnected graphs. Their graphs involve scene graph logics, data processing and timing operations for interaction and animation. The work by Forbes et al. [27] can be seen as orthogonal to the concepts presented here since they could be used together as an additional usage abstraction level with respect to scene graph logics and timing operations. A range of systems are more specialized, such as FAnToM [28], focusing on topological analysis, VAPOR, focusing on data from earth and space sciences and VolumeShop [10], focusing on illustrative visualization of volume data. While most of these systems support multiple computing platforms, none of them allow algorithms implemented for different computing platforms to be seamlessly combined.

Visualization Systems Comparison. In the following, we will compare several major visualization systems with respect to their computing platform interoperability support and cross-layer development capabilities. The ones selected for comparison, seen in Table 1, are actively developed and support the whole visualization design process. Four concepts were considered when it comes to cross-layer development, being able to develop multiple layers of abstraction at runtime, visual debugging of the data flow network, co-locating code and documentation for higher abstraction layers and testing at functional and visualization pipeline levels, i.e., unit and visualization integration and regression testing.

TABLE 1
Comparison of Major Visualizations Systems with Respect to Seamless Combination of Algorithms Implemented on Different Computing Platforms and the Support for Cross-Layer Development

		Inviwo	MeVisLab [11]	Amira [15]	ParaView [18]	VisIt [12]	VisTrails [9]
License		BSD	Commercial	Commercial	BSD	BSD	GNU GPL v2
Integrated computing platform interoperability		•	—	—	—	—	—
Interactive development	C++	•	•	—	—	—	NA
	Shaders	•	•	•	—	—	—
Visual debugging		•	•	—	—	—	—
Integrated documentation		•	—	—	—	—	•
Visualization pipeline testing		•	•	•	•	•	•

As seen in Table 1, many systems have support for multiple computing platforms but only Inviwo has a concept for seamlessly combining algorithms written for different computing platforms. Multiple systems have support for interactive shader editing (Amira, Inviwo, MeVisLab), i.e., changing them at runtime, while only a few provide means for interactively developing algorithms (Inviwo, MeVisLab).

In the software development process, debugging involves locating, identifying, and correcting a defect [29]. Debugging therefore involves everything from printing variables to the console to stepping through code to see where logical errors are made. In the field of information visualization, Hoffswell et al. [30] presented techniques for debugging reactive data visualizations. Hoffswell et al. focus on showing the state and interaction logic over time as the user interacts with a visualization. Their techniques can be seen as orthogonal to the ones presented in this work, where the focus is on the debugging process of algorithms and not the the interaction logic. Debugging of visualization pipelines is supported by Inviwo and MeVisLab. However, the approach in MeVisLab is slightly different. While Inviwo uses visualization pipelines to provide information co-located with the port location, MeVisLab displays the information in a separate pane.

When it comes to documentation of higher abstraction level components, i.e., processors and modules, it is most common to provide it in separate files. For example, MeVisLab requires a .mhelp file for describing fields in their modules. Here, Inviwo and VisTrails stand out. VisTrails allows developers to specify documentation in code for their modules (their equivalent to processor) and ports separately, while Inviwo co-locates this documentation for multiple usage abstraction levels. All of the systems in the comparison provide testing capabilities for both individual functions as well as on visualization pipeline level. The main difference is on how easy it is to add the tests. Here, we believe that the minimal requirement in Inviwo for adding regression tests, e.g., a workspace with its resulting image(s), can lower the threshold for providing such tests. A final note is that all visualization systems included in the comparison supports use of scripting languages, such as Python or Tcl.

Game Engines, such as Unity [31] and Unreal Engine [32], have lately become viable options for building visualization systems. They provide support for multiple computing platforms but do not allow developers to access them. Sicat et al. [33] presented a toolkit based on Unity for creating immersive visualizations. However, compared to Inviwo their toolkit do not provide usage abstraction levels for editing visualization pipelines. In general, the visual debugging support of game engines is more focused on the needs of game developers and not on visualization pipelines. Furthermore, the game engines' focus on game developers can limit their use for practical visualization problems. For example, Unity pre-processes meshes to optimize them for rendering, which might not be desirable for scientific visualization purposes.

3 SYSTEM REQUIREMENTS

Visualization systems commonly use multiple layers of abstraction to realize an abstraction-centered design. As illustrated in Fig. 1, the top abstraction layer is the visualization pipeline, which act as a layer of abstraction for processor composites, i.e., a group of functional units processing data flowing in the visualization pipeline. Each processor is in turn acting as an abstraction layer for C++ code. The lowest abstraction layer provides computing platform level access. Having multiple layers of abstraction poses several challenges that we address in this paper. Foremost, how can a cross-layer development process be provided, where the developer can use concepts from low abstraction layers, i.e., debugging and testing, on the higher abstraction layers. Also, which parts of the lower abstraction layers should be exposed on higher levels. Critical aspects that we consider
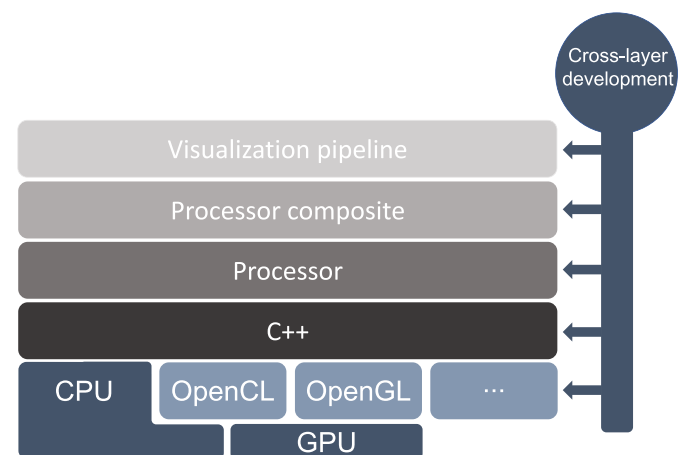

Fig. 1. Illustration of the layers of abstraction in our visualization system. The different layers of abstraction allow access to functionality for different usage levels and parts of the visualization application design process. As an example, our Python integration allows interactive development across layers through editing of the visualization pipeline or definition of new processors.

here are performance and development speed, it must be ensured that the access across abstraction layers does not hamper performance and that it helps in creating tailored visualization applications. Another challenge is communication within the computing platform layer, for example how to deal with the number computing platform combinations possible.

Application oriented research requires a new level of engineering and our aim is to *supply a sustainable system that aids researchers in all stages of the visualization application design process.* We first describe the requirements for the usage abstraction levels and then the computing platform interoperability.

A requirement for visualization systems with usage abstraction levels is that they support rapid development at all levels ranging from application tailoring to computing platform coding. From a developer's perspective, this process involves coding, debugging, documenting and testing. Thus, the system should support interactive C++ coding, shader editing and debugging of data flow networks. Documentation of algorithms is read by other developers and visualization composers, meaning that both groups must have access to it on the level they are using the system. Testing must be supported on both algorithm and visualization pipeline levels. From a visualization composer's perspective, the system must support visual creation and editing of visualization pipelines as well as the possibility to create a tailored visualization application.

Interaction is key in visualization systems. A computing platform interoperable visualization system must therefore provide access to native computing platform capabilities to allow processing of large and complex data at interactive frame rates. In addition, research is commonly at the forefront of technology usage, which means that a visualization system used for research cannot abstract away access to computing platforms. Thus, such a system must instead provide access to the lowest technical layer. For algorithms to remain portable across computing platforms, the system must provide ways of seamlessly converting data between computing platforms, i.e., transfer data between memory locations. While it is possible to manually convert data from one computing platform to another one, it requires knowledge about both computing platforms, and thus does not scale when later introducing new computing platforms. Furthermore, since this is a traveling salesman problem where the highest performance path from one computing platform to another one must be found, it is not feasible to solve this combinatorial problem manually even for a small number of computing platforms. To make things more complicated, different computing platforms also have support for different types of specialized data, such as buffers or textures. Mapping between different types of data occurring in different computing platforms is therefore an additional challenge that needs to be considered.

Concepts addressing the requirements above are presented in the following two sections.

