

---

# Getting Started

## First Steps with MeVisLab



## Getting Started

Published April 2009

Copyright © MeVis Medical Solutions, 2003-2009

MeVisLab  
Documentation

---

# Table of Contents

1. Before We Start .....	1
1.1. Welcome to MeVisLab .....	1
1.2. Coverage of the Document .....	1
1.3. Intended Audience .....	1
1.4. Requirements .....	2
1.5. Conventions Used in This Document .....	2
1.5.1. Activities .....	2
1.5.2. Formatting .....	2
1.6. How to Read This Document .....	2
1.7. Related MeVisLab Documents .....	3
1.8. Glossary (abbreviated) .....	4
2. The Nuts and Bolts of MeVisLab .....	6
2.1. MeVisLab Basics .....	6
2.2. Development in MeVisLab .....	7
2.3. MeVisLab Modules .....	8
2.4. Networks .....	9
2.5. Overview of Important Files .....	10
2.6. User Interfaces Controls .....	11
2.7. How to Find More Information on Networks and Modules .....	12
3. Loading and Viewing Images .....	13
3.1. The MeVisLab GUI .....	13
3.2. Searching and Adding Modules .....	14
3.3. Using the ImageLoad Module .....	16
3.4. Adding Viewers to ImageLoad .....	22
3.4.1. Adding the View2D Module .....	22
3.4.2. Adding the View3D Module .....	25
3.5. Alternative Ways to Load Images .....	26
3.5.1. Dragging Images onto the Workspace .....	26
3.5.2. Adding Images via the DICOM Browser .....	27
3.5.3. Using the LocalImage Module .....	27
3.6. A Note on Importing DICOM Images .....	29
4. Implementing a Contour Filter .....	30
4.1. Loading the Input Image .....	30
4.2. Implementing the Contour Filter .....	31
4.3. Parameter Connection for Synchronization .....	35
5. Defining a Region of Interest (ROI) .....	38
5.1. Creating a Viewer with a Selection Rectangle .....	39
5.2. Adding a Second Viewer for the Subimage .....	39
5.3. Adding the Interactivity for the Viewers .....	40
6. Creating an Open Inventor Scene .....	45
6.1. Introduction to Open Inventor .....	46
6.2. Creating the Applicator .....	48
6.3. Creating the Interaction .....	50
6.4. Creating the Anatomical Image .....	53
6.5. Finishing the Complete Open Inventor Scene .....	55
7. Starting Development with Package Creation .....	59
7.1. What are Packages .....	59
7.2. Creating a User Package for Your Project .....	61
8. Introduction to Macro Modules .....	63
9. Developing a Macro Module for an Applicator .....	65
9.1. Creating a Basic Global Macro .....	65
9.2. Adding the Macro Parameters and Panel .....	70
9.3. Programming the Python Script .....	75
9.4. Addition: Shifting the Whole Tip .....	80
10. Excursion: Image Processing in ML .....	84

10.1. Some Advanced Information on Image Processing .....	84
10.2. Structure of MeVisLab .....	84
10.3. Coordinate Systems .....	85
10.4. Affine Transformations .....	86
10.5. DICOM Data and Coordinates .....	87
10.6. Coordinate Systems in the MeVisLab GUI .....	89
10.7. Data Types for DICOM and TIFF .....	91
10.8. Image Processing Concepts: Pages, Slices, VirtualVolumes and more .....	92
11. Introduction to C++ Modules .....	94
11.1. Module and Connection Specifics on the C++ Level .....	94
11.2. Some Tips for Module Design .....	95
11.2.1. Macro Modules or C++ Modules? .....	95
11.2.2. Combining Functionalities .....	95
11.2.3. Tips for Module Testing .....	96
11.3. Programming Examples .....	96
12. Developing ML Modules .....	98
12.1. Creating a New ML Module for Adding Values .....	98
12.1.1. Creating the Basic ML Module with the Project Wizard .....	98
12.1.2. Preparing the Project .....	102
12.1.3. Programming the Functions of the ML Module .....	104
12.1.4. GUI Creation/Optimizing .....	105
12.1.5. Creating an Example Network and Help File .....	106
12.2. Creating an ML Module For Simple Average .....	107
12.2.1. Creating the Basic ML Module with the Project Wizard .....	108
12.2.2. Editing the Header File of SimpleAverage .....	109
12.2.3. Editing the CPP File of SimpleAverage .....	109
12.2.4. Testing the Module .....	111
12.3. Combining Two Modules in One Project .....	111
12.3.1. Copying the Souce Files .....	111
12.3.2. Editing and Recompiling the .pro File .....	111
12.3.3. Editing the Project in the Development Environment .....	112
12.3.4. Editing the Module Definition (.def) .....	113
12.3.5. Cleaning up Folders and Example Networks .....	113
13. Developing Inventor, WEM and CSO Modules .....	114
13.1. Inventor Modules .....	114
13.2. Winged Edge Mesh Library (WEM) .....	114
13.3. Contour Segmentation Objects (CSO) .....	116

---

## List of Figures

1.1. Welcome Screen and Documentation Links .....	4
2.1. Image Processing Pipeline .....	8
2.2. Network Layout .....	10
2.3. Module Context Menu: Show Help .....	12
3.1. MeVisLab User Interface .....	13
3.2. Viewer Selection .....	14
3.3. Modules Menu and Module Browser .....	15
3.4. Quick Search Options .....	16
3.5. Quick Search Results .....	16
3.6. ImageLoad Module .....	16
3.7. ImageLoad Panel and Output Inspector .....	17
3.8. Adjusting the Windowing .....	18
3.9. Output Inspector with Image Properties .....	19
3.10. Output Inspector with Additional Information Display .....	20
3.11. 3D Output Inspector .....	20
3.12. Connector Details in the Edit Menu .....	21
3.13. Connector Details in the Preferences .....	21
3.14. Connector Details Depending on Zoom .....	22
3.15. Setting up the Connection .....	23
3.16. Panel of View2D .....	23
3.17. Opening the Settings Panel of View2D .....	24
3.18. Settings Panel of View2D .....	24
3.19. Automatic and Settings Panel of View2D .....	25
3.20. Connecting the View3D Module .....	26
3.21. The View3D Panel .....	26
3.22. DICOM Browser .....	27
3.23. LocalImage Module .....	28
3.24. Show the Internal Network .....	28
3.25. Internal Network of the LocalImage Module .....	28
3.26. DicomImport .....	29
4.1. Example Network Contour Filter .....	30
4.2. Viewing the Input Image for the Contour Filter .....	31
4.3. Adjust Filter Parameters .....	32
4.4. Constructing the Filter Pipeline — Convolution Output .....	33
4.5. Constructing the Filter Pipeline — Morphology Output .....	33
4.6. Constructing the Filter Pipeline — Arithmetic2 Output .....	34
4.7. Creating a New Group .....	34
4.8. Resulting Contour Filter Network .....	35
4.9. Establishing the Parameter Connections .....	36
4.10. Resulting Network .....	37
5.1. Example Network ROISelection .....	38
5.2. Viewer with Selection Rectangle .....	39
5.3. Viewer for the Subimage .....	40
5.4. Searching for World to Voxel Conversion .....	40
5.5. WorldVoxelConvert Panel .....	41
5.6. WorldVoxelConvert Modules Added .....	42
5.7. Adding the Parameter Connections .....	43
5.8. Example Network ROI Selection .....	44
6.1. Example Network: Open Inventor Result .....	45
6.2. Applicator Only .....	46
6.3. Traversing in Open Inventor .....	47
6.4. Creating the Applicator Shaft .....	48
6.5. Coloring the Applicator Shaft .....	49
6.6. Adding an Applicator Tip .....	49
6.7. Adding Translation and Grouping .....	50

6.8. Finishing the Applicator .....	50
6.9. Using SoCenterballManip .....	51
6.10. Connecting Parameters .....	52
6.11. Combining Interaction and Applicator .....	52
6.12. Loading a Local Image .....	53
6.13. Adding the GigaVoxel Renderer .....	53
6.14. Copying the Windowing Modules from View3D .....	54
6.15. Adding the Windowing to the Applicator .....	54
6.16. Combining the Groups .....	55
6.17. Combined Graphic Elements .....	56
6.18. Adding the Applicator Scaling .....	57
6.19. Original Applicator/Interaction Arrangement .....	57
6.20. Improved Applicator/Interaction Arrangement .....	58
7.1. Example for a Package Tree .....	59
7.2. Preferences — Packages .....	60
7.3. Package Wizard .....	61
9.1. Starting a new Macro from the Existing Applicator .....	65
9.2. Renaming Instance Names .....	66
9.3. Creating a Local Macro .....	67
9.4. Selecting a Genre .....	68
9.5. Module Properties .....	69
9.6. File Browser with the New Macro Module Files .....	70
9.7. ApplicatorMacro as Macro Module .....	70
9.8. ApplicatorMacro.script in Mate .....	70
9.9. ApplicatorMacro Module with Output Connector .....	71
9.10. Internal Network of the ApplicatorMacro Module .....	72
9.11. Automatic Panel of the ApplicatorMacro Module .....	73
9.12. Panel of the ApplicatorMacro Module .....	74
9.13. Parameters for Diameter Setting .....	76
9.14. Changing the Diameter of the Applicator .....	77
9.15. Strange Behavior of the ApplicatorMacro .....	78
9.16. Adding the Correct Tip Translation .....	79
9.17. Complete ApplicatorMacro .....	79
9.18. Feeding the SoCalculator Module .....	81
9.19. Improved Applicator Macro Module .....	82
10.1. MeVisLab Structure .....	84
10.2. Coordinate Systems .....	85
10.3. Matrix Multiplication .....	86
10.4. World Coordinates in Context of the Human Body .....	88
10.5. The DICOM Tag Viewer .....	89
10.6. Image Properties for an Ideal Image .....	90
10.7. Image Properties for a Sagittal Image .....	90
10.8. Image Properties in the Info Module .....	91
12.1. Entering the ML Module Properties I .....	99
12.2. Entering the ML Module Properties II .....	100
12.3. Entering the ML Module Properties — Fields .....	101
12.4. Project in Visual C++ 2005 .....	102
12.5. Project in Xcode .....	103
12.6. Project in Code::Blocks .....	104
12.7. Example Network for SimpleAdd .....	107
13.1. WEM IsoSurface Example Network .....	115
13.2. WEM Extrude Example Network .....	115
13.3. Freehand Contours with the SoView2CSOEditor Example Network .....	116

---

## List of Tables

1.1. Related Documents .....	3
2.1. Module Types .....	8
2.2. Connectors .....	9
2.3. Connections .....	9
2.4. Important Files .....	11

---

# Chapter 1. Before We Start

## 1.1. Welcome to MeVisLab

MeVisLab is a rapid prototyping and development platform for medical image processing and visualization. With its image processing library, it fulfills the following requirements:

- Able to handle large, six-dimensional images (x, y, z, color, time, user-defined).
- Offers easy ways to develop new algorithms or changing/improving existing ones in a modular C++ interface, perfect for a fast-developing research area.
- Offers easy ways of combining algorithms to algorithm pipelines and networks.
- Fast and easy integration into clinical environments due to standard interfaces, e.g. to DICOM.
- Fair performance for clinical routine due to a page-based, demand-driven approach in the image processing.

Beside general image processing algorithms and visualization tools, MeVisLab includes advanced medical imaging modules for segmentation, registration, volumetry and quantitative morphological and functional analysis.

Based on MeVisLab, several clinical prototypes have been developed, including software assistants for neuro-imaging, dynamic image analysis, surgery planning, and vessel analysis.

The implementation of MeVisLab makes use of a number of well known third-party libraries and technologies, most importantly the application framework Qt, the visualization and interaction toolkit Open Inventor, the scripting language Python, and the graphics standard OpenGL. In addition, modules based on the Insight ToolKit (ITK) and the Visualization ToolKit (VTK) are available.

## 1.2. Coverage of the Document

Reading this document you will become familiar with the basic features of MeVisLab and how to use them. The chapters are going from the easy to the complex, from the visual programming to macros and programming modules in C++. You will get an idea of how to

- work with the graphical module/network interface concept of MeVisLab
- load and view 2D, 3D and 4D images of various formats
- prototype your specific image processing, image visualization or image interaction tasks with a standard set of modules provided by the SDK distribution
- let your own image processing C++-algorithms run in MeVisLab as self-defined module plug-ins
- create compact graphical user interface representations of your image processing and image visualization pipelines, functioning as quasi-applications



### Note

Depending on your software license, not all features of MeVisLab may be available. For licensing information, please refer to the MeVisLab website (<http://www.mevislab.de/>).

## 1.3. Intended Audience

Getting Started is aimed at people new to MeVisLab and those who want to explore more of its options.



The necessary prior knowledge depends on the MeVisLab usage:

- For pure network creation, no programming knowledge is required.
- For macro creation, basic knowledge of Python or JavaScript and the MDL (MeVisLab Definition Language) is required.
- For developing modules, basic C++ knowledge is required.
- For using the visualization options to their best advantage, some knowledge of image processing and computer graphics is required.

## 1.4. Requirements

It is assumed that you have a working installation of the MeVisLab SDK distribution with a standard set of modules. Supported platforms are Windows, Linux and Mac OS X. A complete overview of supported platforms and compilers can be found at the MeVisLab website (<http://www.mevislab.de/>).

## 1.5. Conventions Used in This Document

### 1.5.1. Activities

**Select:** Click an object with the left mouse button.

**Right-click:** Click an object with the right mouse button, usually to open the context menu.

**Double-click:** Click the object twice in fast repetition. Starts the default action of the object (e.g. for a module, opens the default panel).

**Drag:** Click the object with the mouse and keep the mouse button pressed while moving the object to another position. Place/stop by releasing the mouse button.

**Right-drag:** Click the object with the right mouse button and keep it pressed while moving (as described for drag).

**CTRL+N:** Press the keys CTRL and N at the same time.

**ALT + double-click:** Press the ALT key and double-click the object.

**Menuitem** → **Submenuitem:** Open the menu and select the submenu item.

### 1.5.2. Formatting

Views: *Parameter Connections Inspector*

MeVisLab modules: *ImageLoad*:

Parameters: *Diameter*

Programming code: `*outVoxel = *in0Voxel` and also

```
outMin = inMin + constValue
outMax = inMax + constValue
```

## 1.6. How to Read This Document

If these are your first steps with MeVisLab, start with [Chapter 2, The Nuts and Bolts of MeVisLab](#) and proceed to the first network example [Chapter 3, Loading and Viewing Images](#).

