OpenWalnut – An Open-Source Visualization System

Sebastian Eichelbaum[1], Mario Hlawitschka[2], Alexander Wiebel[3], and Gerik Scheuermann[1]

[1]Abteilung für Bild- und Signalverarbeitung,
Institut für Informatik, Universität Leipzig, Germany

[2]Institute for Data Analysis and Visualization (IDAV), and
Department of Biomedical Imaging, University of California, Davis, USA

[3] Max-Planck-Institut für Kognitions- und Neurowissenschaften, Leipzig, Germany

eichelbaum@informatik.uni-leipzig.de

Abstract

In the last years a variety of open-source software packages focusing on visualization of human brain data have evolved. Many of them are designed to be used in a pure academic environment and are optimized for certain tasks or special data. The open source visualization system we introduce here is called *OpenWalnut*. It is designed and developed to be used by neuroscientists during their research, which enforces the framework to be designed to be very fast and responsive on the one side, but easily extendable on the other side. *OpenWalnut* is a very application-driven tool and the software is tuned to ease its use. Whereas we introduce *OpenWalnut* from a user's point of view, we will focus on its architecture and strengths for visualization researchers in an academic environment.

1 Introduction

The ongoing research into neurological diseases and the function and anatomy of the brain, employs a large variety of examination techniques. The different techniques aim at findings for different research questions or different viewpoints of a single task. The following are only a few of the very common measurement modalities and parts of their application area: *computed tomography* (CT, for anatomical information using X-ray measurements), *magnetic-resonance imaging* (MRI, for anatomical information using magnetic resonance esp. for soft tissues), *diffusion weighted MRI* (dwMRI, for directed anatomical information for extraction of fiber approximations), *functional MRI* (fMRI, for activity of brain areas indicated by the blood-oxygen-level dependence (BOLD) effect) and *electroencephalography* (EEG, for activation of certain brain areas indicated by electric fields).

Considering the different applications, it is evident that, for many research areas, only a combination of these techniques can help answering the posed questions. To be able to analyze data measured by the different techniques, a tool that can efficiently visualize different modalities simultaneously is needed. The software (called *OpenWalnut*) we present in this paper aims at exactly this task. It does not only allow to display different modalities together but also provides tools to analyze their interdependence and relations.

Throughout the paper, we describe the general software architecture, its interactive multi-modal visualization capabilities, and how these make it especially suitable for the task of multi-modal analysis of measurements of the human brain. To obtain a first overview of the context we review some related software.

1.1 Related Software

There exist several visualization packages that are similar to *OpenWalnut* in some aspects or that are designed for similar application areas as *OpenWalnut*. Of the packages we are aware of, MeVisLab ([MVL, 2010]) and Amira ([Amira, 2010]) are the programs that come closest to *OpenWalnut*. Both are based on the principle of data flow networks and provide a *graph widget* that allows the user to manipulate this graph directly. While there is a free version of MeVisLab which provides a large subset of the tool's rich feature set, there exist three different variants of Amira. In addition to the two commercial variants of Amira that slightly differ in focus, there is an academic version developed at the Zuse Institute Berlin, which is freely available for collaborating institutes.

Another open-source tool for visualization of biomedical data is developed at the Scientific Computing Institute (SCI) at the University of Utah. It is part of a larger framework for simulation and visualization called SCIRun ([SCIRun, 2010]). Similar to Amira and MeVisLab it is based on a data flow network.

A major difference of *OpenWalnut* compared to these tools is the visibility and use of the data flow network (called *module graph* in *OpenWalnut*) to users. As the complexity of module graphs can grow very fast, its construction yields a fast increasing barrier for the user. In contrast to other open-source tools (like MeVisLab and SCIRun), *OpenWalnut* can hide this complexity completely from the user and is, therefore, also suitable for scientists who simply want to use visualization tools for their data but are not familiar with or do not want to deal with the visualization internals. SCIRun provides so-called *power apps* to hide this complexity of the data flow environment. These, and similar macros in MeVisLab are very helpful to provide simple user interfaces for special tasks. However, they still have to be created with a script that uses the network in the background. Later in this paper we will describe how *OpenWalnut* combines the best out of two worlds: on the one hand, it provides an easy-to-use graphical user interface (inspired by ParaView [Ahrens et al., 2005]), making it a plug-and-play visualization tool. On the other hand, it provides an optional direct access to the data flow network.