4   USAGE ABSTRACTION LEVELS

Supporting interactive coding, shader editing and debugging requires a system allowing development to be performed across layers of abstraction. This section will define key concepts for such a system meeting the previously presented requirements. First, we describe ways of achieving interactive development. This is followed by three concepts enabling cross-layer development in terms of debugging, documenting and testing.

4.1   Interactive Development

High level development means using a visual representation for creating and changing visualization pipelines. In order to allow simultaneous high level editing and low-level C++ coding, the system must recompile and reload the changed parts while the visual editor is running. This concept must be realized through a modular system design, i.e., a plug-in system where dynamic libraries can be unloaded and loaded again to reflect the changes made. Medium usage abstraction level editing, i.e., modifying shaders used by computing platforms, or scripting, can be performed at runtime without unloading or loading dynamic libraries. Thus, interactive development at this level can be realized through either providing a user interface for recompiling the source code or observing the sources of origin, e.g., text files, and automatically recompile the source code when it changes. While the presented interactive development concept is seemingly simple, it allows the user to access multiple layers of abstraction at the same time and thus address the rapid development requirements of the system.

4.2   Visual Data Flow Debugging

Debugging parallel operations is difficult since there are many outputs occurring at the same time. Readily available debugging tools can help developers by allowing the execution of a given thread to be inspected. Inspecting one thread at a time can be tedious and is most effective when the error has been narrowed down to a specific part in the code. Visually inspecting output of parallel operations can help the developer to quickly obtain an overview and thereby identify the source of an error. This can be seen as debugging on a higher level of abstraction, giving quick access to information at lower abstraction layers.

This cross-layer debugging is provided by a concept we refer to as port inspection. A port inspector allows the contents of a processor's port to be viewed. By applying this concept in the visualization pipeline editor, it allows for step-by-step debugging of the data-flow network. For example, hovering or clicking on ports, or their connections, shows debug information about data originating from their associated outport. We can not only inspect data on a higher abstraction level, we can also create the debugging information on a high level. For this purpose, we use a visualization pipeline to provide the debugging information for a particular port type. For example, a volume data port, containing a stack of images, can use a predefined visualization pipeline for slice-based inspection. Thus, the presented concept enables cross-layer visual debugging of data flow networks.

4.3   Documentation

API documentation is more or less standardized nowadays using for example the Doxygen system. However, this

documentation is intended for developers and is not readily available for visualization pipeline editors. Providing documentation at the visualization pipeline editor level can be done in a wide variety of ways. For example, by presenting documentation written in separate files, such as xml, or html. While this makes it possible to use tools suited for text and layout, it also makes it harder to maintain and follow from a developer's perspective. For systems where low maintenance overhead and ease of use is important, we suggest to integrate this documentation procedure into the API documentation. This allows developers to write documentation once but show it in both API-documentation and to visualization pipeline editor users.

While developers are used to textual documentation, pipeline editors often desire richer information. Thus, the documentation must be presented appropriately at the different usage abstraction levels. In practice, our concept involves extracting the documentation of processors from the source code and augmenting it with the visual representation used in the higher abstraction layer. The augmented visual representation of the processor contextualizes the documentation and can be generated automatically since they are already used in the higher abstraction layer.

The documentation concept described above not only makes it easy to maintain, due to its co-location with the actual code, but also provides information on how to use a processor at multiple usage abstraction levels and thereby address the system requirements.

4.4 Testing

Unit testing is commonly used for testing low abstraction level code, but can also be used for testing an entire visualization pipeline. However, unit testing requires programming skills and therefore cannot be created in the higher usage level abstraction.

Our testing concept allows tests of visualization pipelines to be created on a high usage abstraction level, i.e., through the visualization pipeline GUI editor. Once the visualization pipeline to be tested has been defined its results before and after a change can be compared. However, comparing results of a visualization pipeline presents challenges that our concept takes into account. Here, one approach is to generate reference results on demand, i.e., never store them, and compare the new results with the generated references each time a test is run. We have found this approach to be problematic due to two reasons. First, the results can vary between different hardware, but this, possibly incorrect behavior, will never be detected since the same hardware is used for generating reference and comparison images. Second, it can be resource demanding since it requires having, or building, multiple versions for generating the reference data. Thus, reference results should instead be generated once when the test is defined. To deal with the differences in results across hardwares we allow specification of an error threshold such that, in cases an investigation has found that this indeed is the cause of the error, the test can be allowed to pass.

The presented testing concept has addressed the challenge of exposing testing at a high usage abstraction level, and thus enables the system to support testing across layers of abstraction.
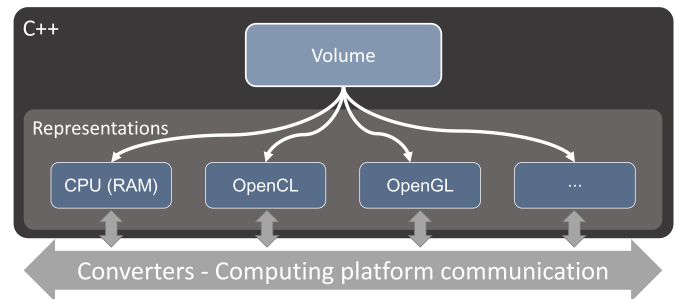

Fig. 2. Illustration of the computing platform communication in the C++ usage abstraction level for a volume data object. The Developer requests representations of the object but data transfer is performed on a layer of abstraction underneath data objects, thus never visible to the developer. Representations are mapped to types supported by hardware, i.e., buffers and textures, for performance reasons.

5 COMPUTING PLATFORM INTEROPERABILITY

The lowest abstraction layer in our system is data abstraction. This layer provides an interface for accessing data on a per computing platform basis without knowing about the other computing platforms, thus enabling developers to utilize platform specific features but still combine algorithms written for different computing platforms.

Designing a computing platform layer of abstraction requires considering both performance and sustainability. For example, the OpenGL computing platform has specialized data types for textures which must be taken into account to not loose performance. Therefore, we focus on creating a layer of abstraction for the data, not for all computing platforms. In other words, our concept allows developers to fully utilize the computing platforms on the premise that they request data for the computing platform they are utilizing. While this might seem like a too narrow abstraction, it is fitting to visualization pipelines since they operate on data flows and each processor is in this way isolated from the other ones. This abstraction layer must provide a common interface for data objects flowing in the visualization pipeline. One way of providing such an abstraction is to disallow direct usage of the underlying computing platform memory location and instead provide a wrapper exposing common functionality, similar to how for example the Java programming language operates. Clearly, this prevents developers from using computing platform specific features. Another alternative is to provide an interface for requesting the data for the desired computing platform. Given that data can be transferred to the requested computing platform, it means that the developer only needs to know about the computing platform for which data is requested for. The downsides of this approach are a more complicated data interface for developers, due to the abstraction, and that data might need to be transferred when used by different algorithms.

Our design consists of a general data interface exposing functions for requesting data for arbitrary computing platforms. Examples of data types utilizing this interface are buffers, images and volumes. Succinctly, the computing platform data representation is simply referred to as representation. The three most important objects of importance in our design are illustrated in Fig. 2 for volume data. In this figure the data interface provides an abstraction for
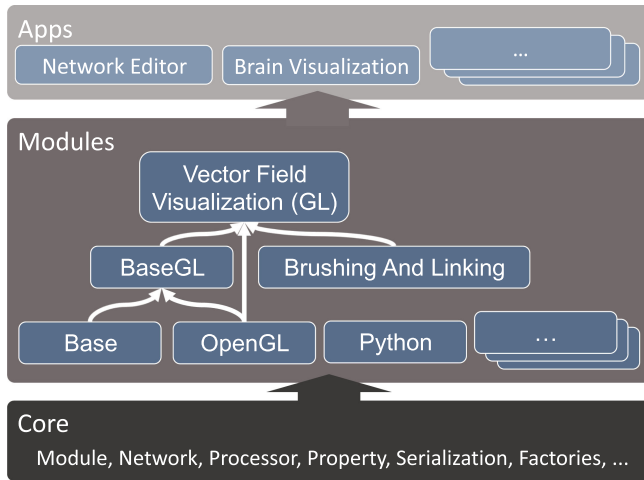
Fig. 3. The Inviwo architecture is divided into three main components. The core provides, for example, processor network functionality, saving/ loading mechanisms, and interfaces for extending the framework. Modules bring in new functionality using the interfaces and factories provided in core, or by exposing external libraries. Applications expose the processor network in various ways, for example using graphical interfaces for editing the network or tailored design for brain activity visualization.

data access, the representation manages the data, and the converter transfers data between different computing platforms. A representation is created lazily upon request. However, once requested it is cached in order to reduce the number of data transfers necessary. As the representations are typically not on the same device, i.e., RAM and video memory, caching is in general a good trade-off. A list of used representations by the data object must be stored. It is necessary to know which representations in the list that are up to date. For this purpose, a representation must be requested with either read or write access. Write access means that other representations must be updated from this representation when requested. The observant reader has identified two types of representation operations, one for creation and one for updating. It is desirable to separate these two operations since creation is generally slower than updating. Therefore, the only two tasks of a converter is to create or update a representation from another one.