If you have basic experience with image processing and want to learn more about visualization and scenes in Open Inventor, read [Chapter 6, Creating an Open Inventor Scene](#).

If you have basic experience with all module types in MeVisLab and think about extending your networks with scripting, read [Chapter 9, Developing a Macro Module for an Applicator](#).

If you have basic experience with the possibilities of MeVisLab networks and think about programming your own modules in C++, start with [Chapter 11, Introduction to C++ Modules](#).

In addition, the following sections might be of help:

- [Chapter 10, Excursion: Image Processing in ML](#) for some background on coordinate systems and how they are used in MeVisLab.
- [Chapter 7, Starting Development with Package Creation](#) for the package structure of the module database and how to create your own packages for development.

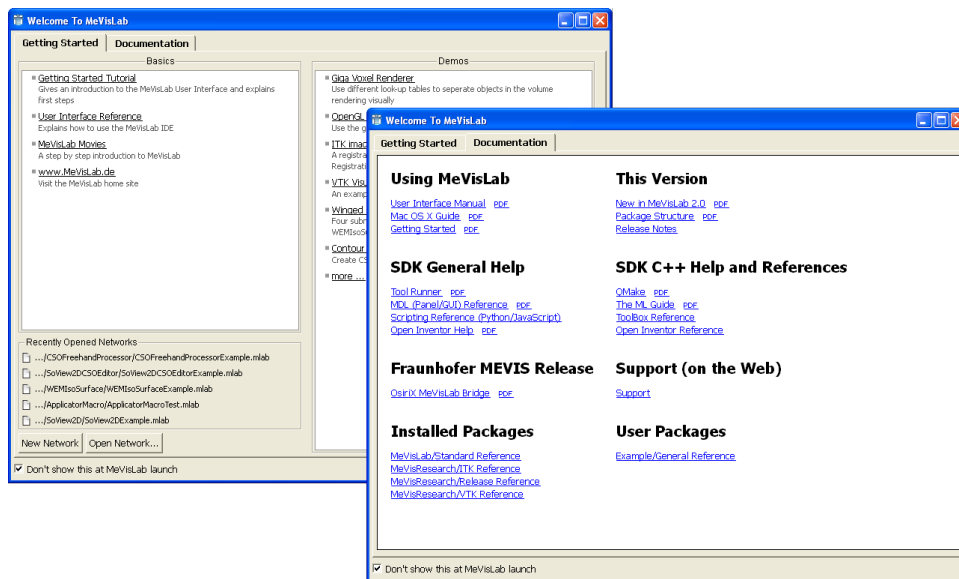
## 1.7. Related MeVisLab Documents

Besides the document at hand, a number of other documents are available.

**Table 1.1. Related Documents**

Title	Contents
MeVisLab Reference Manual	Reference for the MeVisLab user interface
MDL Reference	MeVisLab Definition Language (MDL) reference
ML Guide	MeVis Library Programming Guide
ML Reference (HTML only)	Collected help texts for all modules
Inventor Module Help	Help for Open Inventor modules
Toolbox Reference	MeVisLab Toolbox Class Reference for various libraries
MeVisLab - Mac OS X Guide	Details for MeVisLab on Mac OS X
ToolRunner	Manual for ToolRunner, a stand-alone program delivered with MeVisLab 2.0
qmake	qmake in the MeVisLab context, including explanations for <code>.pro</code> and <code>.pri</code> files

The full list of available documents and resources is available on the Welcome Screen (which can also be opened via **Help** → **Welcome**). While the Getting Started tab offers links to some important resources and demos, the Documentation tab links to all documentation (HTML and PDF, if available).

**Figure 1.1. Welcome Screen and Documentation Links****Tip**

On the Documentation tab, you can also find the help files for all installed packages and your user packages listed. This is possible because the documentation links are created dynamically for your installation. For more information on packages, see [Chapter 7, Starting Development with Package Creation](#).

For all questions related to programming that are not covered by the documentation, please refer to the MeVisLab forum where you can search old topics or post new questions.

## 1.8. Glossary (abbreviated)

For an extensive glossary, see the ML Guide.

### ML, MDL, Open Inventor — Some Important Terms Explained

Base	Base fields/objects, for example the connectors for base objects. Base connectors handle pointers to an abstract data object defined by the user. How the base object is handled depends on how it is integrated in the module.
BaseOp	The base class (superclass) of all ML modules (page-based, demand-driven). Not to be confused with the base object described above. WEM and CSO modules are also derived from BaseOp.
ITK™	The Insight Segmentation and Registration Toolkit™. A large, well known, open source image processing library which has been wrapped in many parts for MeVisLab to work seamlessly with other ML modules. See <a href="http://www.itk.org">www.itk.org</a> and <a href="http://www.mvislab.de">www.mvislab.de</a> for details.
ML	MeVis Image Processing Library, also called MeVis Library at times.
MDL	MeVis Description Language, the language in which user interfaces of modules and applications are written.

MFL	Formerly the MeVisLab File Library, the library that is used for reading and writing any image format (for example, DICOM/TIFF). As of MeVisLab 2.0, it is named “MLImageIO”.
MeVisLab IDE	The Integrated Development Environment.
Open Inventor	Object-oriented 3D toolkit on top of OpenGL, a library of objects and methods used for interactive 3D graphics
VTK™	The Visualization Toolkit™. A large, well known, open source visualization library which has been wrapped in many parts to work also in MeVisLab. See <a href="http://www.vtk.org">www.vtk.org</a> and <a href="http://www.mevislab.de">www.mevislab.de</a> for details.

---

# Chapter 2. The Nuts and Bolts of MeVisLab

In the following chapter, we give you a brief (and dry) introduction into the nuts and bolts of MeVisLab, that is:

- [Section 2.2, “Development in MeVisLab”](#)
- [Section 2.3, “MeVisLab Modules”](#)
- [Section 2.4, “Networks”](#)
- [Section 2.5, “Overview of Important Files ”](#)
- [Section 2.6, “User Interfaces Controls”](#)
- [Section 2.7, “How to Find More Information on Networks and Modules”](#)

## 2.1. MeVisLab Basics

Some of the most prominent features of MeVisLab:

- Full 6D image processing (x, y, z, color, time, user dimensions)
- Paging
- Caching
- Multithreading support
- Platform-independent
- Scripting support (Python and JavaScript)
- Macro system
- Defining of GUI elements with the MDL scripting language
- C++ programming interface
- Pure C++ and object-oriented design
- Self-descriptive module and application interfaces
- Error handling: configurable exception usage; configurable error handling; diagnosis modules, automatic module tester
- Runtime type system
- Extensible voxel type
- Resources-friendly memory usage
- Supports highly complex module networks
- Based on standard libraries

- Currently about 1300 modules
- Long time maintenance

## 2.2. Development in MeVisLab

In MeVisLab, development can be done on three levels:

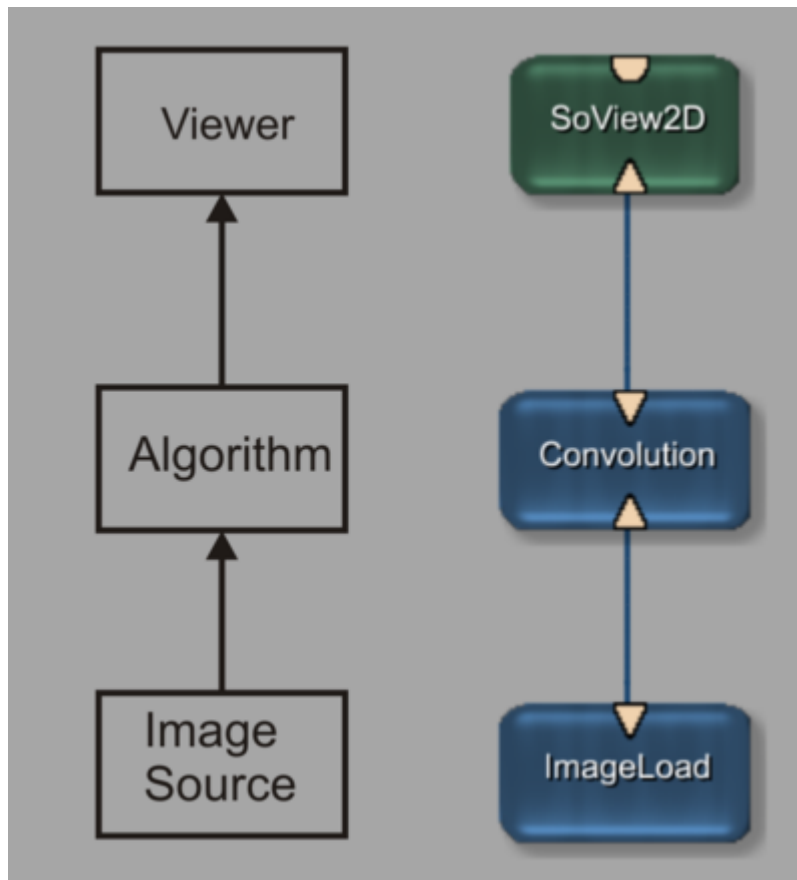
- **Visual level:** Programming with “plug and play”: Individual image processing, visualization and interaction modules can be combined to complex image processing networks using a graphical programming approach.
- **Scripting level:** Creating macro modules and applications based on macro modules: Python or JavaScript scripting components can be added to implement dynamic functionality on both the network and the user interface level.
- **C++ level:** Programming modules: New algorithms can easily be integrated using the modular, platform-independent C++ class library.

In addition, the abstract, hierarchical MeVisLab Definition Language (MDL) allows designing efficient graphical user interfaces, hiding the complexity of the underlying module network to the end user.

From a workflow point of view, an application development would look as follows:

1. Connect existing modules to networks.
2. Develop new modules, if necessary
3. Build user interface (GUI).
4. Build macro modules to recycle complex functionality.
5. Use scripts to control networks, GUIs and macros.
6. Build installer (only with a special ADK license).

In MeVisLab, the algorithms are visualized in a network of modules (graphs). In a minimalist approach, an image processing pipeline would consist of an image source, some algorithm/image processing step in the middle and a viewer for displaying the output. This pipeline is mirrored in the MeVisLab GUI.

**Figure 2.1. Image Processing Pipeline**

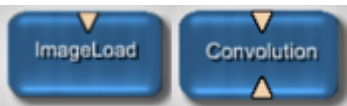


Modules can be connected in various ways which will be described in the following paragraphs.

## 2.3. MeVisLab Modules

Within the concept of MeVisLab the basic entities we are working with are graphical representations of modules with their specific functions for image processing, image visualization and image interaction.

The three basic module types (ML, Inventor and macro) are distinguished by their colors:

**Table 2.1. Module Types**

Type	Look	Characteristics
ML Module (blue)		page-based, demand-driven processing of voxels
Open Inventor Modules (green)		visual scene graphs (3D); naming convention: all modules starting with "So"
Macro Module (brown)		combination of other module types, allowing implementing hierarchies and scripted interaction

Most modules have connectors which are displayed on the module. These represent the inputs (bottom) and outputs (top) of modules.

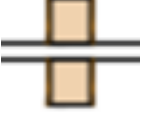

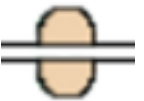
In MeVisLab, three types of connectors are defined.



### Note

In principle, every module type can have any kind of connector.

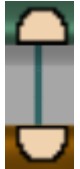

**Table 2.2. Connectors**

Look		Definition
	square	Base objects: pointers to data structures
	triangle	ML images
	half-circle	Inventor scene

By connecting these connectors and therefore establishing a so-called data connection, image data or Open Inventor information is transported from one module to one or more others.

Besides connecting connectors, any field of modules can be connected to other compatible fields of modules with a parameter connection.

**Table 2.3. Connections**

Type	Look	Characteristics
Data connections (connector connections)		The direct connection between connectors. Depending on which connectors are involved, the connection is rendered in a different color: blue for ML, green for Open Inventor, brown for Base.
Parameter connections (field connections)		Connections created by connecting parameter fields within or between modules



### Tip

For more display options, see the MeVisLab Reference Manual, chapter “Modules and Networks”.

## 2.4. Networks

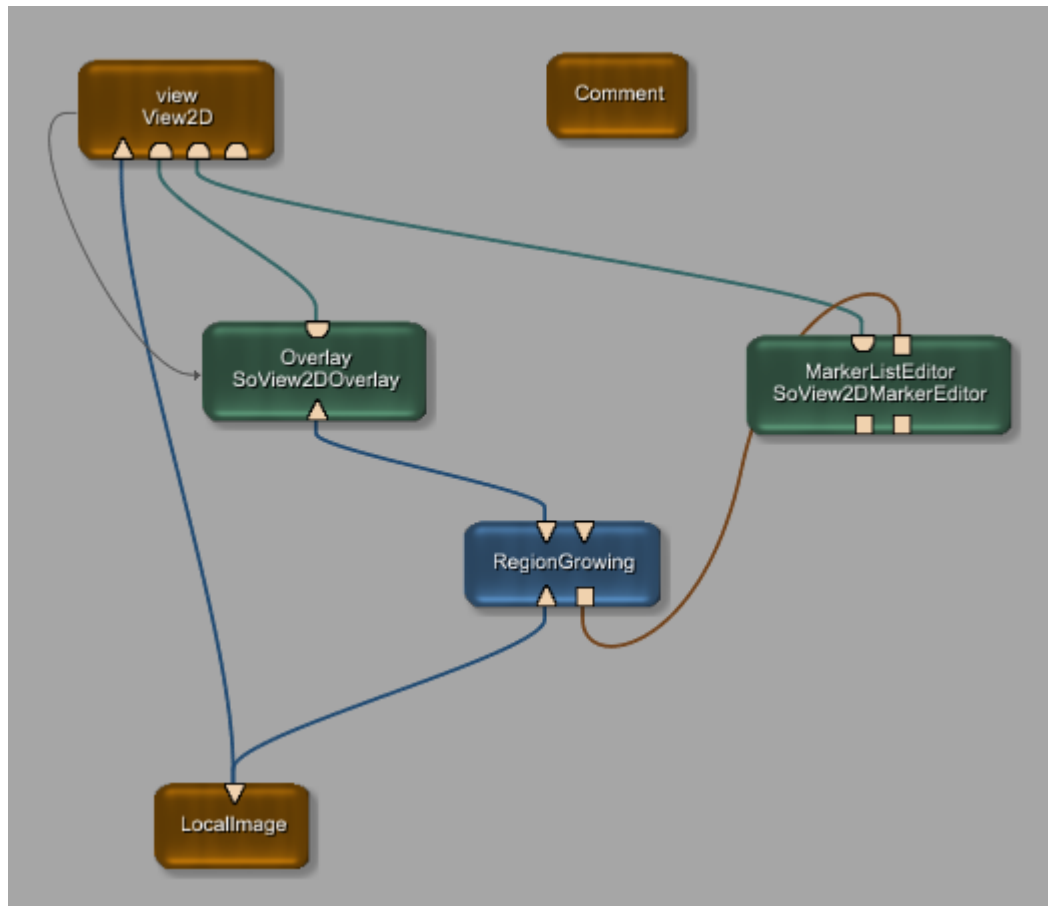
Networks are connections between modules with which you can implement complex processing tasks from sets of standard ML, Inventor, WEM, CSO, ITK, or VTK modules.