Another package for analyzing imaging data of the human brain is FSL

([FSL, 2008]). It consist of a number of loosely coupled tools and a main GUI for starting sub-tools that serve different tasks like image registration, image visualization, and segmentation. The last application we want to mention here is MedINRIA ([MedINRIA, 2009]), which also provides modules for brain visualization, fiber tracking, and processing of tracking data. All of these tools are integrated in a common windows as user interface, where the window adapts to the chosen task.

Another approach has been chosen by Kindlmann in the *teem* ([teem, 2009]) library. It is not a visualization tool in the sense that it does not provide an interactive graphical user interface, but rather provides a large number of algorithms that are useful for the analysis and visualization of medical imaging data. Its command-line interface communicates through pipes and allows to build simple visualization pipelines and store intermediate data as well as final images in files. As it provides a C interface as well and as it is published under a free software license, other tools, similar to *OpenWalnut*, can benefit directly from teem's data processing capabilities.

The choice for developing *OpenWalnut* from scratch came out of the needs of the neuroscientists at the Max Planck Institute for Human Cognitive and Brain Sciences (MPI CBS) in Leipzig: For their research, they wanted an open-source tool, that is usable for people not familiar with data flow networks and allows for multi-modal visualization of the human brain in a single, coherent environment. Unfortunately, none of the above mentioned tools could fulfill all these needs for them.

Finally, it should be mentioned that there exist even more visualization tools that have a somehow similar approach concerning the user interface but are fitted to other user communities (i.e. not bio/neuro/medical). Examples are Vish [Benger et al., 2007], Mayavi [Enthought Inc., 2010], and the very popular ParaView [Ahrens et al., 2005].

2 Design and Architecture

OpenWalnut's design was mainly steered by two criteria: Firstly, it has to be a powerful and easily expandable framework for visualization researchers allowing them to implement algorithm prototypes and ideas quickly and easily while, secondly, providing an intuitive graphical user interface for neuroscientist researchers who include *OpenWalnut* in their daily research tasks. Whereas the first criterion asks for a flexible and extendible framework, the second criterion introduced the need for a high level of interactivity and responsiveness of the application.

To achieve these ambitious goals, it is important to split functionality and interface. Known and famous in the context of object-oriented programming ([Gamma et al., 1994]) this principle allows a powerful and complex framework under the hood of a simple interface, the GUI in our case. This way, the addition of new modules to the system does not require any changes in the GUI or other parts of the software as they are integrated using abstract interfaces and provide

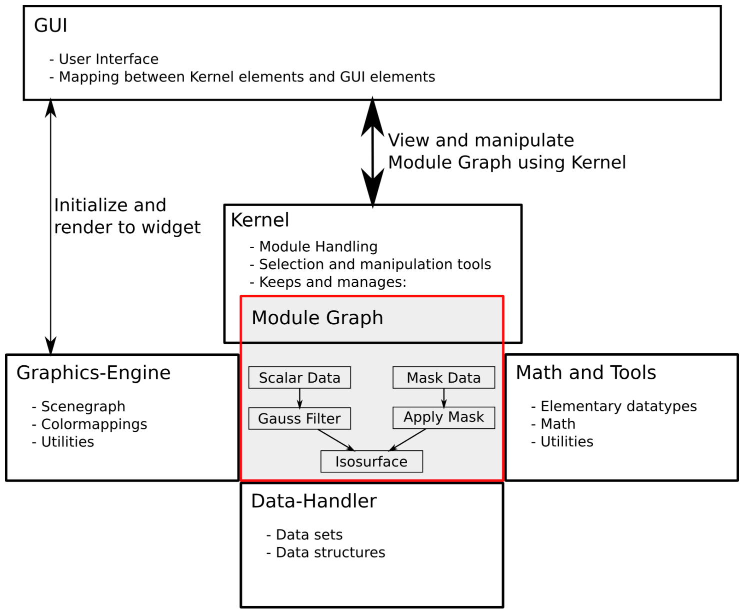


Figure 1: Software architecture. The graphical user interface sits on top of the kernel and maps the module graph and its properties to GUI elements. The modules utilize the core functionality and are handled by the kernel.