In practice, computing platforms such as OpenGL and OpenCL have special data object types for which hardware accelerated operations are supported. These need to be considered to obtain interactive frame rates. Based on currently existing hardware we have identified three types of fundamental data objects. Buffers, which can store arbitrary data, 2D textures storing image data and 3D textures storing volume data. This separation ensures that hardware accelerated filtering operations can be utilized on GPU computing platforms. Moreover, while the computing platforms have evolved, these underlying types have remained the same and due to their ubiquitous usage it is not likely that they will be removed in the near future when new hardware is developed. Each of these fundamental types requires one converter per combination of computing platform. In order to form an optimal conversion path when requesting data, all combinations between the existing computing platforms must be considered. Thus, it quickly becomes unfeasible to manually implement all possible converter combinations. This challenge is addressed by algorithmically creating converter packages

from the individual converters provided by the developers. One such package contains a path to transfer data from one representation to another. These packages can be sorted in order of performance and be selected at runtime. In practice, this algorithmic converter path creation means that a developer only has to implement one converter between the new and an existing representation for it to function. Specifying more data transfer combinations can increase performance, such as in the case of shared data.

More complex data structures, such as meshes containing buffers of vertices and other attributes, can utilize the fundamental types provided as building blocks and thereby gain the same properties. The presented concept improves the sustainability of a visualization system while still allowing access to native computing platform capabilities. The presented usage abstraction levels and computing platform interoperability concepts have been implemented in Inviwo. Details of these implementations are provided in the following section.

6 REALIZATION IN THE INVIWO FRAMEWORK

The previously described concepts have been implemented in the Inviwo system. We will first provide an overview of Inviwo and describe how its different parts relate to these concepts. Then, we will discuss how the usage level abstractions and computing platform interoperability are realized in Inviwo.

Inviwo can be divided into three main components illustrated in Fig. 3. First is the core, containing functionality for evaluating a data-flow network, fundamental data structures, such as images, volumes and meshes, as well as interfaces and factories for adding additional functionality. The core has no dependencies on any GPU platform but provides the foundation for computing platform communication. Second is the module system, which is used to extend the core with new functionality at runtime in a plug-in based manner. A new module can, for example, add support for data readers, computing platforms or new visualization algorithms. Third is the visual editor in which visualization pipelines can be created, parameters can be tuned and scripting can be edited. The visual editor provides access across the layers of abstraction and supports interactive C++ coding, scripting and shader editing. It has two modes, developer and application modes. The developer mode allows data flow creation as well as parameter editing, while the application mode hides the data flow network and only shows a subset of parameters determined in the developer mode. Thus, a tailored visualization application can quickly be created.

The following sub sections will first provide a brief overview of Inviwo, then describe how the usage abstraction levels are implemented followed by the details on how algorithm interoperability is achieved.

6.1 Inviwo Overview
Inviwo is built around a central concept of a data flow graph where each node in the graph represents a functional unit. The edges in the graph represent data flowing into and out of the functional units. A functional unit, its input and output data and its parameters are encapsulated by a processor. A processor has inports, encapsulating the input data,
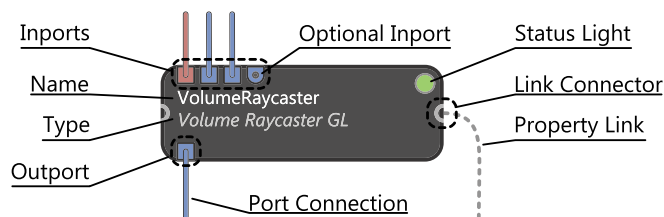
Fig. 4. Visual representation of a processor, the encapsulation of an algorithm in Inviwo. Data is flowing from the top into the inports of the processor. The resulting output data continuous its flow through the outport. Synchronization of properties between processors are represented through property links.
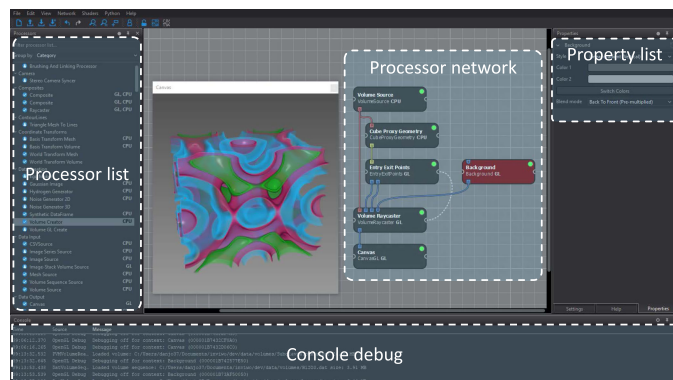

Fig. 5. Screenshot of the Inviwo network editor for rapid visualization pipeline creation. The processor network in the center loads and visualizes the charge density of the chemical boron element. New processors can be dragged from the processor list and dropped into the processor network, where a visual representation of the processor and its components will be shown. Pipelines are created by connecting processor ports in the network. Properties exposed by the selected processor are automatically shown in the property list to the right.

outports, encapsulating the output data, and properties, encapsulating parameters. Guidelines for how to best encapsulate a functional unit into a processor are provided in the appendix of this paper. A group of processors can be combined in the form of a processor composite and thus provide an abstraction for a more complex task. The graph of connected processors and processor composites is referred as the processor network, which is equivalent to a visualization pipeline.

Data can be raw data in memory, geometries, volumes, images, data frames or any other arbitrary data structure. Note that while the types provided by Inviwo generally use the data interface presented in Section 5, there is no such requirement on the port data. This means that data types in external libraries can easily be integrated and benefit from the usage abstraction levels, even though they will not gain the computing platform interoperability of the data types provided by Inviwo. Fig. 4 shows the visual representation used for a processor and its components. The processor itself is represented by a box, where input data is flowing from the top into the inports and the output data is exiting in the bottom through the outport. A processor is evaluated when all its inports, and at least one outport, are connected. The low abstraction layer concept of function overloading in programming is captured using optional inports, which do not need to be connected for the processor to evaluate.

6.2 Usage Abstractions For Interactive Development

The implementations of interactive development in Inviwo can be categorized into three usage abstraction level categories. The high usage abstraction level is based on graphical interfaces, the medium usage abstraction level contains scripting and shader editing while the lowest usage abstraction level is based on C++ coding. Next, we describe how the developer can move between these categories and achieve interactive development in Inviwo.

6.2.1 High Usage Abstraction Level Editing

Visualization pipeline abstraction level editing is performed in the processor network editor (network editor), seen in Fig. 5. The network editor is an application for visually building, editing, debugging and running processor networks. It is a key component as it provides means for crosslayer development by allowing the developer to seamlessly move between visually editing the visualization pipeline, scripting in Python and coding in C++.

The network editor uses the drag-and-drop metaphor to include processors in the network. Processors added by modules can be selected from a list and dropped into the processor network, whereby a graphical element, exemplified in Fig. 4, is created to represent the selected processor in the network. Its inports and outports are displayed as colored squares laid out in rows at the top and the bottom of the element, respectively. The color of the ports reflect the type of data they manage. Images are blue, volumes are red and meshes are yellow.

The network is automatically evaluated while building the pipeline and the results can be inspected either in a canvas or through visual debugging (see Section 6.2). Selecting a processor in the network shows graphical user interfaces (GUIs) for editing all of its properties, see the Property List in Fig. 5. The same property type can have different visual representations, which we refer to as property semantics. For example, four floating point values can be represented by four sliders or as a color. The property semantics can be set at different usage abstraction levels, i.e., in code or visually using the property's context menu. Multiple processors can be aggregated into one processor, which thereby enable high usage abstraction level access to the third layer of abstraction, processor composites.