Networks are edited and saved as \*.mlab files in MeVisLab.



In [Figure 2.2, “Network Layout”](#), the example network of the `RegionGrowing` module is shown. It consists of all three types of modules and shows data connections as well as parameter connections.

**Figure 2.2. Network Layout**



Remember that macro modules are encapsulated networks of their own, so you effectively work with subnetworks (see [Chapter 8, Introduction to Macro Modules](#) for more information).



### Tip

For information on the involved classes for the programming of modules, connectors and connections, see [Chapter 11, Introduction to C++ Modules](#).

## 2.5. Overview of Important Files

Here a list of the most important file types:

**Table 2.4. Important Files**

File type	Contents
.mlab	Network file, includes all information about its modules and their connections and settings.
.def	Module definition file, necessary for a module to be added to the common MeVisLab module database. May also include all MDL script parts (if they are not sourced out to the .script file).
.script	MDL script file, typically includes the user interface definition for panels. See <a href="#">Section 9.2, “Adding the Macro Parameters and Panel”</a> for an example on GUI programming.
.py	Python file, used for scripting in macro modules. See <a href="#">Chapter 9, Developing a Macro Module for an Applicator</a> for an example on macro programming.
.js	JavaScript file, used for scripting in macro modules.
.dcm	DCM part of the imported DICOM file, see <a href="#">Section 10.7, “Data Types for DICOM and TIFF”</a> .
.tiff	TIFF part of the imported DICOM file, see <a href="#">Section 10.7, “Data Types for DICOM and TIFF”</a> .

For files related to module programming in C++, see [Chapter 11, Introduction to C++ Modules](#).

## 2.6. User Interfaces Controls

MeVisLab uses QT for rendering the GUI (panels etc.) and offers a scripting interface.

Every module comes with an automatic panel on which all fields and available settings are listed.

For improving the handling, user interfaces (“panels”) can be added for modules, see [Figure 3.19, “Automatic and Settings Panel of View2D”](#) for an example. Panels are written in MDL and offer the following possibilities:

- layouting and grouping of fields
- excluding some of the available fields (to make the panels more user-friendly)
- adding additional fields
- adding additional functionality by calling script methods

The components of the user interface are controls.

- User input controls, like text and number edit controls; popup menus, radio buttons, checkboxes, and trigger buttons. They are typically, but not necessarily linked to a field. Several controls can be linked to the same field.
- Layout controls, like for horizontal/vertical grouping
- Decoration controls, complex controls, dynamic controls...

To these controls, scripting can be added.

An example for the programming of a small module panel is given in [Section 9.2, “Adding the Macro Parameters and Panel”](#).



## Tip

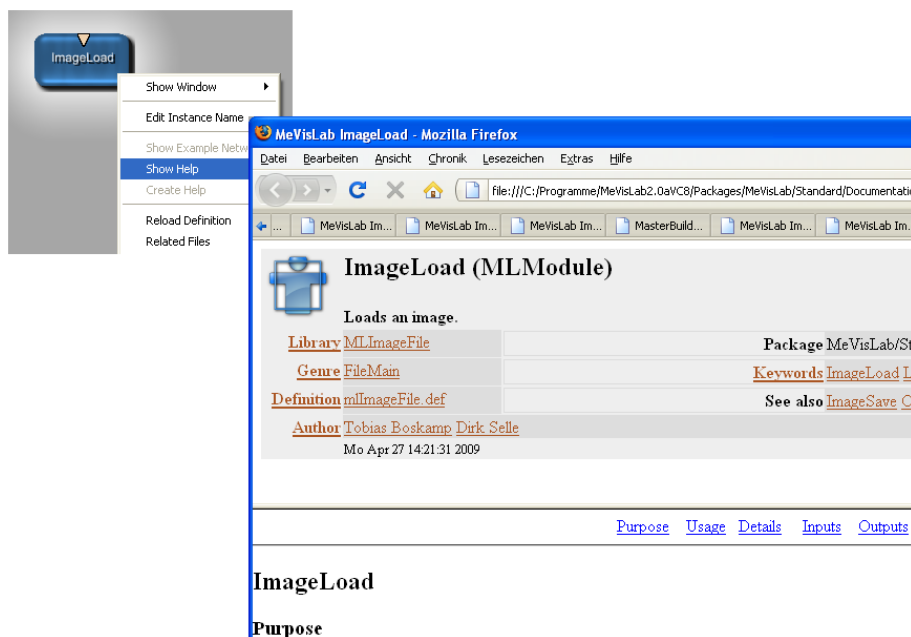
Example GUI modules are available; enter “Test” in the quick search to get a list of available modules.

For further details on panel scripting, please refer to the MDL Reference.

## 2.7. How to Find More Information on Networks and Modules

1. When you enter the module name in the quick search, the About information of the module is displayed.
2. If the View **Module Inspector** is open, you can find the About information on the respective tab.
3. To get a detailed description of the module's function and how to use it, refer to its help file.
  - a. Right-click the module to open the context menu.
  - b. Select **Show Help** to open the module's HTML help in your default browser.

**Figure 2.3. Module Context Menu: Show Help**



4. To see how the module is working, an example network is delivered with most modules.
  - a. Right-click the module to open the context menu.
  - b. Select **Show Example Network** to open the example network on another network tab.

# Chapter 3. Loading and Viewing Images

In the following chapter, we will walk through an example network for loading and viewing images.

- [Section 3.1, “The MeVisLab GUI”](#): first steps in the MeVisLab user interface
- [Section 3.2, “Searching and Adding Modules”](#): searching and finding modules
- [Section 3.3, “Using the ImageLoad Module”](#): loading images
- [Section 3.4, “Adding Viewers to ImageLoad”](#): adding viewers (View2D and View3D)

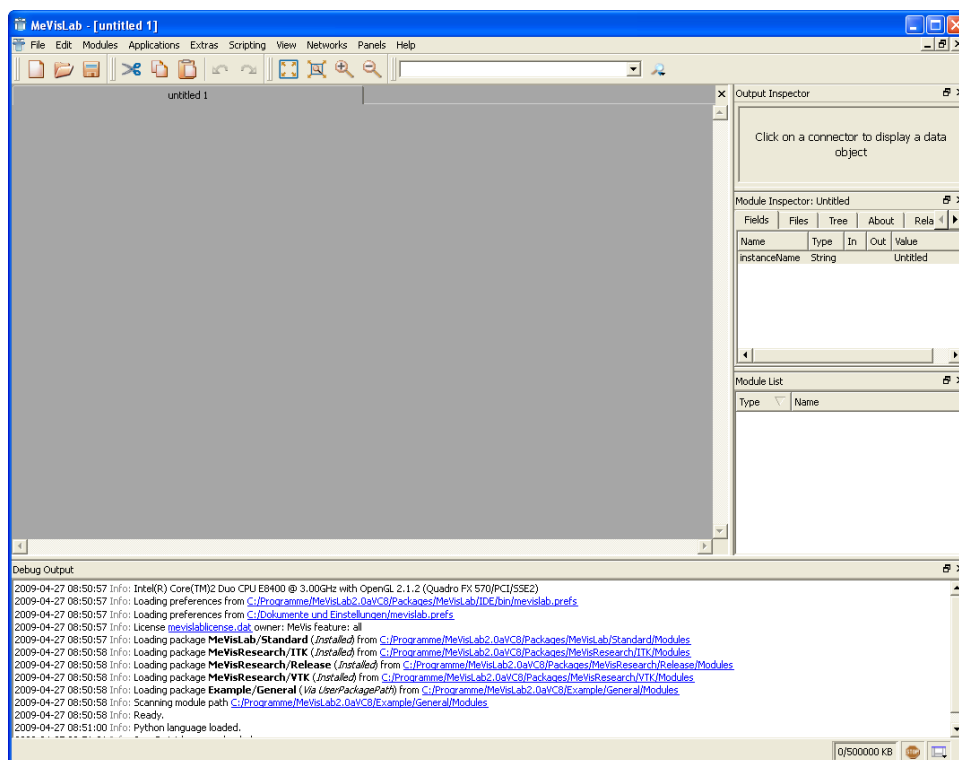
In addition, two special topics are discussed:

- [Section 3.5, “Alternative Ways to Load Images”](#): alternative ways to load images
- [Section 3.6, “A Note on Importing DICOM Images”](#): importing and converting DICOM images to the internal image format of MeVisLab

## 3.1. The MeVisLab GUI

First, start MeVisLab (the “how” depends on your platform). After the Welcome Screen (see [Figure 1.1, “Welcome Screen and Documentation Links”](#)), the start view opens.

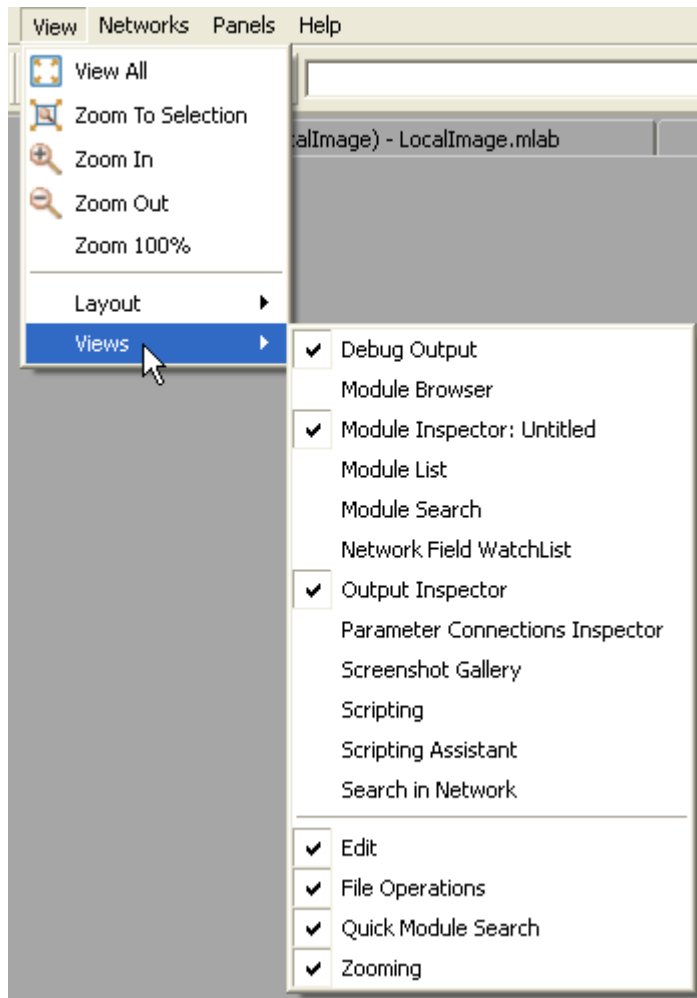
Figure 3.1. MeVisLab User Interface



By default, MeVisLab starts with an empty workspace and some Views on the right (like the **Output Inspector**) and bottom of the screen (usually the **Debug Output**). In the **Debug Output**, you can find information about your MeVisLab installation and start-up, which preferences and license file are loaded, and if all packages loaded correctly or with errors.

Views can be configured via the menu bar, **View** → **Views**, or by a right-click on the border of Views.

**Figure 3.2. Viewer Selection**



Some Views arrangement are pre-defined as layouts, which can be selected via **View** → **Layout**. If you are working in the **User Default Layout**, all changes you make in the Views configuration are persistent and will be saved as your “User Default Layout”. Therefore, most screenshots in the MeVisLab documentation are only examples — your own MeVisLab GUI may look different. Only the workspace always remains visible.



### Tip

For details on layouts, see the MeVisLab Reference Manual, chapter “Menu Bar”.

The workspace is the place for constructing and editing module networks. If more than one network is open, tabs appear on top of the workspace. To create, open and save one or more networks, use the tool bar buttons or the **File** menu in the menu bar. To switch between different network tabs, use the **Networks** menu in the menu bar or press **Tab**.

For more detailed information, see the following examples and the MeVisLab Reference Manual.

## 3.2. Searching and Adding Modules

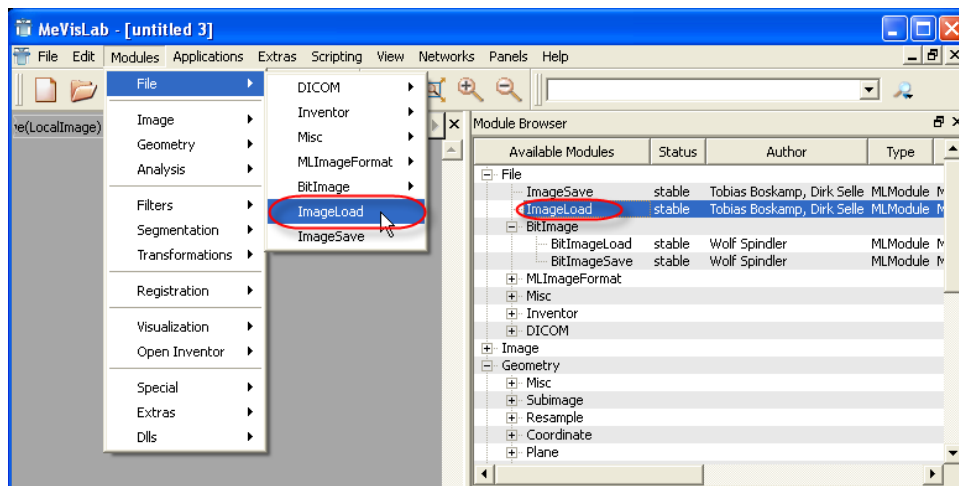
There are several ways to add a module to the current network, for example:

- via the menu bar, entry **Modules**.
- via the menu bar, **Quick Search**.
- via the View **Module Search**.
- via the View **Module Browser**.
- via copy and paste from another network.
- by scripting, see the Scripting Reference.