their parameters, settings, and I/O information using a standardized interface. To furthermore allow the user to modify data, tune algorithm parameters, or simply load and execute new algorithms while other algorithms are running, a multi-threaded approach is nearly unavoidable. Modules and parts of the basic framework should be able to work independent of each other. We avoid the *data pull* principle, also known as polling, wherever possible, because it causes many synchronization issues. We strongly focus on the push approach. Thus, data changes are not queried by parts of the program, but they are propagated automatically whenever a change occurs. Figure 1 illustrates the architecture of our medical visualization system called *OpenWalnut* and its algorithm-centric layout which is illustrated in the next section in more detail.

2.1 Architecture

This section covers the details of the software architecture, which is shown on an abstract level in Figure 1, and its implementation. The core parts provide the graphics engine, all basic data handling facilities, and basic mathematical and utility functions to the modules. The *kernel* with the *modules* and the module management is stacked on top of it and provides the actual interface implementation used to map the module graph structure and its properties to an end-user interface. From the module-programmer's point of view, the framework is designed entirely based on the aim to make programming easy and to enable us to achieve results in a minimum of time. The data structures provided for module parameters and data exchange allow the modules to provide information using data and parameters in a very abstract fashion without any knowledge of the actual user interface or other modules in the module graph. Therefore, the programmer can focus on module programming, only; no boilerplate code is needed for any kind of GUI interaction or graphics setup.

Graphics Engine

Starting bottom-up, let us first have a closer look at the graphics engine. The graphics engine mainly provides an interface to OpenSceneGraph ([OSG, 2010]). OpenSceneGraph is designed to be used in multi-threaded environments and provides most of the tools and structures required for creating and modifying graphics data. It contains tools to manage large triangle meshes, textures, shaders, and encapsulates most current features of OpenGL ([Khronos, 2010]). This ensures a maximum of flexibility during module development. One example of our extensions of OpenSceneGraph in the *Graphics Engine* are flexible color maps: They can simply be added to any other kind of rendering data, be it geometrical data or not. It automatically manages the loaded data volumes used for color-mapping, their ordering, blending factors, or the actual color map of each volume. This helps the module programmer to provide surface coloring in a clean and straight-forward way and fulfills the basic requirement to quantitatively analyze data.

Data Handler

The data handler provides the different kinds of data sets and data structures. Besides this, the data handler provides supportive algorithms for analyzing data or spatial partitioning of volume data. For example, the data handler provides the tools to convert and scale volume data to be used as texture. These textures and scaling information are then used by the color-mapping facility in the graphics engine to provide proper color mapping for loaded data on arbitrary graphical scene graph elements.

Due to the strong focus of *OpenWalnut* towards medical, especially neurological data, most volumetric data is stored using an implicitly defined grid. Besides this regular three-dimensional data, other kinds of grid structures exist. *OpenWalnut* provides an abstract kind of grid which can be used to implement nearly all other kinds of grid structures. As the grid is stored implicitly, additional transformations are stored inside the grid to ensure that the orientation of a volume in space is right according to the dataset or previously applied registration algorithms.

Additionally, the data handler provides the input and output interfaces to read and write datasets from and to files.