Properties in the network can be linked (synchronized) between processors in a similar, but orthogonal, way to connections between ports. A dotted line between link connectors of two processors indicate that one or more of their properties are linked.

This high usage abstraction level editing enables interactive development of a visualization pipeline. However, the network editor also integrates with the medium and low usage level abstractions as described next.

6.2.2 Medium Usage Abstraction Level Editing

Inviwo has integrated support for the widely used Python scripting language. The Python integration in Inviwo can be used in several ways. Firstly, it is possible to perform batch processing using an integrated Python editor in the processor network editor. As an example, camera parameters can

be scripted for performance benchmarking. The integrated Python editor thereby enables the developer to seamlessly move between the high and medium usage abstraction levels. Secondly, data can be transfered back and forth to Python for computations within a processor. This is particularly valuable due to the rich data processing capabilities in Python, but also means that algorithms available in Inviwo can be used in Python. The Python data transfer is typically used within a processor, but the script can be exposed through a property which thus enables high usage abstraction level editing. Third, a processor and its processing can be defined entirely in Python. This is advantageous for developers more comfortable with Python, and of course inherits the benefits of working with a scripting language while still taking advantage of the usage abstraction level concepts in Inviwo. Finally, we point out that Python is still integrated through a module, i.e., it is not included in Inviwo core, which demonstrates the modularity of Inviwo.

Interactive shader editing in the network editor is also supported by observing all shader files. Saving a shader file will notify the network editor, which will invalidate the processor network, causing it to update the output. Similiar to Python scripting, some processors also expose shaders through properties. Changing a shader in this case invalidates the property, which also causes the processor network to update.

6.2.3   Low Usage Abstraction Level Editing

The Inviwo system is based on a modular structure, which is the foundation for achieving interactive development. All modules depend on the core and, in addition, they can depend on other modules or external libraries, see Fig. 3. Dependencies on external libraries occur when integrating their functionality into Inviwo. The module system allows new functionality to be included without introducing any dependencies into the core system. For example, the OpenGL module adds OpenGL computing platform support to the Inviwo system. Other modules can depend on the OpenGL module to build on those capabilities. Inviwo takes care of managing the dependencies between the different modules. When possible, external library source code should be brought into the build system to ensure version compatibility in deployed applications.

Interactive coding is enabled by allowing modules to be loaded at runtime when they change, also known as hot reloading. The network editor observes all module shared library files for change, meaning that the developer can recompile the module while the application is running and immediately see the effects of the change. The processor network is serialized, modules are reloaded, and the network is deserialized again. It should be noted that the developer has the responsibility to ensure that a module can be unloaded and loaded repeatedly, for example by not making breaking API changes, as the system cannot regulate this. A module version system ensures that modules of correct version are loaded at runtime. Modules are only loaded if they are linked to the correct Inviwo core version used by the application. Thus, it is only necessary to bump the module version if the module is released in between Inviwo core releases. This version dependency means that module developers rarely need to care about the module version.
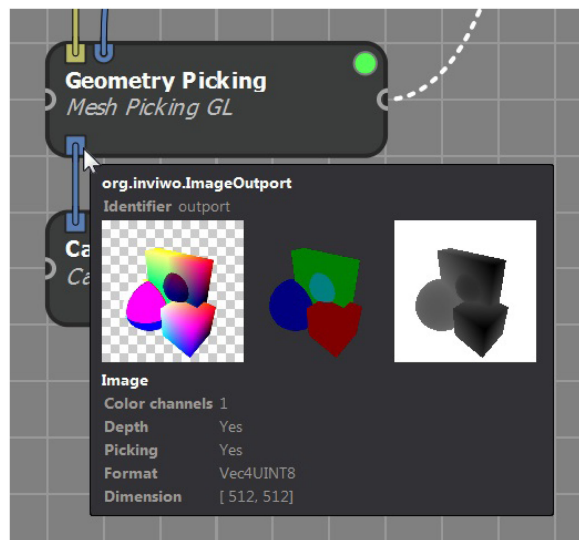

Fig. 6. Example of visual debugging of a visualization pipeline. In this instance, a port inspector shows the rendered color layer, the picking buffer, and the contents of the depth buffer. Details, such as data format and dimension are, displayed below the images.

In essence, the presented interactive development implementation allows developers to rapidly move between a low usage abstraction level, i.e., editing the code, and a high usage abstraction level, i.e., visually editing the visualization pipeline.

6.3   Visual Debugging

Inviwo provides port inspectors based on processor networks for three types: meshes, images and volumes. Modules can add more port inspectors if desired. Events are forwarded to port inspectors meaning that interaction is possible. As an example, the volume port inspector, showing a series of slices, allows for changing the current slice using the scroll wheel of a mouse. Since the pipelines created for port inspection by design are computing platform interoperable, it is possible to use the port inspectors on data residing in another computing platform's memory.

Creating an entire processor network for less commonly used port types can be to high of a threshold. Inviwo therefore also allows port debug information to be created using C++ template traits for the port type. The port template trait output debug text, expected in HTML-format, will be displayed together with the results of the data flow port inspection, if existing. This enables visual debugging information to be created at both low and high usage abstraction levels.

Fig. 6 depicts the inspection of an image port as realized in the Inviwo system. It shows the output images generated or forwarded by the respective processor, including additional information such as image format and dimension. A default implementation for ports displays the name of the port, meaning that the port inspection can be applied to all ports and port connections without any efforts from a developer.

6.4   Two-Level Documentation

Inviwo implements the documentation concept combining low level API-documentation and high level documentation in the network editor using the Doxygen system. The Doxygen docpage command is used in the header file of the processor, thus co-locating the documentation for developers

```
/** \docpage{org.inviwo.Background, Background}
 * ![](org.inviwo.Background.png?classIdentifier=org.inviwo.Background)
 * Adds a background to an image.
 * The following mixing is applied
 *
 *     out.rgb = in.rgb + color.rgb * color.a * (1.0 – in.a)
 *     out.a = in.a + color.a * (1.0 – in.a)
 *
 * ### Inports
 *   * __ImageInport__ Input image.
 *
 * ### Outports
 *   * __ImageOutport__ Output image.
 *
 * ### Properties
 *   * __Style__ The are three different styles to choose from Linear gradient, uniform color,
 *     or checker board.
 *   * __Color1__ Used as the uniform color and as color 1 in the gradient and checkerboard.
 *   * __Color2__ Used as color 2 the gradient and checkerboard.
 *   * __Checker Board Size__ The size of the rectangles in the checker board.
 *   * __Switch colors__ Button to switch color 1 and 2.
 */


/**
 * \brief Adds a background to an image.
 *
 */
class IVW_MODULE_BASEGL_API Background : public Processor {
public:
    Background();
    virtual ~Background();
```

Background

inport

Background
Background GL

outport

Adds a background to an image. The following mixing is applied

$out.rgb = in.rgb + color.rgb * color.a * (1.0 - in.a)$
$out.a = in.a + color.a * (1.0 - in.a)$

Inports

* **ImageInport** Input image.

Outports

* **ImageOutport** Output image.

Properties

* **Style** The are three different styles to choose from Linear gradient, uniform color, or checker board.
* **Color1** Used as the uniform color and as color 1 in the gradient and checkerboard.
* **Color2** Used as color 2 the gradient and checkerboard.
* **Checker Board Size** The size of the rectangles in the checker board.
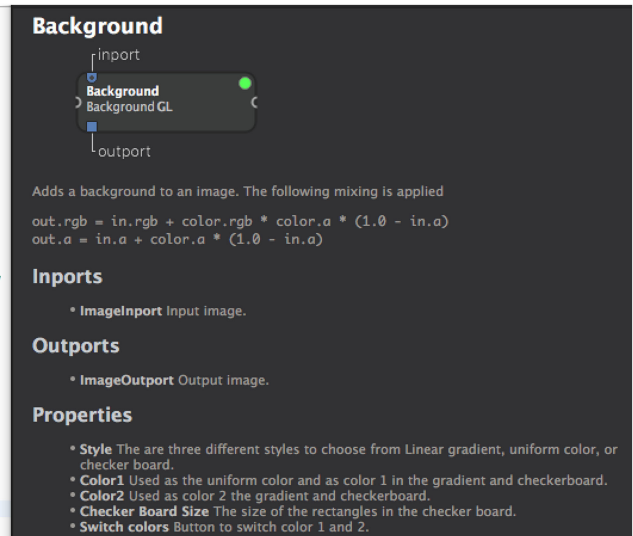* **Switch colors** Button to switch color 1 and 2.