Both the **Modules** menu and the **Module Browser** display all available modules. The modules are sorted hierarchically by topics and by module name, as given in the file `Genre.def`.

Therefore, both places are a good starting point when in need of a specific function, like an image load module.

**Figure 3.3. Modules Menu and Module Browser**



The last entry **DLL** lists the modules by their main DLL name.

The advantage of the **Module Browser** is that you can right-click the entries, open the context menu and, for example, open the help (in your default Internet browser) or the module files (in Mate, the in-built text editor).



### Note

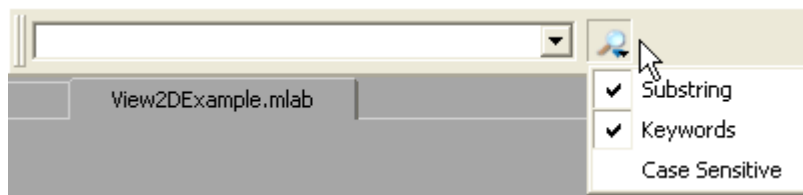
For a module to get listed, it has to be available in the SDK distribution or in your user-defined packages. If in doubt or missing something, check out the loaded packages in the Preferences (on Windows and Linux: **Edit** → **Preferences** → **Packages**; on Mac OS X: **MeVisLab** → **Preferences** → **Packages**). For details on packages, see [Chapter 7, Starting Development with Package Creation](#).

Usually the quickest way to add modules to a network is the quick search in the menu bar. It offers you the possibility to search for modules by module name. By default, the search will also be extended to keywords and substrings and is case-insensitive. To change these settings, click the magnifier button for the search options.

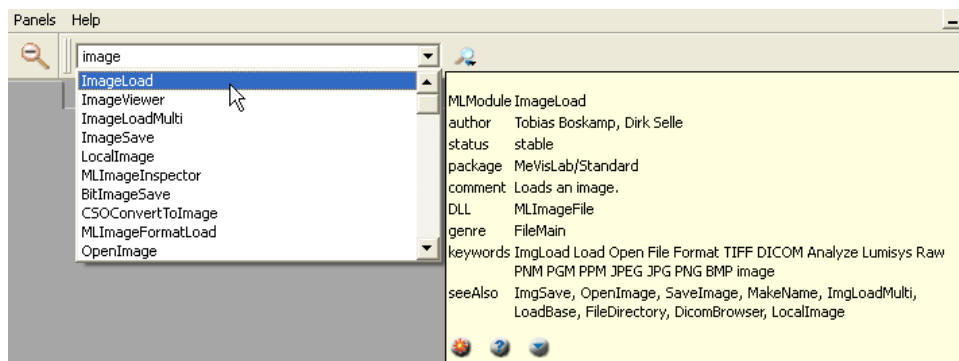


### Tip

The quick search field does not need to have the focus — any time you enter something in the MeVisLab GUI while not being in a dialog window, this will be entered into the quick search automatically.

**Figure 3.4. Quick Search Options**

To search for a module to load an image, you could either type “load” or “image”. Let us go with the second option this time. While typing “image”, the possible results appear. Use the up/down keys on your keyboard to move to one of the listed modules. The module's About information will appear next to it, allowing you to decide if this is the right module for you.

**Figure 3.5. Quick Search Results****Tip**

For a more complex search, use the **Module Search** View.

Select `ImageLoad` and press **ENTER** to add the module to a new network.

**Figure 3.6. ImageLoad Module**

The module is an ML module, as can be seen by the blue color. It offers one image output connector (triangle for image, output because it is on the top of the module; see [Chapter 2, The Nuts and Bolts of MeVisLab](#)).

In the next section, we will have a closer look at the module details.

## 3.3. Using the ImageLoad Module

For the following section, we expect that the Views **Output Inspector** and **Module Inspector** are open. If necessary, add them via **View** → **Views**.

1. First, we need to load an image.

- a. Double-click the `ImageLoad` module to open its panel.
- b. Click **Browse** to select a file for display. The default file browser opens.
- c. Go to the MeVisLab DemoData directory at `$(InstallDir)Packages/MeVisLab/Resources/DemoData` in the MeVisLab installation path and select a file, for example a head shot (`Head4_t1_small.tif`). The image is loaded immediately. (Instead of `ImageLoad`, you could also use `LocalImage` which is optimized for loading images in relative paths, as explained in [Section 3.5.3, "Using the LocalImage Module"](#)).



### Tip

If you would like to start with your own image data immediately, please see the chapter [Section 3.6, "A Note on Importing DICOM Images"](#) on how to convert your DICOM slices into the internal file format of MeVisLab first. Then continue in place.

Module panels are intended to stay open, so keep the panel open or minimize it if it gets in your way. There are two ways to minimize a panel:

- Click the minimize button on the top right of the panel window: this will minimize only this panel.
- Select **Panels** → **Minimize All Open Panels** (or press the respective keyboard shortcuts): this will minimize all panels of this network.



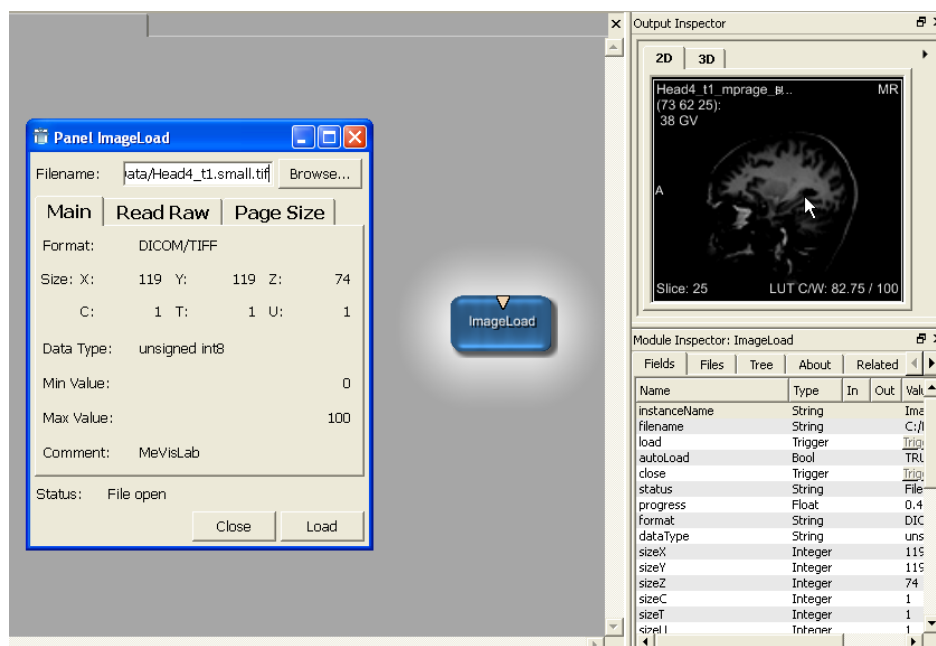
### Note

Do not use the **Close** button on the `ImageLoad` panel as this will close (and unload) the image.

2. For display, you can either add a viewer (we will do this later in this example) or you can click the module's output connector to display the image in the **Output Inspector**.

The great thing about the **Output Inspector** is that it will display the output of any connector in the process chain (as long it is a format the inspector can interpret). So if you are ever unsure about what is actually the input or output of a module, simply click the connector to find out.

**Figure 3.7. ImageLoad Panel and Output Inspector**



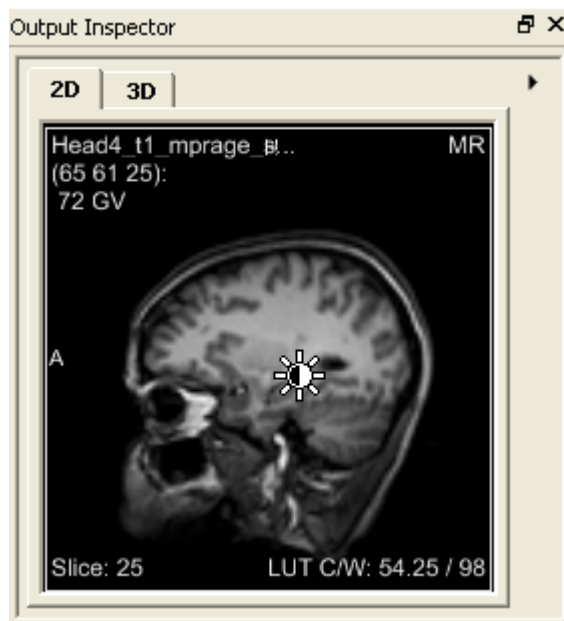



Your image does not look like this? One reason might be that the slice of the image you are looking at has no information. Click on the **Output Inspector** and scroll through the slices by

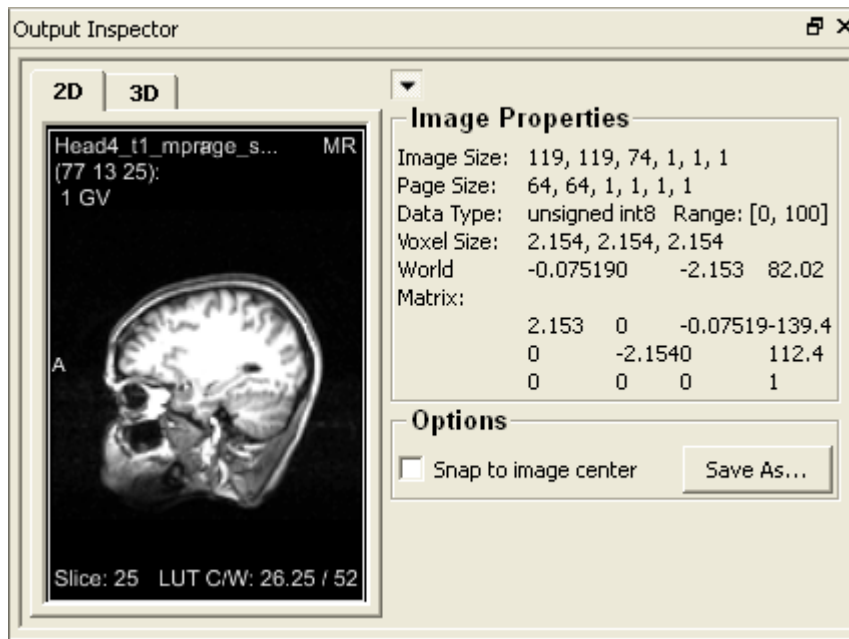
- using the mouse wheel
- keeping the middle mouse button (mouse wheel) pressed and moving the mouse up and down
- pressing the arrow keys

Still not seeing anything? Then try to adjust the visibility range by changing the windowing. For this, keep the right mouse button pressed while moving the mouse up/down (for window width) or left/right (for window center). During these actions, the mouse cursor changes into a contrast symbol.

**Figure 3.8. Adjusting the Windowing**



Both on the panel and on the additional information of the **Output Inspector**, the image properties can be found. In the **Output Inspector**, you can open them by clicking .

**Figure 3.9. Output Inspector with Image Properties**

The image properties show the following information (see [Chapter 10, Excursion: Image Processing in ML](#) for more information):

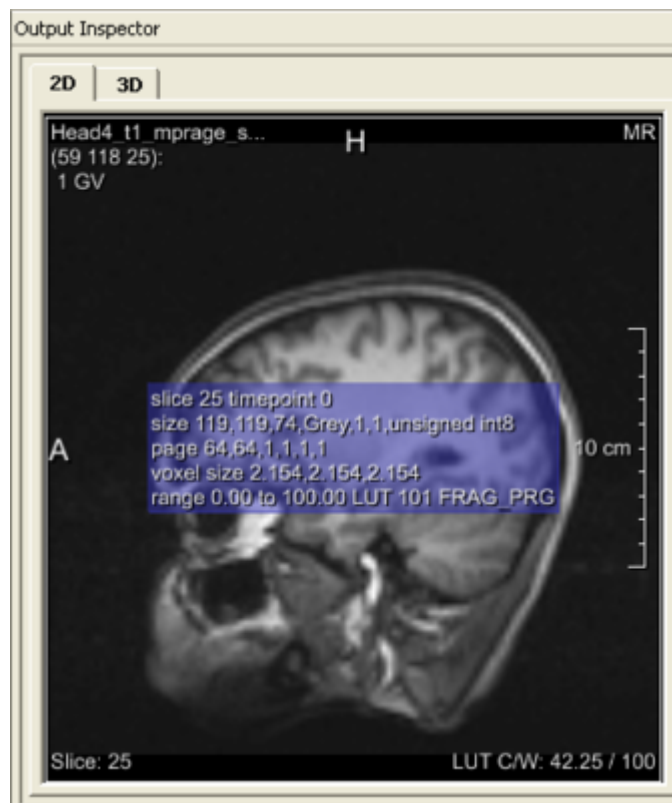
- Image Size in x, y, z, c, t, n
- Page size in x, y, z, c, t, n
- Data type and range
- Voxel size in mm
- World matrix

Two options are available:

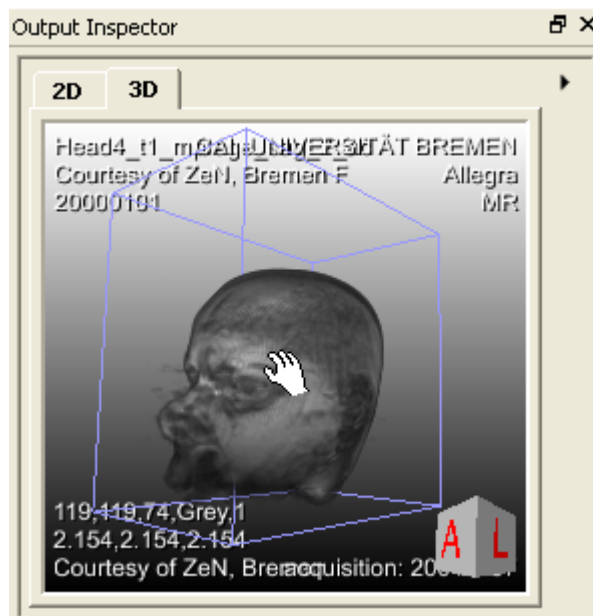
- **Snap to image center:** If selected, the image is centered, that is the middle z slice is shown (only effective when opening a new display).
- **Save as:** Opens a Save dialog.

In addition, two key shortcuts are available:

- **A:** Toggle the display of the annotations.
- **I:** Toggle the display of an additional information display.