Kernel and GUI

The core component of *OpenWalnut* is the *kernel.* It accommodates running modules, organizes them in a data flow network (or *module graph*), and handles all operations thereon. The most interesting part in the kernel is how modules get integrated into the system. Generally, modules do not have any knowledge about the GUI or other modules in the graph and run in their own thread. Besides this, the literal meaning of graphical user interface might confuse at this point. The task of the GUI, in our case, is mainly to provide the interface to the kernel and to the module graph including its properties. This might be a real graphical representation as *OpenWalnut* provides but can also be a simple command-line interface or a script interface that do not provide graphical output. This is essentially possible due to the abstract command-like interfaces provided in the kernel. In the next paragraph, we will introduce the module graph, the module's communication possibilities, and how this interacts with the GUI.

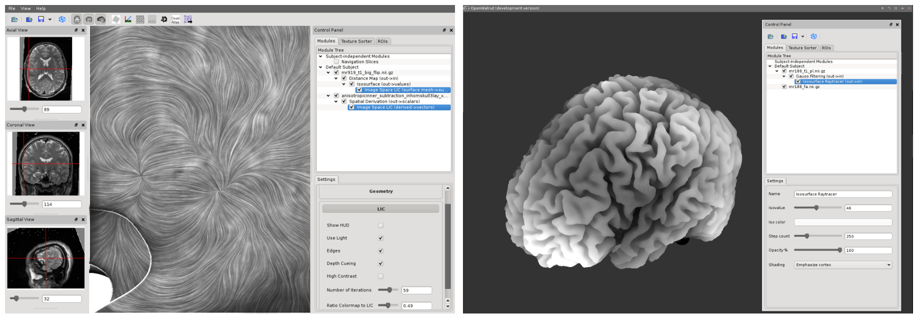
A module has exactly one possibility to interact with other modules residing in the kernel. Modules can define so called *connectors.* These connectors define the input and output channels of a module and define the exact type of data this connector supports. This ensures that modules always get the right kind of data to the correct input, are notified of changes to their input, and can update their output data which may be intermediate data structures or graphics stored in the scene graph. This way, changes in one module propagate along the graph and wake up directly depending modules allowing them again to process the new data. The typed connectors allow the kernel to decide whether module inputs match to a module output. The kernel uses this to provide facilities to the GUI to get lists of compatible connections in other modules or other connection possibilities. The kernel uses this information to create a so called *combiner* which represents the connection or module creation request. This combiner is the abstraction used by the GUI to display those options. As each module and connector provides its name, icon, and description text, the GUI can automatically create a button or menu entries using this information. As module instantiation can take a while, especially if a module does some costly initialization, applying a combiner is done asynchronously. While the kernel processes a combiner, it uses callbacks to inform the GUI about its progress and possible state changes. As mentioned earlier, the whole architecture is designed to use the push mechanism to propagate data, states, and other information. The kernel makes heavy use of this and provides callback and signaling mechanisms for nearly all possible operations. This ensures that the kernel does not need to be polled somehow.

The remaining task is the communication of module parameters and other settings to the user. During the design process, we always avoided that modules have to know the GUI or need to specify their GUI representation directly. Modules therefore are equipped with a mechanism called *properties.* These properties enabled the module developer to simply define parameters or settings without any knowledge of their representation. A module can, for example, define a property of type `double` with a name and a description associated with it. In addition, it can define constraints, for example that it only accepts positive values, to exactly define valid values. The interesting part here is, that it is up to the GUI to decide about the graphical representation of those parameters. To stick with the double-precision floating-point property example, the GUI can decide whether a slider or a text box is more appropriate for a property depending where it is displayed. It therefore mainly uses the constraints defined on a property. Another example would be a property representing a four-times-four matrix that can be represented by sixteen text fields or by several sliders defining rotation, scaling, and translations. Whenever the user modifies a property in the GUI, the property automatically checks whether the new value is valid by using the before-mentioned property constraints. If the value is invalid, the property rejects it and the GUI can somehow show it to the user. If the value is valid, it gets set for the property and the automatic change-propagation ensures that all observers, especially the module owning the property, are notified about the value change. A module can then wake up from its sleep state to handle the new value. It is also possible to use these properties directly in scene graph nodes in conjunction with OpenSceneGraph's callback mechanism to directly modify graphical entities. As the properties implement the *observable pattern*, they can be used in a variety of ways.