Fig. 7. Inviwo processor documentation, based on Markdown and doxygen, serves as both API (left) and end user documentation (right). The visual processor representation seen at the top in the right image is automatically generated by the Inviwo documentation system.

and visualization editor users. Each docpage provides descriptions of how to use the processor, its inports, outports and parameters. In practice, we use Markdown language for this purpose, since it is supported by Doxygen and thereby provides a rich way for describing the module. The documentation itself is generated at compile time using a separate compile target. The processor identifier, which is a unique identifier available in all processors, is used to provide a connection between the documentation and the processor. This makes it possible to also automatically generate images of the visual representation of each processor. These images provide a contextual overview of the processors in the documentation. An example of API and end user documentation based on the same comments can be seen in Fig. 7. In order to reduce copy-paste errors, and to make it quick and easy to create as well as document a processor, boiler plate processor code can be generated through a python script.

6.5 Cross-Layer Testing

The Inviwo system tries to make it as easy as possible for developers to create tests across the layers of abstraction, i.e, code and entire processor networks. For example, the tests made for Inviwo can run on the developer's machine, they do not require a separate testing machine. The system currently provides two means for testing, unit tests, which focus on testing each element of the code, and regression tests, which focus on testing integration between processors, i.e., the top abstraction layer described in Section 4.4.

Module developers can add tests by creating a folder called "tests" in their module directory and putting unit tests in sub-folder "unittests" and regression tests in sub-folder "regression". All tests organized according to this structure will automatically be added by the framework. Further details about the two types of testing is provided below.

6.5.1 Unit Testing

Inviwo relies heavily on Google Test [34] for its unit testing and, instead of providing its own unit testing framework,

tries to provide a smooth integration. The unit tests are performed on a per-module basis. As mentioned above, all files having cpp file-ending residing inside the unit testing folder of the module are automatically considered. Each test source file can include one or multiple tests. By default, unit testing of all modules is included in the build process of the system. This means that the tests are run after a successful build and output information about the tests is shown in the integrated developer environment (IDE). Unit tests can also be run individually if desired.

6.5.2 Regression Testing

While writing unit tests for each individual unit of code ensure correctness of the unit, it does not necessarily test its integration with other units. Regression tests in Inviwo execute a whole network and compare its output with a result deemed correct. Creating a regression test is therefore only a matter of saving the output in the processor network, commonly the canvas images, along with the workspace containing the network. Future changes producing a different output will fail the regression test. Thus, this is a quick and easy way of detecting if code changes have an undesirable effect on the output of existing algorithms. While this approach does not tell exactly which unit of code caused the failure, it does tell which code changes that did. From experience, we have found that GPU hardware might produce small numerical differences depending on driver version or manufacturer. To resolve this, the developer can configure the regression test error threshold per output. Note that allowing errors should be used with caution and only after thorough investigation.

Python scripts can be used to simulate interaction or other runtime changes. All Python scripts in the folder of the regression test will be executed after loading the network, meaning that all functionality exposed through the Inviwo Python integration can be utilized in a regression test. The Inviwo regression testing environment itself is setup using Python scripts. The Python scripts manage the execution of the tests and also generate a report in HTML. An example of a regression test report is depicted in Fig. 8.
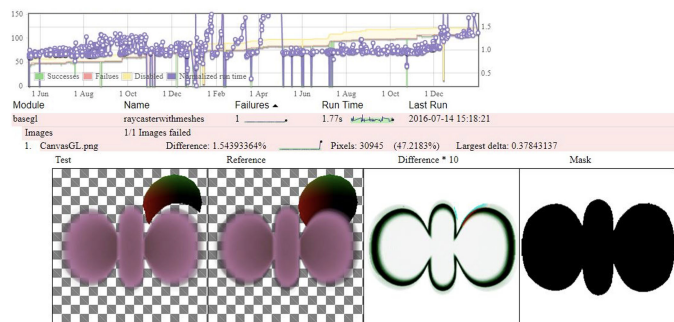
Fig. 8. Depiction of a regression test report. The plot in the top provides an overview of the regression test history. Test details show the new output, the reference, magnified difference between them and a binary mask of where error is above the error threshold. Additional details include the runtime log and explanation of the error causing the failure.

Besides details about the outcome of the tests, the report allows developers to see the difference between the result image and the reference image as well as analyzing performance measurements over time.

Regression tests need to be executed on demand since they are too time-consuming to run as a build-step. Correctness and stability of the public repository code is ensured by utilizing a continuous integration server, which runs all tests before merging code changes.

6.6 Algorithm Interoperability

Inviwo has implemented the data interface described in Section 5 using template functions for retrieving representations. The returned representations are therefore strongly typed, which aids in the development process. Currently, CPU, OpenGL and OpenCL computing platforms are supported. We foresee that future heterogeneous computing platforms may integrate CPUs and GPUs into the same hardware. The data location will in this case be be shared among the different computing platforms. Inviwo system takes this scenario into account by allowing the underlying data to the shared among different computing platforms. Currently, Inviwo supports shared OpenGL and OpenCL buffers and textures. No data transfer is in this case necessary, since they reside at the same video memory location. There are a range of implementation aspects that need to be considered when using shared OpenCL representations. First, it requires that data has first been allocated in OpenGL. Thus, if data is located on RAM it must first be transfered to OpenGL before a shared OpenCL representation can be created. The algorithmically created converter packages described in Section 5 substantially reduces the implementation effort in this case. Second, OpenCL supports a limited subset of data formats compared to OpenGL. Lower versions of OpenCL do for example not support depth textures. Therefore, Inviwo currently throws errors when these unsupported formats are used. Ideally, they should be converted without errors even though performance might suffer, which can be seen as a future improvement for the system. The implemented computing platform converters makes Inviwo computing platform interoperable and allows current and future computing platforms to be used without changing existing implementations as required in Section 3.

7 APPLICATION EXAMPLES

Inviwo has successfully been used in numerous scientific publications within different application areas, commercial products, and university-level courses. The scientific contributions include work on advanced volumetric illumination [35], [36], [37], medical and molecular visualization [38], [39], [40], transfer function design for volume rendering [21], [41], crowd-sourcing-based user studies [42], [43], [44], topological analysis [45], [46], [47], as well as multi-variate data and flow visualization [48], [49], [50]. In addition, Inviwo is to the best of our knowledge currently used by four different universities and two commercial companies.

We have selected four examples demonstrating usage of Inviwo at different usage abstraction levels as well as its computing platform interoperable capabilities. Admittedly, some of these concepts are hard to demonstrate since they are integrated in the development process. We start by demonstrating the core concepts in a fictitious example starting from importing data over adding additional functionality in shaders and processors to creating regression tests. Afterward, we continue with real-world examples including large scale pathology imaging, public brain visualization exhibition, and education.

7.1 Usage Abstraction Level Work-Flow

This example demonstrates how the presented concepts are used in the process of creating a visualization. It starts on a high usage abstraction level, goes into medium and low usage abstraction levels before finishing with creating a visualization pipeline test. The reader is strongly encouraged to have a look at the video in the supplementary material which complements this textual description.

In order to create a volume rendering pipeline, the user starts by dragging and dropping a volume source processor set into the network editor and referring to a CT scan of a salmon. Next, three more processors are added to the pipeline; one for providing the bounding box geometry for the volume, one for generating entry and exit points for each pixel, and a volume raycaster. This processor renders the volume by considering the output of the other processors including the volume source. Finally, a canvas processor is added to show the resulting image. The volume data is then further explored by adjusting the transfer function. To quickly get an understanding of the data flow in the pipeline, the user hovers over the outports of each processor and thereby sees their content by means of the port inspector concept.