**Figure 3.10. Output Inspector with Additional Information Display**

A 3D display is possible (in case of a single slice, its depth is the voxel depth). For this, click the 3D tab in the **Output Inspector**.

**Figure 3.11. 3D Output Inspector****Note**

The 2D and 3D views are independent of each other.

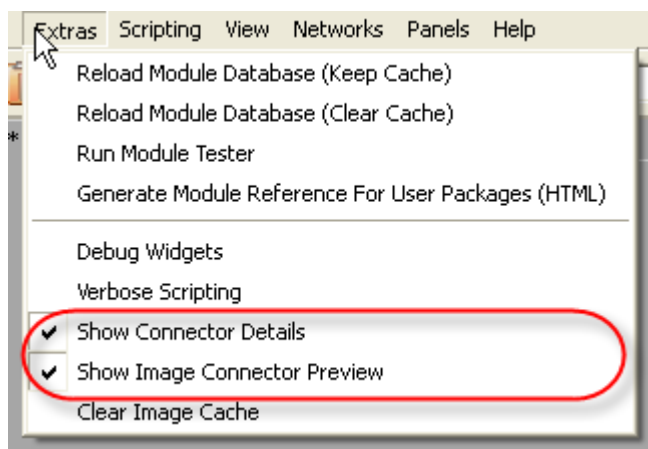
The 3D display can be rotated. The orientation can be seen on the little cube in the lower right corner of the viewer (Notation: A = anterior, front; P = posterior, back; R = right side; L = left side; H = head; F = feet). You can also use the windowing described above for the 2D view.

The information given in the panel and the 2D view image properties of the **Output Inspector** can also be displayed right next to the module connector. For this, check

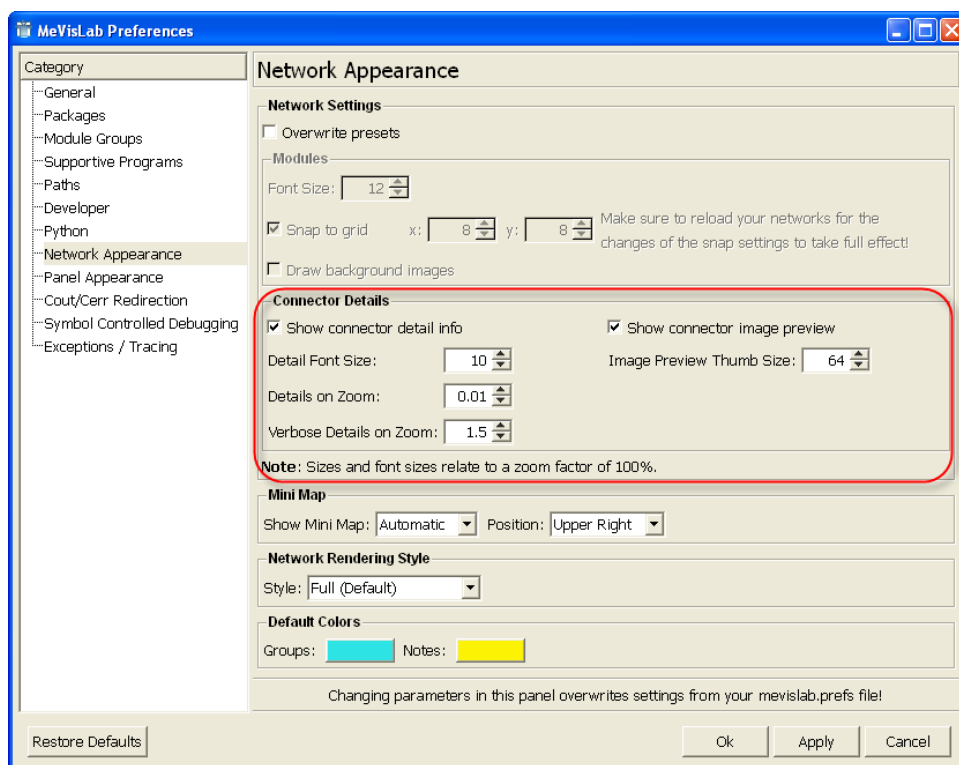
- **Extras** → **Show Image Connector Preview** for a thumbnail preview and/or
- **Extras** → **Show Connector Details** for connector details.

Alternatively, activate the respective options in the Preferences, section “Network Appearance” (on Windows and Linux: **Edit** → **Preferences**; on Mac OS X: **MeVisLab** → **Preferences**).

**Figure 3.12. Connector Details in the Edit Menu**



**Figure 3.13. Connector Details in the Preferences**



The additional information is displayed when single-selecting a module. The amount of displayed information depends on the zoom factor. To zoom in/out of a network, scroll with the mouse wheel.

**Figure 3.14. Connector Details Depending on Zoom**



For this example, we will work without the connector details display, because it tends to clutter the interface.

## 3.4. Adding Viewers to ImageLoad

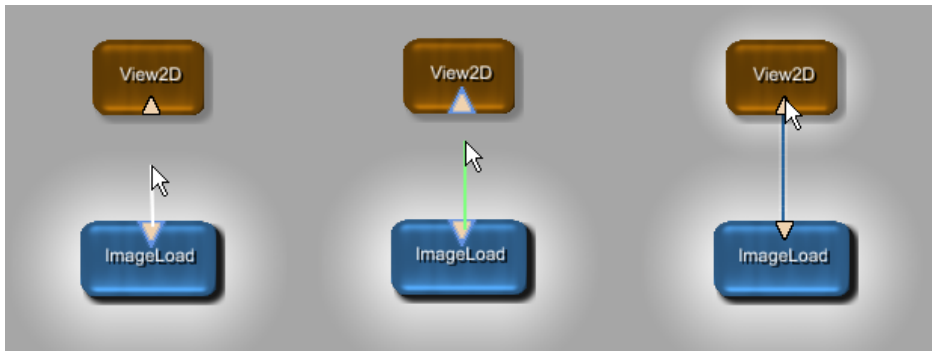
Instead of using the *Output Inspector* (whose display might change with every clicked connector), it is sensible to add a viewer to the network. There are two standard macro modules available in MeVisLab which provide standard viewer configurations for 2D and 3D rendering, namely *View2D* and *View3D*. Especially the 2D Viewer is frequently used to examine image processing results within a module pipeline, for example. Once you begin to implement your own applications, you are free to create your own viewer implementations adapted to your special tasks.

### 3.4.1. Adding the View2D Module

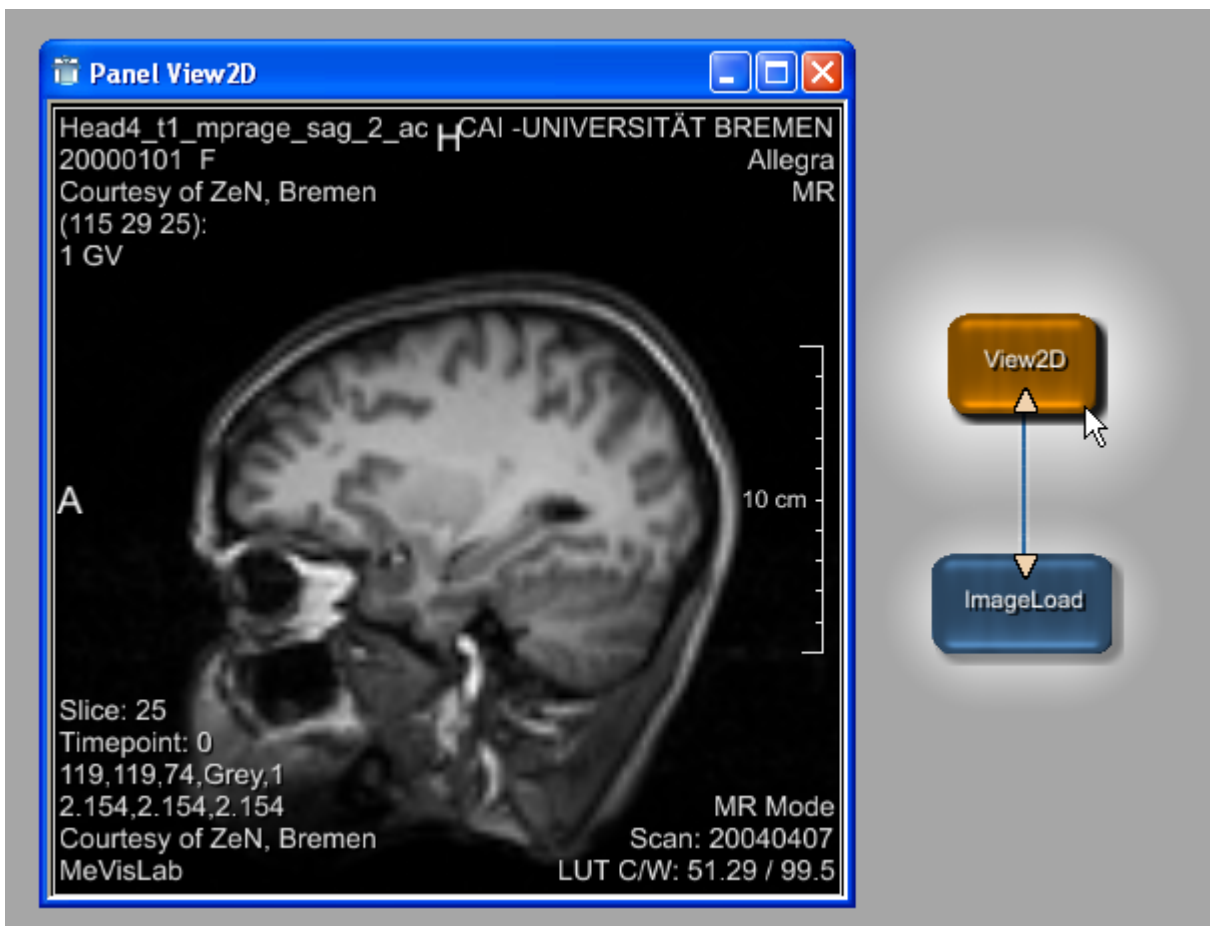
1. Add a *View2D* module to your network. In the **Modules** menu it is located at **Modules** → **Visualization** → **2D Viewers** → **View2D**.

The *View2D* module has one input connector for the image to be rendered. (It also has three Inventor inputs which are hidden by default, see [Chapter 5, Defining a Region of Interest \(ROI\)](#).)

2. Feed in the image by connecting the image output of the *ImageLoad* module with the image input of the *View2D* module. This is done as follows:
  - a. Click the output connector of *ImageLoad*.
  - b. Keep the left mouse button pressed while dragging the connection to the input connector of *View2D* (white line).
  - c. Check that the connection is well-defined (green line).
  - d. At the input connector of *View2D*, release the mouse button and establish the connection (blue line).

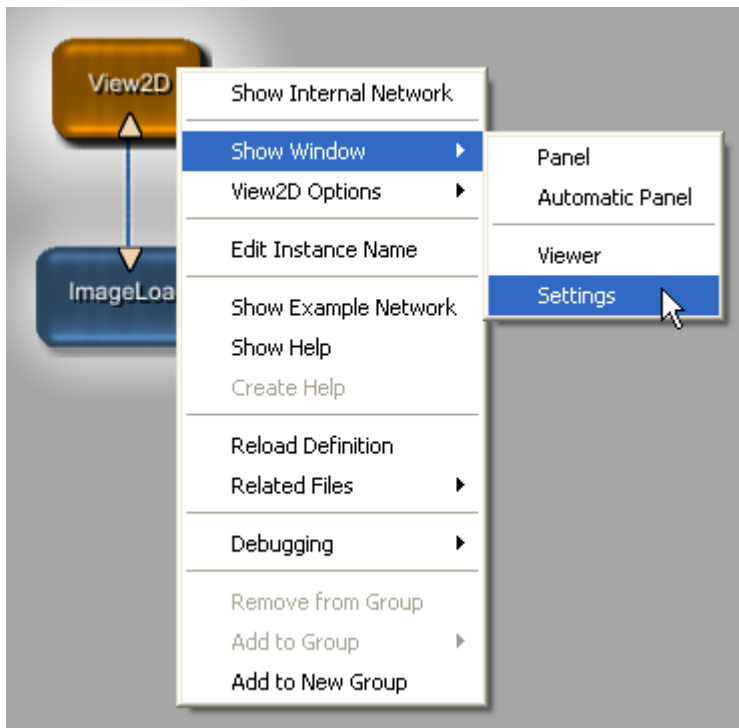
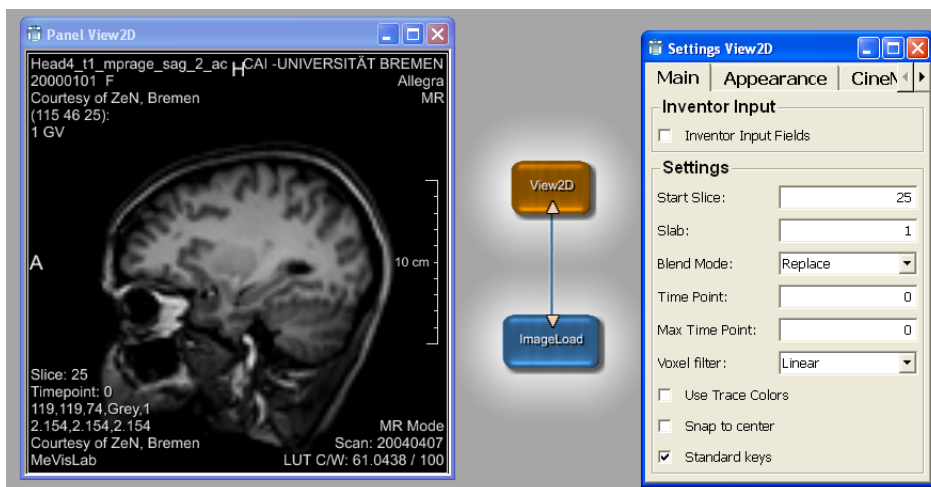
**Figure 3.15. Setting up the Connection**

Although the connection is established, no image rendering has started yet. To initialize rendering, open the `View2D` panel by double-clicking the `View2D` module in your network. As you can see, the default panel is the viewer itself.

**Figure 3.16. Panel of View2D**

The `View2D` panel provides a standard viewer with many features, like slicing, zooming, windowing, annotations, slab view, cine mode, and many more. A full description of all supported features and how to use them can be found on the `View2D` help page which you can open from the module's context menu.

The `View2D` module offers various settings. As the default panel is the viewer, the Settings panel needs to be opened explicitly from the context menu via **Show Window** → **Settings**.