With an increasing amount of fine-grained algorithm implementations in modules, the complexity of needed GUI interaction increases tremendously if results of algorithms need to be reused as input for other modules. To circumvent the problem, *OpenWalnut* provides *module containers*. These containers can accommodate multiple modules and forward their connectors and properties. This allows a module container to look and behave like a normal module. The module programmer can re-combine modules in a certain way to map a workflow or visualization use-case without revealing the underlying complexity. This, on the one hand, hides complexity from the user and, therefore, makes the software more intuitive and, on the other hand, allows programmers to reuse existing modules and algorithms.

2.2 Control Panel — Hiding the Module Graph

As mentioned above, *OpenWalnut* hides its module graph by default. Instead, user interaction is performed using a tree widget in the control panel (right in Figure 2). While this tree still allows complete access to the module graph for advanced users (not described in this paper), it provides a simplified interface to *OpenWalnut's* functionality for users who are not interested in the graph. The tree widget acts similar to that of ParaView [Ahrens et al., 2005]: In the first level of the tree the loaded data sets are shown. The branches below indicate the modules that were applied to the data or other modules in higher levels of the tree. To adjust settings of a specific module, a user selects the module and the properties of the module appear in the "Settings" widget below. Clicking onto a module also has a second use: It updates the toolbar (top row in Figure 2) to show only the modules whose input connectors fit the output connectors of



Subject-independent Modules
Navigation Slices
Default Subject
mr919_t1_big_flip.nii.gz
Distance Map (out->in)
Isosurface (out->values)
Image Space LIC (surface mesh->su...
anisotropicinner_subtraction_inhomskull3lay_x...
Spatial Derivation (out->scalars)
Image Space LIC (derived->vectors)

Show HUD
Use Light ✓
Edges ✓
Depth Cueing ✓
High Contrast
Number of Iterations 59
Ratio Colormap to LIC 0.49

Subject-independent Modules
Default Subject
mr188_t1_pl.nii.gz
Gauss Filtering (out->in)
Isosurface Raytracer (out->in)
mr188_fa.nii.gz

Name Isosurface Raytracer
Isovalue 46
Iso color
Step count 250
Opacity % 100
Shading Emphasize cortex

Figure 2: Left: *OpenWalnut's* default GUI. It mainly is split in three areas. The navigation windows on the left showing axial, coronal and sagittal slices through the anatomy which allow easy orientation inside the data. The main 3D view contains the scene itself. Most of the user-interaction with several modules and data can be done in the control panel on the right. It represents how the modules are wired to each other with their connectors and provides an intuitive panel for changing a module's properties. Right: The GUI can be highly customized.

the selected module. Thus, the toolbar always presents all currently applicable modules to the user. Clicking an icon in the toolbar adds the corresponding module in a branch below the currently selected module in the tree and executes the module. To ensure consistency, only leafs in the tree can be removed. This ensures that no module looses the modules it depends on. Removing a module undoes all effects that have been caused by the module.

3 Results and Conclusion

Today, *OpenWalnut* is a visualization tool heavily used by the Max Planck Institute for Human Cognitive and Brain Sciences and the Max Planck Institute for neurological research. Besides the usage as a pure visualization tool, it is also a powerful and handy framework for visualization researchers. In this paper, we gave an overview of *OpenWalnut's* architecture and design which mostly is interesting from the visualization researcher's point of view. We have shown that *OpenWalnut* uses several abstract methods to allow modules to communicate and process data. For visualization researchers, this minimizes efforts and allows them to focus on development of novel algorithms.