In the next step, the user wants to clip the volume programmatically in the shader, which we refer to as medium usage abstraction level. The raycasting shader is extended accordingly and the changes are immediately propagated to the network after saving. The necessary parameters, i.e. a boolean flag and the spatial position of the clipping, are exposed via shader uniforms. On the low usage abstraction level, the user can now add matching properties to the C++ source of the volume raycaster processor. These properties are then used to set the uniforms in the shader. A recompilation of the code causes the network editor to reload the network, which demonstrates the interactive development possibilities even at the low usage
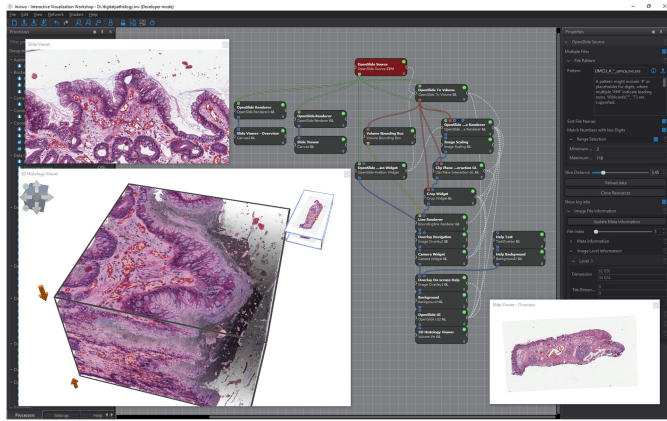
Fig. 9. Digital pathology: Visualizing a stack of 100 colored microscopic slides, each with a resolution of 63,000×35,000 pixels and totaling in 110 GiB of compressed TIFF image data.

Fig. 10. Public exhibition of human brain activity depicting where active areas are located in the brain when performing different tasks.

abstraction levels. Moving back to high level editing in the network editor, the newly added properties are accessible in the property list and can be used to interactively adjust the clipped region.

High and medium usage abstraction levels are exposed by the Python scripting integration. Here, the user decides to export each xy slice of the volume dataset individually which requires a script looping over all slices in the volume and writing the respective contents of the canvas to disk.

To ensure that the results remain consistent despite future changes, a regression test is created by selecting create regression test in the menu. This demonstrates the ability to quickly create tests at high usage abstraction levels. The regression test is run locally and the resulting regression report is inspected.

7.2 Applied Research Usage Example

In recent years, hospitals have started to digitize tissue samples on glass slides using scanners to obtain digital high-resolution color images of the samples [51]. Multiple digitized slide images of the same sample can be co-registered and combined into a single volume, thereby closely resembling the original block of tissue. The challenges associated with this kind of data include dealing with large amounts of image data, ways of interacting with this data, and the ability to look at the inside of the volume, i.e., the image stack.

The visualization of the image stack is based on brick-based volume raycasting [52], thus enabling large scale data handling of volumetric data within Inviwo. Inviwo provides a thread pool, which is used for on demand asynchronous loading of bricked image data. Image tiles are first loaded using the OpenSlide image library [53] and then uploaded into a 3D texture atlas. A volume, indicating the brick indices, is forwarded to the raycasting along with the texture atlas. The existing raycasting processor was extended in order to account for RGB image data instead of scalar values. A 2D transfer function based on color similarity [41] enables the user to adjust opacity as well as replacement colors, which could, e.g., be used for highlighting features.

Fig. 9 shows an interactive rendering of 100 slide images as well as domain specific navigation. The TIFF jpeg compressed image data consumes about 100 GiB on disk and features details like individual nuclei. In order to address
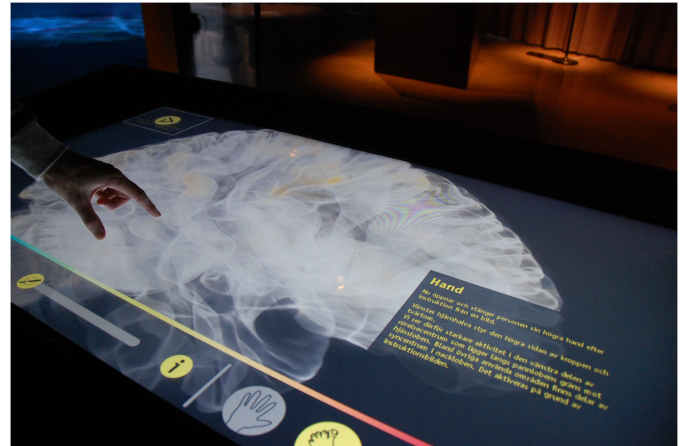
the challenges related to interacting with the stack of slide images, Inviwo provides a user interface, a minimap, and navigation widgets. The result is a tailored application that is used by a number of pathologists located at several hospitals and was evaluted in a user study with practising pathologists [39]. This scenario has demonstrated the ability to develop at both low and high abstraction layers in Inviwo as well as its capabilities in tailoring an application to domain scientists.

7.3 Public Dissemination Usage Example

Taking research into public display, such as science centers and museums, poses different demands on a system compared to prototyping ideas for research. Stability, in case the application will be running all day, seven days a week, is a an even more important requirement. In addition, the application must be responsive and the user interface should be tailored to suit visitors without prior experience regarding the underlying algorithms [21].

Fig. 10 depicts a brain visualization application where the entire user interface has been tailored for public use. The application visualizes where active areas in the brain are located while performing different tasks, such as wiggling toes or listening to music. It uses volume visualization techniques to fuse data from magnetic resonance imaging (MRI) and functional MRI, inspired by Nguyen et al. [54]. The graphical user interface communicates with the processor network driving the brain visualization. It uses the built-in processor network evaluation, touch interaction, and data management, while extending Inviwo with new modules for multi-modal volume visualization.

In this scenario, the work-flow for creating the application was as follows. In a first step, the visualization pipeline was set up in the network editor using readily available processors for loading MRI and fMRI neuroimaging data, data processing, and interaction. A new volume visualization processor was created for fusing the MRI and fMRI signals. Finally, Inviwo was used as an API in an application with a custom user interface, where the GUI elements were connected to the corresponding properties in the processor network using their identifiers.

The application has been running every day during opening hours since 2015 at the Tekniska museet (Technical

Museum) in Stockholm, Sweden, and the Visualiseringscenter C (Visualization Center C) in Norrköping, Sweden. A big advantage of this setup has been that the actual data processing, rendering, and interaction could be fully created and adjusted on a higher level of abstraction within the Inviwo network editor. This application scenario demonstrates the use of Inviwo at a low level, as an API, at a high level through the visual network editor, as well as its stability, responsiveness, and user interface flexibility.

7.4 Educational Usage Example

Inviwo has been used as the underlying platform for a number of master thesis' and is currently being used in, to our knowledge, four courses on visualization. In these courses, the students are primarily using the system to get an in-depth understanding of visualization algorithms by implementing them, but also to explore how common visualization algorithms work in practice. VTK [8], Paraview [18] and VisTrails [9] have previously been used for two of these courses. From this experience, it was found that use of pure VTK imposed a steep learning curve and that the tree-view of data-flow in Paraview was hard for the students to understand with respect to how data is flowing in the visualization pipeline. VisTrails has succesfully been used for education [55] and provides a similar setup as Inviwo with respect to the visualization pipeline. However, VisTrails tends to expose many inports and outports per node/processor, which was confusing to the students. In Inviwo, the number of ports of processors are reduced by applying the processor creation guidelines presented in the appendix, which make it easier for new users of the system. The visual debugging concepts further help the students to understand the data flows and find errors in their implementations.

As a contrast to the courses targeting visualization, Inviwo is also used in a course for physics students on Bachelor level. The physics students design visualizations of electron structure simulation data for common tasks involving analysis of charge densities, molecular dynamics and crystal structures. They use both the Python scripting in Inviwo as well as the high level network editor to accomplish their goals.

To summarize, Inviwo has been used by hundreds of students to both get in-depth understanding of visualization algorithms and how to use them to understand data. The students use the system at all levels of abstraction.

8 DISCUSSION AND CONCLUSIONS

In this paper we have described how to design a computing platform interoperable visualization system with usage abstraction levels through a layered architecture. The proposed computing platform interoperable solution allows algorithms developed for different computing platforms to be used together while still allowing developers to access the underlying computing platforms.