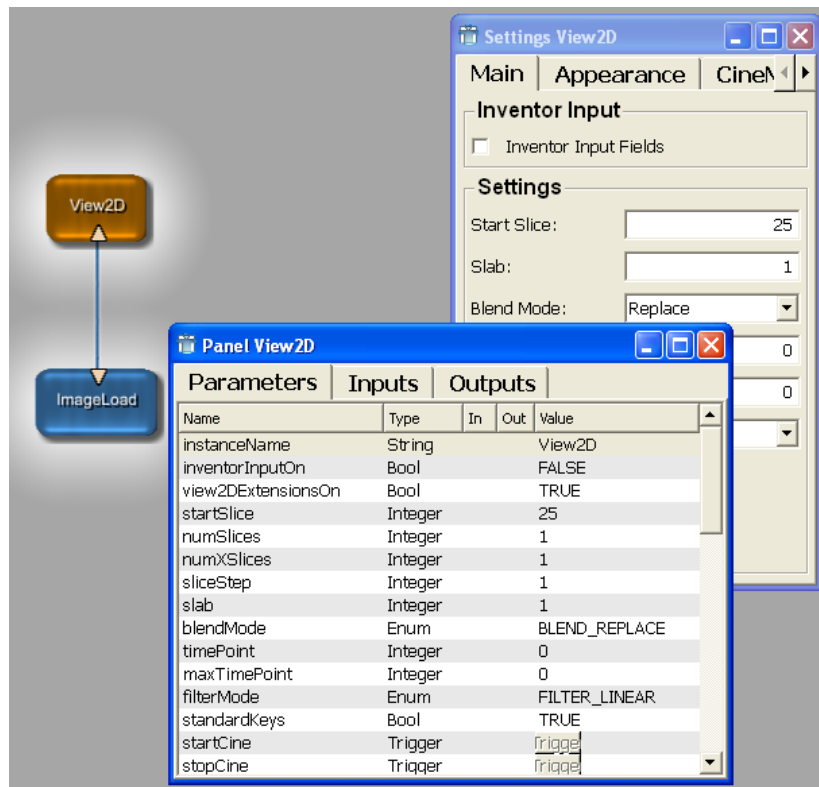
**Figure 3.17. Opening the Settings Panel of View2D****Figure 3.18. Settings Panel of View2D**

As you can see on the Settings panel, the View2D module also offers Inventor inputs that are usually hidden. Take a look at the module's example network (context menu, **Show Example Network**) for the usage of these Inventor inputs connectors. Another module that might get connected here is the View2DExtension macro module, which extends the viewer for drawing (image overlays, contours, ROIs), measuring and more.



## Note

A module always has one automatic panel and may have an arbitrary number of additional panel windows, as defined in an MDL file (in the .script file by default). The automatic panel lists all variables, fields and inputs/outputs of the module; the scripted panels may only include a fraction of these fields (see also [Section 2.6, "User Interfaces Controls"](#)).

**Figure 3.19. Automatic and Settings Panel of View2D**

3. Now is a good time to save your network as `MyFirstNetwork.mlab`. You can do this in several ways:

- Select **File** → **Save** or press the respective keyboard shortcut (for a list for all operating systems, see the MeVisLab Reference Manual, chapter “Shortcuts”).
- Click the disk symbol in the toolbar.

The network modules and all module parameters are stored. Next time you open the network, you will get access to the loaded image at the output of the `ImageLoad` module immediately.



### Tip

You can quickly re-open the last twenty networks via the menu bar, **File** → **Recent Files**.



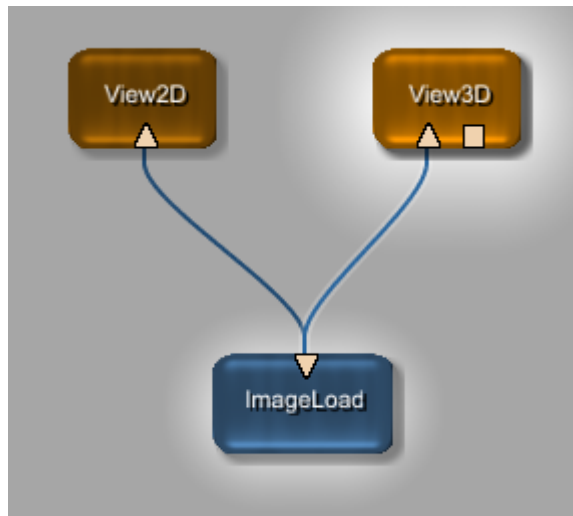
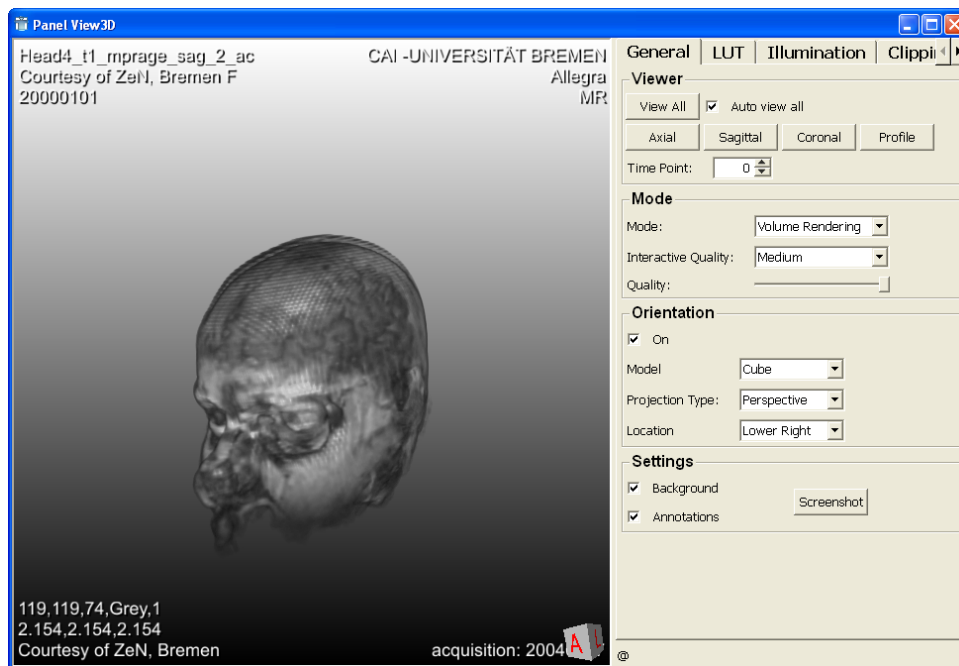
### Tip

If the option **Auto save MeVisLab documents** in the Preferences is selected, MeVisLab networks are auto-saved as `<NetworkName>.mlab.auto` upon major changes. This allows for restoring in case of system crashes. Auto-saved copies are deleted when the according networks are saved.

## 3.4.2. Adding the View3D Module

The `View3D` macro module is an easy-to-use application of the `SoGVRVolumeRenderer` module, which is a high-end, hardware-based image rendering module using 3D textures. Adding the `View3D` module to the network, we get access to a 3D scene of our example image.



**Figure 3.20. Connecting the View3D Module****Figure 3.21. The View3D Panel**

In addition to the 3D display offered by the **Output Inspector**, the View3D viewer comes with several panels on which you can set display details or even record a movie.

## 3.5. Alternative Ways to Load Images

Besides the way described above, there are variations.

### 3.5.1. Dragging Images onto the Workspace

Instead of adding the module, you can drag the image file

- onto the workspace: An `ImageLoad` module is created automatically in the current network when you drag a DICOM or TIFF image file from a file browser onto the MeVisLab workspace. The dragged file is loaded automatically and available at the image output connector of the created `ImageLoad` module.



### Tip

This mechanism also works for WEM files (creates a WEMLoad module) and CSO files (creates a CSOLoad module). For these module classes, see [Chapter 13, Developing Inventor, WEM and CSO Modules](#).

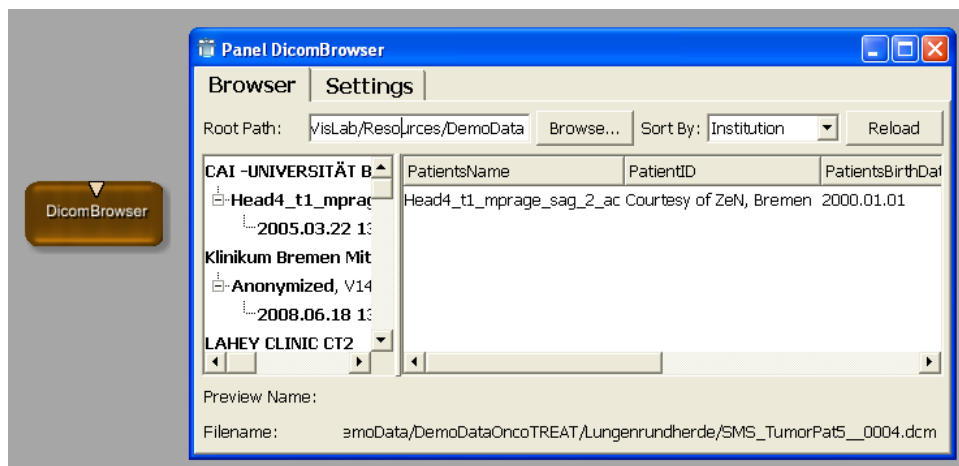
- onto an existing ImageLoad module
- onto the filename field of an existing ImageLoad module

## 3.5.2. Adding Images via the DICOM Browser

For loading DICOM files (or DCM/TIFF pairs, see [Section 10.7, “Data Types for DICOM and TIFF”](#)), you can use the `DicomBrowser` module.

With the `DicomBrowser`, DICOM images can be sorted by DICOM tags like institution, patient, modality etc. The default browser path is set to the MeVisLab image path at `$(InstallDir)/data`. You can set your own default `DicomBrowser` path in the Preferences, section “Paths”.

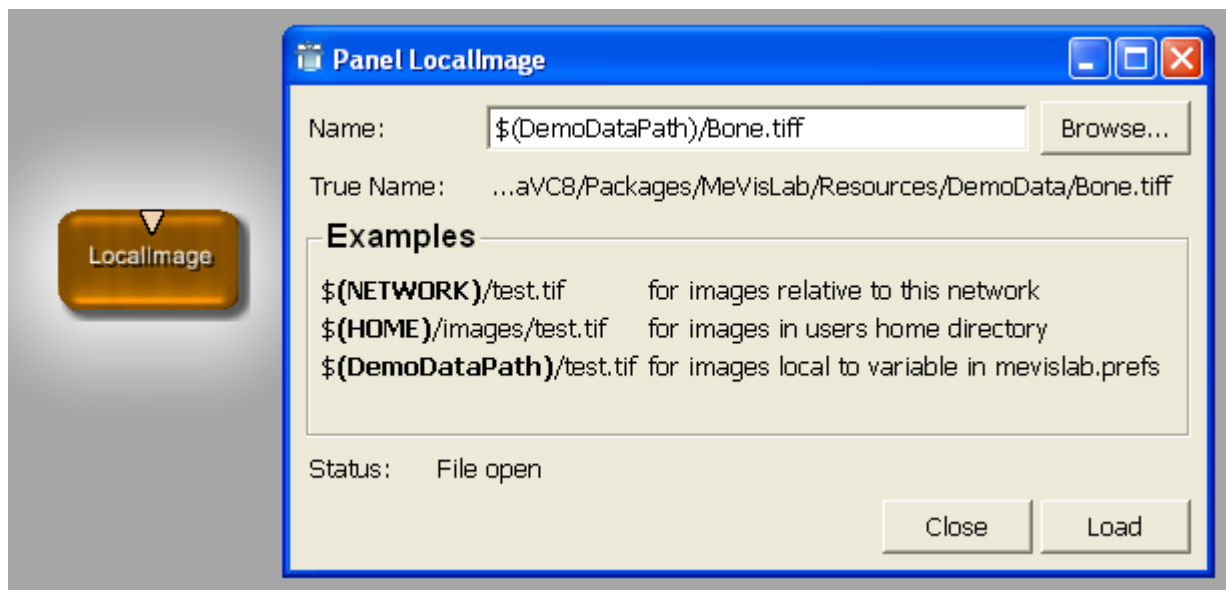
**Figure 3.22. DICOM Browser**



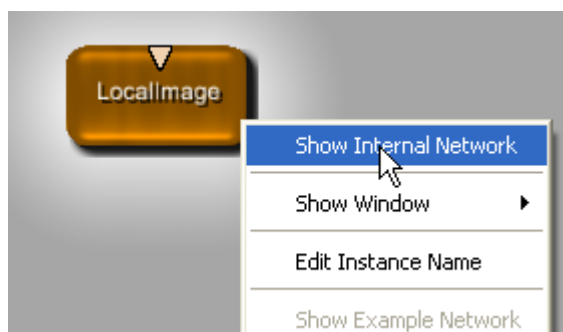
## 3.5.3. Using the LocalImage Module

Instead of using the `ImageLoad` module, you can use `LocalImage`.

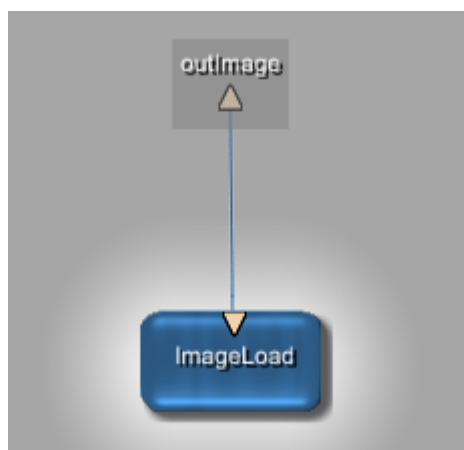
`LocalImage` is a macro module that allows for image selection based on relative paths. This method is recommended for image referencing because it enables an easier exchange of networks between cooperating parties. On the panel, the list of supported variables and their meaning is displayed.

**Figure 3.23. LocalImage Module**

Macro modules are a combination of an internal network and a script. You can open the internal network via the module's context menu or by pressing **SHIFT** and double-clicking the module.

**Figure 3.24. Show the Internal Network**

In the case of `LocalImage`, the internal network consists of an `ImageLoad` only. The difference to that module is only in the scripting that offers relative instead of absolute paths to the file.