The module-graph allows the arbitrary combination of modalities even among multiple subjects, which makes it a very handy tool for neuroscientists during their research tasks for inter-subject and/or multi-modal analyses. The graphical user interface only provides the editing features for the module graph and can provide very distinct GUI elements for the module graph representation or the representation of module parameters and settings. This way, the GUI can be seen as a generalized *view* on top of the functionality of *OpenWalnut*. The GUI itself is designed to be clean, structured and easy to understand but it provides

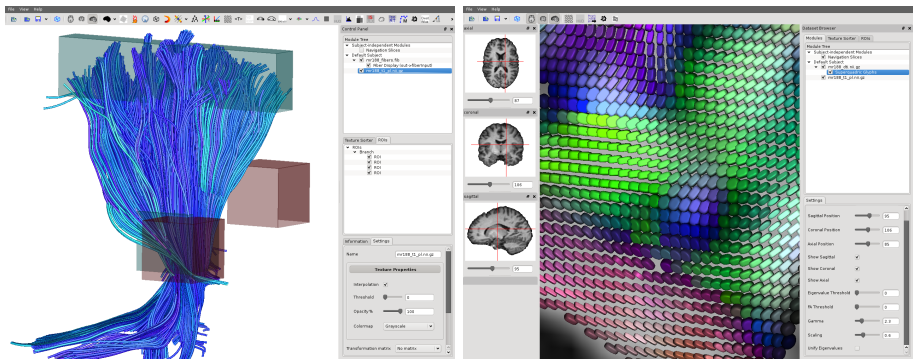


Figure 3: Two screenshots showing *OpenWalnut's* selection tools. On the left, regions of interest (ROI) have been used to select a part of the corticospinal tract of a fiber-tract dataset. On the right, the navigation slices have been used to select and view a slice in a DTI dataset in which superquadric tensor glyphs are shown ([Hlawitschka et al., 2008]).

advanced elements for experienced user.

As *OpenWalnut* is completely open source, it can be used, extended, and customized by everyone to fulfill their needs and provides a framework for testing and implementing algorithms in a very easy way. Unlike other software tools, the strict coding standard and documentation standard ensures a consistent code style and a very detailed documentation of the involved classes. An extensively documented example module helps developers to directly start programming own modules. It is ideal for researchers in the area of visualization and a nice and intuitive tool for visualization users.

The website `http://www.openwalnut.org` provides a lot of information, screenshots, and source code.

Acknowledgments

The authors are grateful for the advice, help and ideas of Christian Heine, Mathias Goldau, Alfred Anwander, and Thomas Knösche. This work was supported by AIF (ZIM grant KF 2034701SS8), NSF CCF-0702817, and NIH R21EB009434-01S1.

References

[Ahrens et al., 2005] Ahrens, J., Geveci, B., & Law, C. (2005). Paraview: An end-user tool for large data visualization. In C. Hansen & C. Johnson (Eds.), *Visualization Handbook*. Elsevier.

[Amira, 2010] Amira (2010). Amira - visualize analyze present. http://www.amira.com/.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The concepts of vish. In 4th *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Enthought Inc., 2010] Enthought Inc. (2010). Mayavi - 3d scientific data visualization and plotting. http://code.enthought.com/projects/mayavi/.

[FSL, 2008] FSL (2008). FMRIB software library. http://www.fmrib.ox.ac.uk/fsl/.

[Gamma et al., 1994] Gamma, E., Helm, R., & Johnson, R. E. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman.

[Hlawitschka et al., 2008] Hlawitschka, M., Eichelbaum, S., & Scheuermann, G. (2008). Fast and memory efficient GPU-based rendering of tensor data. In *Proceedings of the IADIS International Conference on Computer Graphics and Visualization 2008* (pp. 36–42).

[Khronos, 2010] Khronos (2010). The OpenGL Graphics System: A Specification. URL: `http://www.opengl.org`.

[MedINRIA, 2009] MedINRIA (2009). http://www-sop.inria.fr/asclepios/software/MedINRIA/.

[MVL, 2010] MVL (2010). MeVisLab - development environment for medical image processing and visualization. http://www.mevislab.de/.

[OSG, 2010] OSG (2010). OpenSceneGraph. URL: `http://www.openscenegraph.org/`.

[SCIRun, 2010] SCIRun (2010). SCIRun: A scientific computing problem solving environment, scientific computing and imaging institute (SCI). http://www.scirun.org.

[teem, 2009] teem (2009). Teem: Tools to process and visualize scientific data and images. http://teem.sourceforge.net/.