We presented several concepts for interactive development, debugging, documentation and testing across layers of abstraction. More specifically, the concepts allow tasks commonly performed by a developer at lower levels, i.e., debugging in IDEs, to be performed on a visualization pipeline level. While some of these concepts are found in other systems, we have formalized them and brought them

together in one system. It is the combination of the presented concepts that enable tailored visualization applications to be rapidly developed. For example, since the system is computing platform interoperable, it is possible to design a port inspector using OpenGL and apply it to data residing in OpenCL. This also demonstrates the crucial success factor in view of the amount of work that goes into creating and maintaining advanced visualization systems.

The concepts were demonstrated in the Inviwo system along with accounts for implementation choices and technical details, such as use of shared computing platforms, that need to be considered when realizing the concepts. Naturally, not all details can be provided in a systems paper but a comprehensive account for technical aspects is available on the Inviwo website (www.inviwo.org), where the source code is also distributed freely under the BSD license.

Several application examples, implemented using the Inviwo system, demonstrated the wide usage levels made available through the described concepts. It was shown that Inviwo meets the requirements and can be used throughout the whole visualization application design process, from writing functional units, through visualization pipeline editing, to tailored application creation.

While the presented computing platform interoperable solution allows for high performance and novel utilization of each computing platform, it can also be seen as one of the limitations since algorithms will be designed for a specific computing platform and will have to be ported to devices not supporting them. For example, algorithms using OpenCL cannot run on mobile phones. The presented concepts have been designed and demonstrated for single-computer usage. An interesting direction for future work would be to take the demonstrated out-of-core concepts and extend them to distributed computing.

REFERENCES

[1] D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski, "A survey of volumetric illumination techniques for interactive volume rendering," *Comput. Graph. Forum*, vol. 33, no. 1, pp. 27–51, 2014.
[2] B. Cabral and L. C. Leedom, "Imaging vector fields using line integral convolution," in *Proc. 20th Annu. Conf. Comput. Graph. Interactive Techniques*, 1993, pp. 263–270.
[3] G. Kindlmann, "Superquadric tensor glyphs," in *Proc. 6th Joint Eurographics-IEEE TCVG Conf. Vis.*, 2004, pp. 147–154.
[4] N. Smit, K. Lawonn, A. Kraima, M. DeRuiter, H. Sokooti, S. Bruckner, E. Eisemann, and A. Vilanova, "Pelvis: Atlas-based surgical planning for oncological pelvic surgery," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 741–750, Jan. 2017.

[5] C.-M. Chen, S. Dutta, X. Liu, G. Heinlein, H.-W. Shen, and J.-P. Chen, "Visualization and analysis of rotating stall for transonic jet engine simulation," *IEEE Trans. Vis. Comput. Graph.*, vol. 22, no. 1, pp. 847–856, Jan. 2016.

[6] M. Le Muzic, L. Autin, J. Parulek, and I. Viola, "cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets," in *Proc. Eurographics Workshop Visual Comput. Biomed.*, 2015, Art. no. 61.

[7] A. Bock, A. Pembroke, M. L. Mays, L. Rastaetter, A. Ynnerman, and T. Ropinski, "Visual verification of space weather ensemble simulations," in *Proc. IEEE Sci. Vis. Conf.*, 2015, pp. 17–24.

[8] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, 4th ed. Clifton Park, NY, USA: Kitware, 2006.

[9] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "Vistrails: Enabling interactive multiple-view visualizations," in *Proc. IEEE Vis*, 2005, pp. 135–142.

[10] S. Bruckner and M. E. Gröller, *Volumeshop: An Interactive System for Direct Volume Illustration*. Piscataway, NJ, USA: IEEE, 2005.

[11] MeVis Medical Solutions AG, "Mevislab," [Online]. Available: www.mevislab.de, Accessed on: Oct. 31, 2018

[12] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An End-User tool for visualizing and analyzing very large data," in *Proc. High Perform. Vis.–Enabling Extreme-Scale Sci. Insight*, Oct. 2012, pp. 357–372.

[13] J. Clyne, P. Mininni, A. Norton, and M. Rast, "Interactive desktop analysis of high resolution simulations: application to turbulent plume dynamics and current sheet formation," *New J. Phys.*, vol. 9, no. 8, 2007, Art. no. 301.

[14] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs, "Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations," *IEEE Comput. Graph. Appl.*, vol. 29, no. 6, pp. 6–13, Nov./Dec. 2009.

[15] Thermo Fisher Scientific, "Amira," [Online]. Available: www.fei.com, Accessed on: Oct. 31, 2018

[16] C. Reade, *Elements of Functional Programming*. Reading, MA, USA: Addison-Wesley, 1989.

[17] K. Moreland, "A survey of visualization pipelines," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 3, pp. 367–378, Mar. 2013.

[18] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application*. USA: Kitware, 2015.

[19] J. J. Van Wijk, "The value of visualization," in *Proc. IEEE Vis.*, 2005, pp. 79–86.

[20] R. Westermann and T. Ertl, "Efficiently using graphics hardware in volume rendering applications," in *Proc. ACM 25th Annu. Conf. Comput. Graph. Interactive Techn.*, 1998, pp. 169–177.

[21] D. Jönsson, M. Falk, and A. Ynnerman, "Intuitive exploration of volumetric data using dynamic galleries," *IEEE Trans. Vis. Comput. Graph.*, vol. 22, no. 1, pp. 896–905, Jan. 2016.

[22] C. Konig, R. Skånberg, I. Hotz, A. Ynnerman, P. Norman, and M. Linares, "Binding sites for luminescent amyloid biomarkers from non-biased molecular dynamics simulations," *Chem. Commun.*, vol. 54, pp. 3030–3033, 2018. [Online]. Available: http://dx.doi.org/10.1039/C8CC00105G

[23] S. Kottravel, R. Volpi, M. Linares, T. Ropinski, and I. Hotz, "Visual analysis of stochastic trajectory ensembles in organic solar cell design," *Informat.*, vol. 4, no. 3, 2017, Art. no. 25.

[24] K. Moreland, C. Sewell, W. Usher, L.-T. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, et al.," VTK-M: Accelerating the visualization toolkit for massively threaded architectures," *IEEE Comput. Graph. Appl.*, vol. 36, no. 3, pp. 48–58, May/Jun. 2016.

[25] M. D. Hanwell, U. Ayachit, D. A. Muller, and R. Hovden, "Tomviz," [Online]. Available: www.tomviz.org, Accessed on: Oct. 31, 2018

[26] Advanced Visual Systems Inc., "Avs express," [Online]. Available: www.avs.com/solutions/express, Accessed on: Oct. 31, 2018

[27] A. Forbes, T. Höllerer, and G. Legrady, "Behaviorism: A framework for dynamic data visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 16, no. 6, pp. 1164–1171, Nov.-Dec. 2010.

[28] R. Bujack, J. M. Hlawitschka, M. Hütter, J. Kasten, S. Koch, W. Reich, S. Schlegel, and S. Volke, "FAnToM - Field analysis using topological methods," [Online]. Available: www.informatik.uni-leipzig.de/fantom/, Accessed on: Oct. 31, 2018

[29] D. Gelperin and B. Hetzel, "The growth of software testing," *Commun. ACM*, vol. 31, no. 6, pp. 687–695, 1988.

[30] J. Hoffswell, A. Satyanarayan, J. Heer, S. Kairam, J. Kaye, J. Guerra-Gomez, D. A. Shamma, Y.-S. Kim, J. Hullman, M. Agrawala et al., "Visual debugging techniques for reactive data visualization," *Comput. Graph. Forum*, vol. 34, 2015, Art. no. 3.

[31] Unity Technologies, "Unity," [Online]. Available: www.unity3d.com.com, Accessed: Oct. 31, 2018

[32] Epic Games, "Unreal engine," [Online]. Available: www.unrealengine.com, Accessed: Oct. 31, 2018

[33] R. Sicat, J. Li, J. Choi, M. Cordeil, W.-K. Jeong, B. Bach, and H. Pfister, "DXR: A toolkit for building immersive data visualizations," *IEEE Trans. Vis. Comput. Graph.*, vol. 25, no. 1, pp. 715–725, Jan. 2018.

[34] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*. Reading, MA, USA: Addison-Wesley, 2012.

[35] E. Sundén and T. Ropinski, "Efficient volume illumination with multiple light sources through selective light updates," in *Proc. IEEE Pacific Vis. Symp.*, 2015, pp. 231–238.