**Figure 3.25. Internal Network of the LocalImage Module**

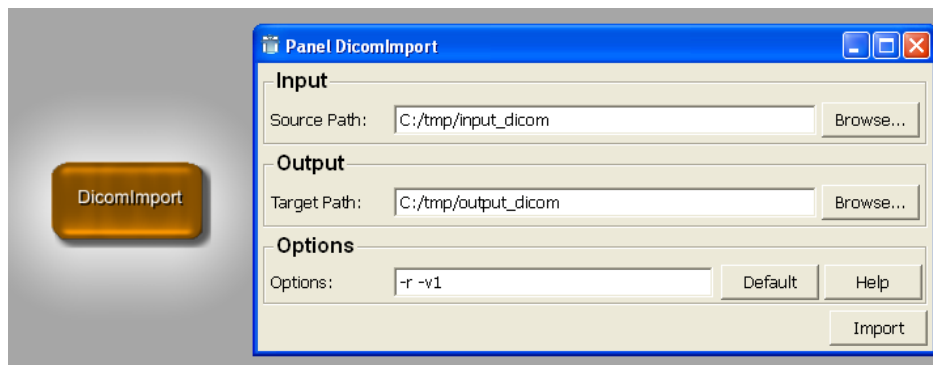
## 3.6. A Note on Importing DICOM Images

MeVisLab works with its own 3D file format which stores the image values and the image DICOM tags separately in two files with same name but different extensions: `<filename>.tiff` and `<filename>.dcm`. Without importing your DICOM slices to MeVisLab DICOM/TIFF format, the MeVisLab image loading modules will only be able to load single DICOM slices separately. For further information, see the chapter [Chapter 10, Excursion: Image Processing in ML](#).

The DICOM import is provided by the module `DicomImport`.

1. Add the module to the network via the quick search or the menu bar, **Modules** → **File** → **DICOM** → **DicomImport**. Open the module panel with double-click on the module.

**Figure 3.26. DicomImport**



2. Enter the necessary data.
  - a. Select the `Source Path` where your DICOM slices are located. In the MeVisLab installation path you can find some example DICOM slices in the `$(InstallDir)/MeVisLab/data/demodata/BrainT1Dicom` directory. All subdirectories will be scanned recursively and each series will be converted into the 3D DICOM/TIFF format.
  - b. Select the `Target Path` where your imported DICOM/TIFF files will be stored in. If you want to import the example DICOM slices, we suggest using the `$(InstallDir)/MeVisLab/data/demodata` path.
  - c. Click the **Import** button. A window pops up showing the import progress. Close the window when the import has been finished (`Successfully terminated`).

Depending on how much series have been imported, you will find one or more DICOM/TIFF file pairs in the `Target Path`. For the example slices import, two files should have been created: `TestPatien_id0__0001.dcm` and `TestPatien_id0__0001.tiff`.

If your DICOM import fails, check if some optional flags in the `Options` field are missing. You can find more information either via the options description (**Help** button in the module panel) or via the module's help page (context menu, **Show Help**).



### Tip

DICOM multi-frame files can be opened directly in MeVisLab; therefore, the import step is not necessary for displaying the data. (For image processing, it is still recommended to import the files.)

# Chapter 4. Implementing a Contour Filter

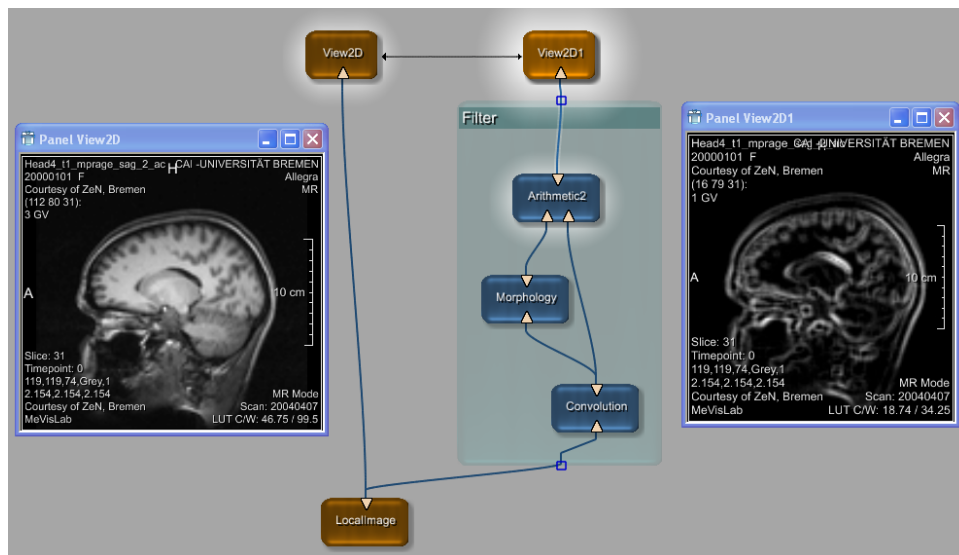
In this chapter we will introduce to you how an image processing pipeline is implemented by means of a MeVisLab network. We are going to implement a contour filter which is based on the elementary image processing steps average, dilation and subtraction. To get a visual impression of what the filter is doing, we will also implement two synchronized render pipelines with 2D viewers for the filter in- and output.

Following this chapter you will get an idea about how to

- implement an image processing pipeline (see [Section 4.2, “Implementing the Contour Filter”](#)).
- synchronize parameters between different modules by establishing parameter connections (see [Section 4.3, “Parameter Connection for Synchronization”](#)).

This will be our resulting network:

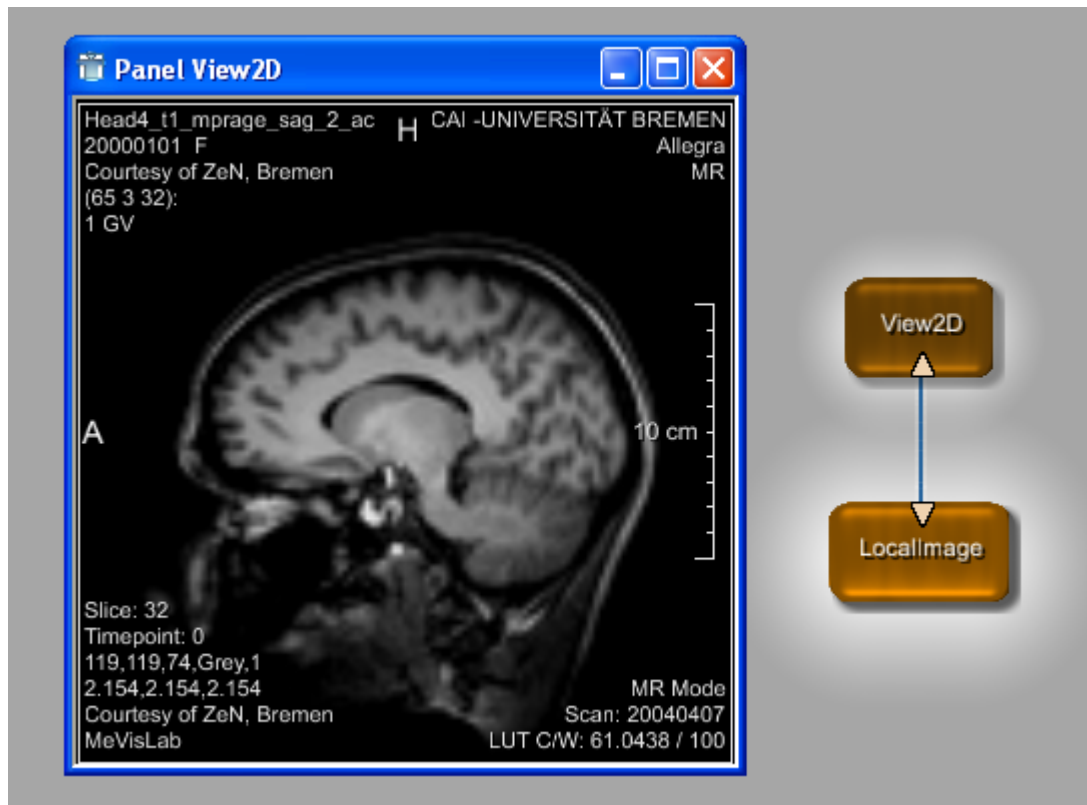
**Figure 4.1. Example Network Contour Filter**



## 4.1. Loading the Input Image

First, we need an image as input. This image will be used as the input image for the normal viewer as well as as the input and filter image for the image processing pipeline.

1. Create a new network (**File** → **New**) and save it to disk.
2. Find and add the `LocalImage` module via the Quick Search. As image input, use an image from the default MeVisLab demo data path.
3. Choose an image filename by opening the module's panel and set the module parameter `Name` to the value `$(DemoDataPath)/Head4_t1_small.dcm` or to any other image name located in the `$(DemoDataPath)` directory.
4. For the output, find and add the `View2D` module via the Quick Search and connect it to the `LocalImage` output. Double-click `View2D` to see the original image. Later, we will compare this output with the image resulting from the filter process.

**Figure 4.2. Viewing the Input Image for the Contour Filter****Tip**

To see an immediate (albeit small) preview of the input image, you can enable the preview modus in the menu bar, **Extras** → **Show Image Connector Preview**.

## 4.2. Implementing the Contour Filter

We want to implement a contour filter that is composed of the following image processing pipeline:

1. Take an input image *a*.
2. Smooth the input image with an average kernel: `Average[image a] -> image b`.
3. Dilate the smoothed image by means of a morphological kernel operation: `Dilate[image b] -> image c`.
4. Subtract the smoothed image from the dilated and smoothed image: `Subtract[image c, image b] -> image d`.
5. Output the filter output image *d*.

For this processing pipeline we need the following basic image operators:

- Average operator: a search yields the module `Convolution`. From the description: "Simple constant convolution filters like Average, Gauss, Sobel, Laplace."
- Dilation operator: a search yields the module `Morphology`. From the description: "Implements dilation and erosion filters that separately act on single bits."
- Subtraction operator: a search yields various arithmetic modules. How to decide which module is the correct one? When you add the modules and have a look at the modules' help, you will find that `Arithmetic0` is for arithmetic operations on scalars or 3D vectors, `Arithmetic1` is for arithmetic operations on a single image, and `Arithmetic2` is for arithmetic operations on two images. As we want to subtract two images, `Arithmetic2` is the correct module.

Proceed as follows:

1. Add the modules `Convolution`, `Morphology`, and `Arithmetic2` to the network.

Alternatively you could find and add the modules to the network via the **Modules** menu:

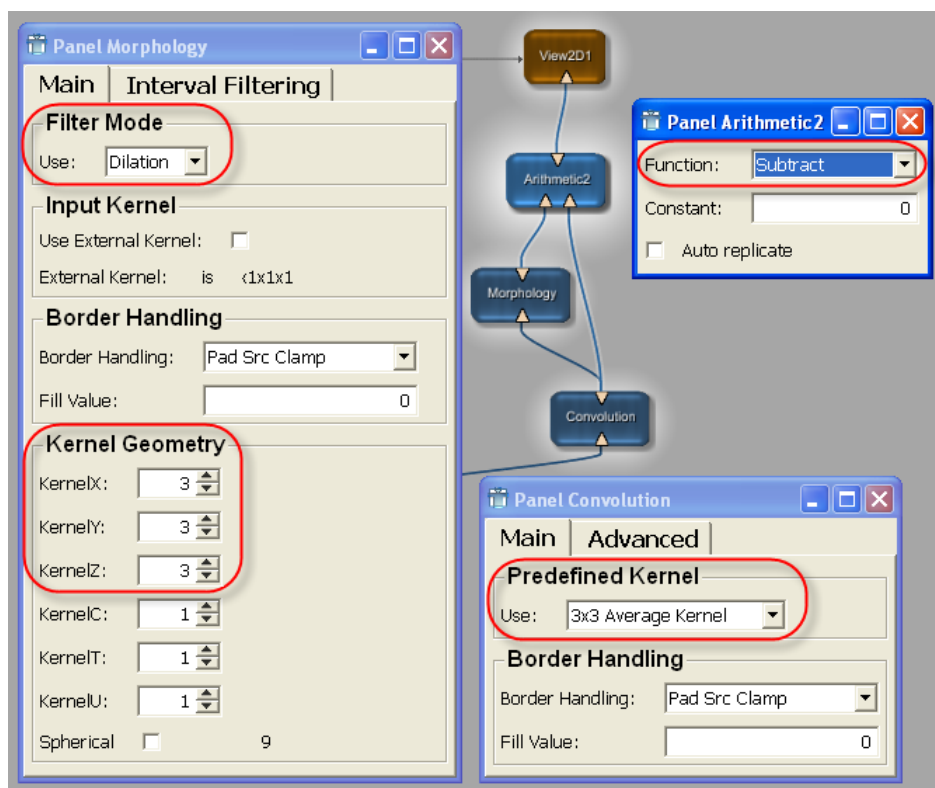
- a. via **Modules** → **Filters** → **Kernel** → **Convolution**,
- b. via **Modules** → **Filters** → **Morphology** → **Morphology** and
- c. via **Modules** → **Analysis** → **Arithmetic** → **Binary** → **Arithmetic2**.

The image we use as input has to be processed first via the `Convolution` module. After that, the resulting convoluted image will be processed and also output directly to the `Arithmetic2` module where the two images will be subtracted.

For the subtraction, the following information is offered in the help of `Arithmetic2`: "The input image 1 decreased by input image 2 is passed to the output." Therefore, it is important to connect the images in the correct order, otherwise the result will look rather strange.

2. Open the panels of `Convolution`, `Morphology` and `Arithmetic2` by double-clicking the modules. Then adjust/check the default values of the following parameters:
  - a. Module `Convolution`: Keep the default kernel type "3x3 Average Kernel" for `predefKernel`.
  - b. Module `Morphology`:
    - i. In the field `Filter Mode`, keep the default value "Dilation".
    - ii. For the `Kernel Geometry`, use a kernel of the size 3x3x3.
  - c. Module `Arithmetic2`: In the field `Function`, change the default value "Add" to the value "Subtract".

**Figure 4.3. Adjust Filter Parameters**



### Tip

You can view and edit module field values also in the **Module Inspector** View. On the **Fields** tab, all fields of the currently selected module are listed by names and values.