[36] D. Jönsson, "Enhancing salient features in volumetric data using illumination and transfer functions," Ph.D. dissertation, Linköping University, Linköping, Sweden, 2016.

[37] D. Jönsson and A. Ynnerman, "Correlated photon mapping for interactive global illumination of time-varying volumetric data," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 901–910, Jan. 2017.

[38] J. Kreiser, J. Freedman, and T. Ropinski, "Visually supporting multiple needle placement in irreversible electroporation interventions," *Comput. Graph. Forum*, vol. 37, no. 6, pp. 59–71, 2018.

[39] M. Falk, A. Ynnerman, D. Treanor, and C. Lundström, "Interactive visualization of 3D histopathology in native resolution," *IEEE Trans. Vis. Comput. Graph.*, vol. 29, no. 1, pp. 1008–1017, Jan. 2019.

[40] S. Kottravel, M. Falk, T. B. Masood, M. Linares, and I. Hotz, "Visual analysis of charge flow network for complex morphologies," *Comput. Graph. Forum*, vol. 38, no. 3, 2019. [Online]. Available: https://diglib.eg.org/handle/10.1111/cgf13704

[41] M. Falk, I. Hotz, P. Ljung, D. Treanor, A. Ynnerman, and C. Lundström, "Transfer function design toolbox for full-color volume datasets," in *Proc. IEEE Pacific Vis. Symp.*, 2017, pp. 171–179.

[42] R. Englund and T. Ropinski, "Evaluating the perception of semi-transparent structures in direct volume rendering techniques," in *Proc. SIGGRAPH ASIA Symp. Vis.*, 2016, Art. no. 9.

[43] R. Englund, S. Kottravel, and T. Ropinski, "A crowdsourcing system for integrated and reproducible evaluation in scientific visualization," in *Proc. IEEE Pacific Vis. Symp.*, 2016, pp. 40–47.

[44] R. Englund and T. Ropinski, "Quantitative and qualitative analysis of the perception of semi-transparent structures in direct volume rendering," *Comput. Graph. Forum*, vol. 37, pp. 174–187, 2018, doi: 10.1111/cgf.13320.

[45] A. Bock, H. Doraiswamy, A. Summers, and C. Silva, "Topoangler: Interactive topology-based extraction of fishes," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 812–821, Jan. 2018.

[46] W. Köpp and T. Weinkauf, "Temporal treemaps: Static visualization of evolving trees," *IEEE Trans. Vis. Comput. Graph.*, vol. 25, no. 1, pp. 534–543, Jan. 2019. [Online]. Available: http://tinoweinkauf.net/publications/abskoepp19a.html

[47] J. Jankowai, B. Wang, and I. Hotz, "Robust extraction and simplification of 2D tensor field topology," *Comput. Graph. Forum*, vol. 38, no. 3, 2019. [Online]. Available: https://diglib.eg.org/handle/10.1111/cgf13693

[48] J. Jankowai and I. Hotz, "Feature level-sets: Generalizing isosurfaces to multi-variate data," *IEEE Trans. Vis. Comput. Graph.*, pp. 1–1, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8453863

[49] R. Englund, T. Ropinski, and I. Hotz, "Coherence Maps for Blood Flow Exploration," in *Eurographics Workshop Visual Comput. for Biol. and Med.*. Bergen, Norway: Eurographics Association, 2016.

[50] R. Englund, K. Lundin Palmerius, I. Hotz, and A. Ynnerman, "Touching data: Enhancing visual exploration of flow data with haptics," *IEEE Comput. Sci. Eng.*, vol. 20, no. 3, pp. 89–100, May./Jun. 2018.

[51] S. Thorstenson, J. Molin, and C. Lundström, "Implementation of large-scale routine diagnostics using whole slide imaging in sweden: Digital pathology experiences 2006–2013," *J. Pathology Informat.*, vol. 5, no. 14, 2014, Art. no. 14.

[52] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-art in GPU-based large-scale volume visualization," *Comput. Graph. Forum*, vol. 34, no. 8, pp. 13–37, 2015.

[53] A. Goode, B. Gilbert, J. Harkes, D. Jukic, and M. Satyanarayanan, "OpenSlide: A vendor-neutral software foundation for digital pathology," *J. Pathology Informat.*, vol. 4, no. 1, p. 27, 2013, doi: 10.4103/2153-3539.119005.

[54] K. T. Nguyen, A. Eklund, H. Ohlsson, F. Hernell, P. Ljung, C. Forsell, M. T. Andersson, H. Knutsson, and A. Ynnerman, "Concurrent volume visualization of real-time fMRI," in *Proc. IEEE/EG Int. Conf. Volume Graph.*, 2010, pp. 53–60.

[55] C. T. Silva, E. Anderson, E. Santos, and J. Freire, "Using VisTrails and provenance for teaching scientific visualization," *Comput. Graph. Forum*, vol. 30, no. 1, pp. 75–84, 2011.

Daniel Jönsson received the MSc and PhD degrees in media technology from Linköping University, Sweden, in 2009 and 2016. Since 2017, he is a research fellow at the division for Media and Information Technology (MIT) at the Department of Science and Technology (ITN), Linköping University. His current main research interests include the intersection between visualization and artificial intelligence.

Peter Steneteg received the PhD degree in theoretical physics from Linköping University, in 2012. He is a principle research engineer in the Scientific Visualization Group at Linköping University. He is currently the project manager and lead core developer of Inviwo.

Erik Sundén received the MSc degree in media technology from Linköping University, in 2010, and worked as a research engineer between 2010 and 2015 within Scientific Visualization. Since 2015 he is Technology Manager at Norrköping Visualization Center C, partly run by the the division for Media and Information Technology (MIT) at the Department of Science and Technology (ITN). He conducts productions and research projects with aims of both state-of-the-art research in visualization as well as public outreach.

Rickard Englund received the master's degree in media technology from Linköping University, in 2014. He is working toward the PhD degree in scientific visualization at Linköping University. His research interests include within scientific visualization with a focus on interactive exploration of temporal vector fields.

Sathish Kottravel received the master's degree in advanced computer graphics from Linköping University, in 2014. He is working toward the PhD degree in Scientific Visualization at Linköping University. His research focus on parallel algorithms for scientific visualization.

Martin Falk received the PhD degree (Dr.rer. nat.) from the University of Stuttgart, in 2013. He is a research fellow in the Scientific Visualization Group at Linköping University. His research interests include large-scale volume rendering, visualizations in the context of pathology and systems biology, large spatio-temporal data, topological analysis, glyph-based rendering, and GPU-based simulations. He is a member of the IEEE.

Anders Ynnerman received the PhD degree in physics from Gothenburg University, 1992. During the early 90s he was doing research at Oxford University, United Kingdom, and Vanderbilt University. In 1996 he started the Swedish National Graduate School in Scientific Computing, which he directed until 1999. From 1997 to 2002 he directed the Swedish National Supercomputer Centre and from 2002 to 2006 he directed the Swedish National Infrastructure for Computing (SNIC). Since 1999 he is holding a chair in scientific visualization at Linköping University and he is the director of the Norrköping Visualization Center - C, which currently constitutes one of the main focal points for research and education in computer graphics and visualization in the Nordic region. The center also hosts a public arena with large scale visualization facilities. He is also one of the co-founders of the Center for Medical Image Science and Visualization (CMIV).

Ingrid Hotz received the PhD degree from the Computer Science Department, University of Kaiserslautern, Germany. She is a professor in Scientific Visualization at the Linköping University in Sweden. Her research interests include the data analysis and scientific visualization, ranging from basic research questions to effective solutions to visualization problems in applications. This includes developing and applying concepts originating from different areas of computer sciences and mathematics, such as computer graphics, computer vision, dynamical systems, computational geometry, and combinatorial topology.

Timo Ropinski received the PhD degree from the University of Münster, Germany, where he also finished his Habilitation. He is a professor in Visual Computing at Ulm University, Germany, where he is heading the Visual Computing Research Group. Before his time in Ulm, he was professor in Interactive Visualization at Linköping University, Sweden. His research interests include the data visualization and visual data analysis. Together with his research group, he works on biomedical visualization techniques, rendering algorithms and deep learning models for spatial data. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.