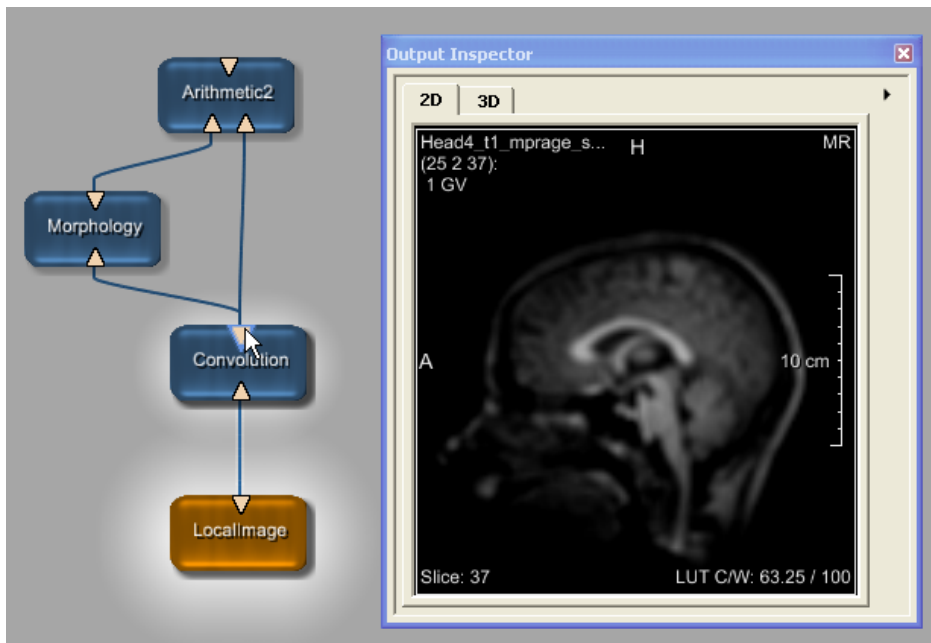


## Note

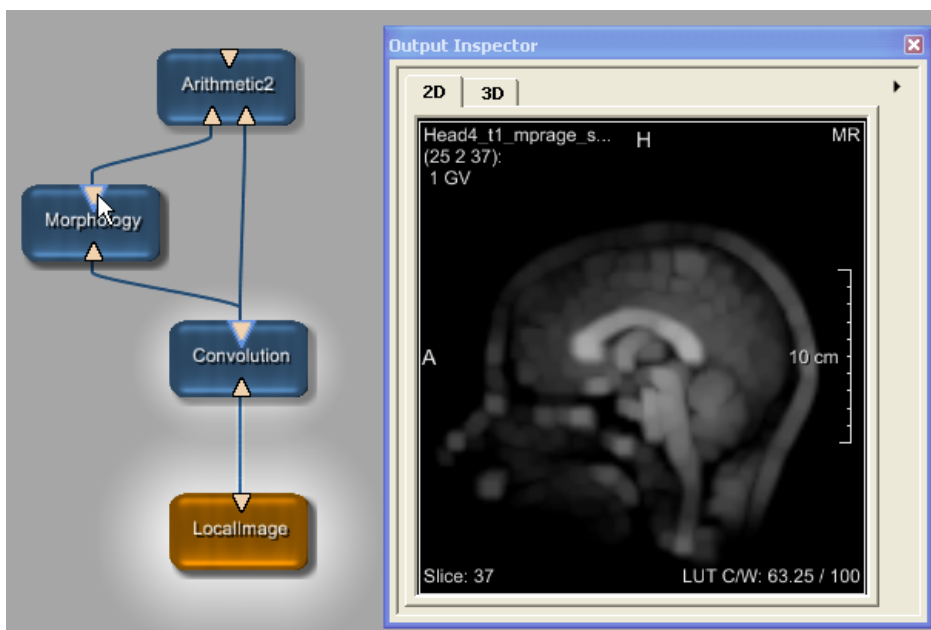
Field names (in the module) and field labels (in the interface of the module panel) do not have to be the same. To find the field name, right-click the field label on the panel; the field name is listed as first entry of the context menu.

- To view the results of every step in the processing pipeline, use the **Output Inspector**, which can be opened via the menu bar, **View** → **Views**. Click each connector to follow the image processing.

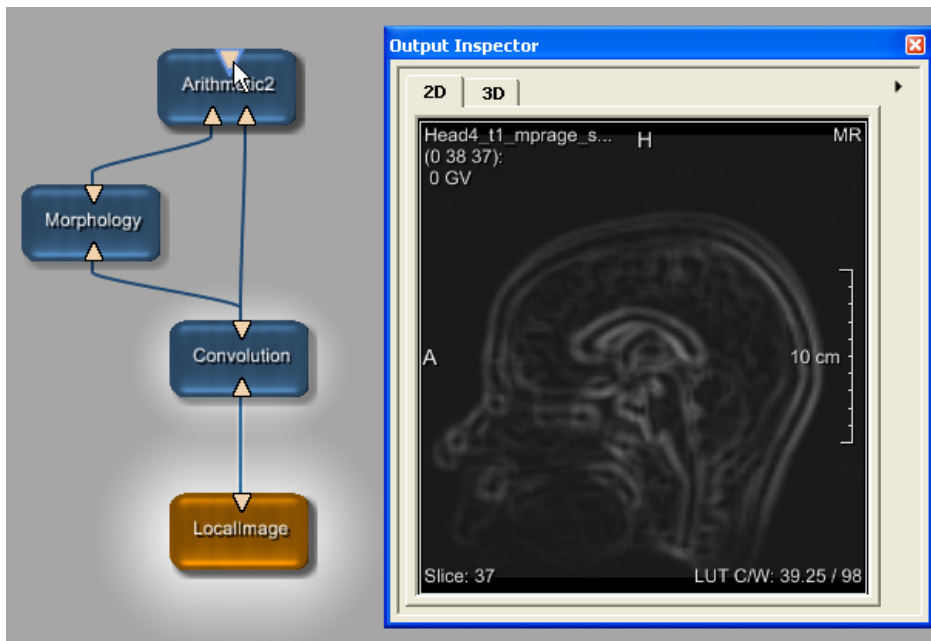
**Figure 4.4. Constructing the Filter Pipeline — Convolution Output**



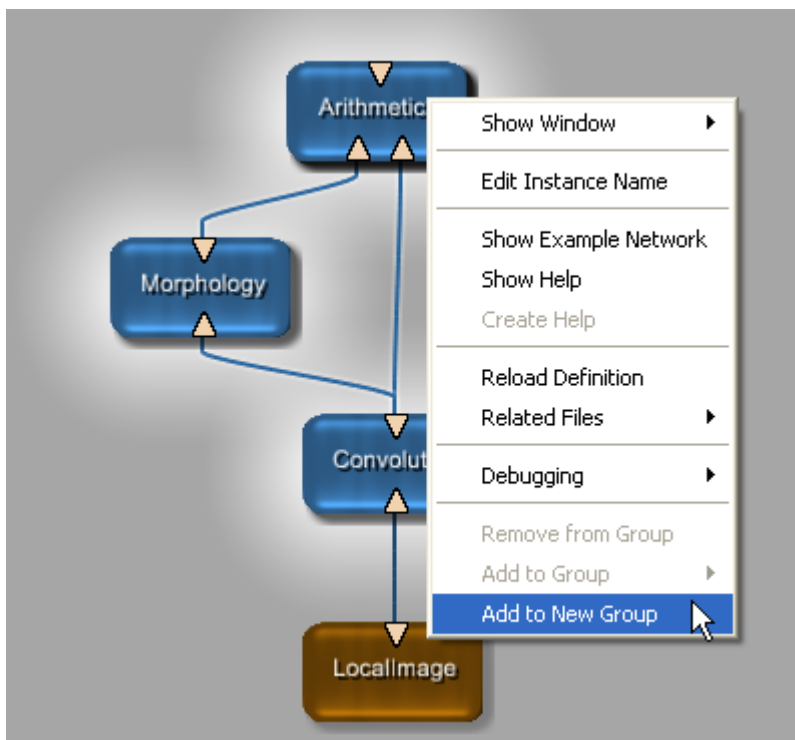
**Figure 4.5. Constructing the Filter Pipeline — Morphology Output**





**Figure 4.6. Constructing the Filter Pipeline — Arithmetic2 Output**

4. To distinguish the image processing pipeline, you can create a group for it. For that:
  - a. Select the three modules, for example by dragging a selection rectangle around them, or by single-selecting the modules while pressing **SHIFT**.
  - b. Right-click the selection to open the context menu and select **Add to New Group**.
  - c. Enter a name for the new group, for example "Filter".

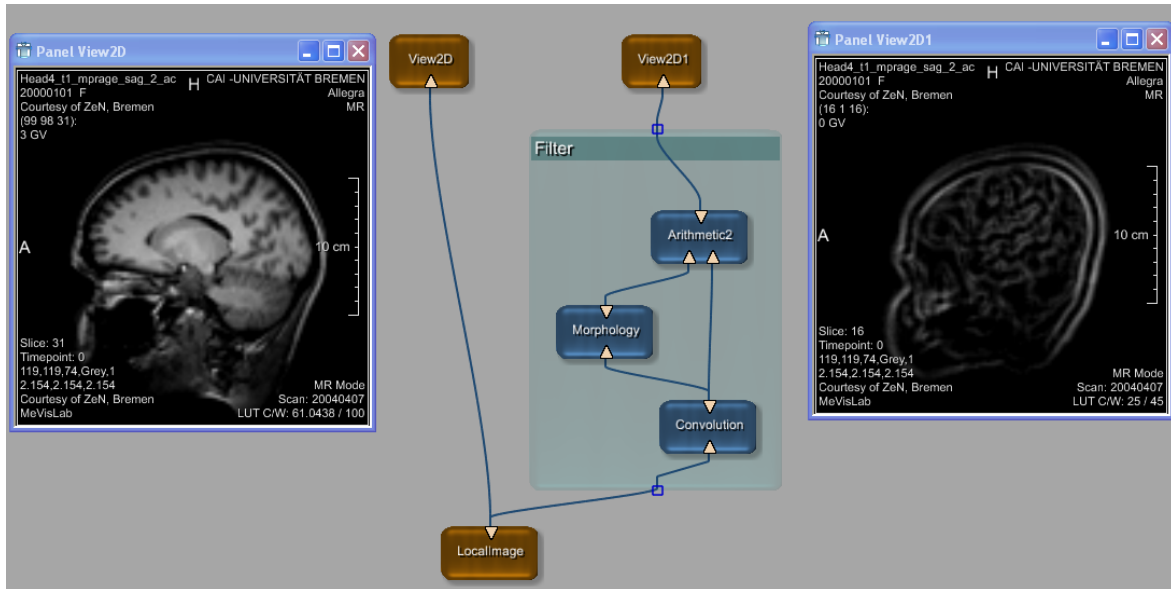
**Figure 4.7. Creating a New Group**

The new group is created and displayed as a green rectangle. The group allows for quick interaction; for example, a double-click on its title bar zooms in and centers the group; a right-click on the title

bar opens a menu for editing and deleting the group. You can also change the default color in the Preferences. For further information on groups, please refer to the MeVisLab Reference Manual.

5. For the output, add another `View2D` module, either via the quick search or by selecting the existing `View2D` module in the network and duplicating it (via **Edit** → **Duplicate** or by pressing the keyboard shortcuts given there).

**Figure 4.8. Resulting Contour Filter Network**



### Tip

The filter can be tuned via some parameters given in the `Convolution` and `Morphology` modules. Changing the convolution kernel size (field `predefKernel` of the `Convolution` module) and/or the dilation kernel (fields `kernelX`, `kernelY`, `kernelZ` of the `Morphology` module) will enhance contours at different scales.

In a final step, we will synchronize the Viewers of the two `View2D` modules by establishing parameter connections between them.

## 4.3. Parameter Connection for Synchronization

Besides data connections between module inputs and outputs (Image, Inventor and Base connectors) there is also the possibility to connect module fields via a parameter connection. The values of connected fields are synchronized, that means when changing the value of one field, all fields connected to this field will be adapted to the same value.

Some important points:

- Fields can be connected to an arbitrary number of other fields as source, but only once as destination. (Similar to data connections, for which an output connector can be connected to an arbitrary number of other connectors but an input connector can only be connected once.)
- Connections between fields may be unidirectional or bidirectional.

Unidirectional: Field A is the output and field B the input. Changes in field A reflect in field B but changes in field B have no effect on field A.

Bidirectional: Field A is the output and field B the input and vice versa (two parameter connections). Changes in field A reflect in field B and changes in field B reflect in field A. (This is the setting we will use in our example.)



### Tip

MeVisLab prevents the creation of infinite loops.

- Not all connections between all fields are sensible. Usually the connected fields should be of the same type.
- Parameter connections may be established both between fields within the same module and between fields of different modules.
- On the MeVisLab user interface, parameter connections are established by dragging fields onto the labels of automatic panels (and most scripted MDL panels, see the MeVisLab Reference Manual, chapter “Parameter Connections Inspector” for details).

In our example, a bidirectional parameter connection is the way to synchronize the `View2D` modules so that the same slice is rendered in both viewers. To establish this, proceed as follows:

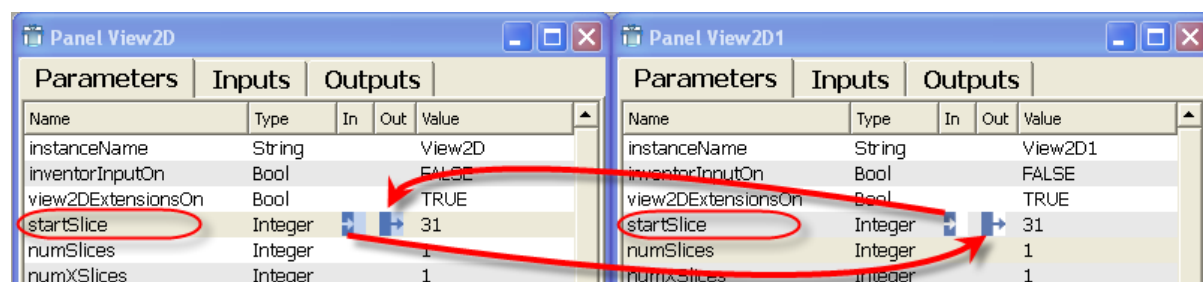
1. Right-click each `View2D` module to open the context menu and select **Show Window** → **Automatic Panel** (alternatively, press **ALT** and double-click the module). The field that controls the currently rendered slice in a `SoView2D` module is the `startSlice` field.
2. On the `SoView2D` panel, select the label of the `startSlice` field and drag the (invisible) connection onto the label of `startSlice` field on the `SoView2D1` panel. The connection is drawn as thin grey arrow with the arrowhead pointing to the module that receives the parameter as input.
3. Repeat the process in the other direction by dragging the `startSlice` field from the `SoView2D1` panel to the `SoView2D` panel. The bidirectional connection is drawn as a thin, grey double arrow.

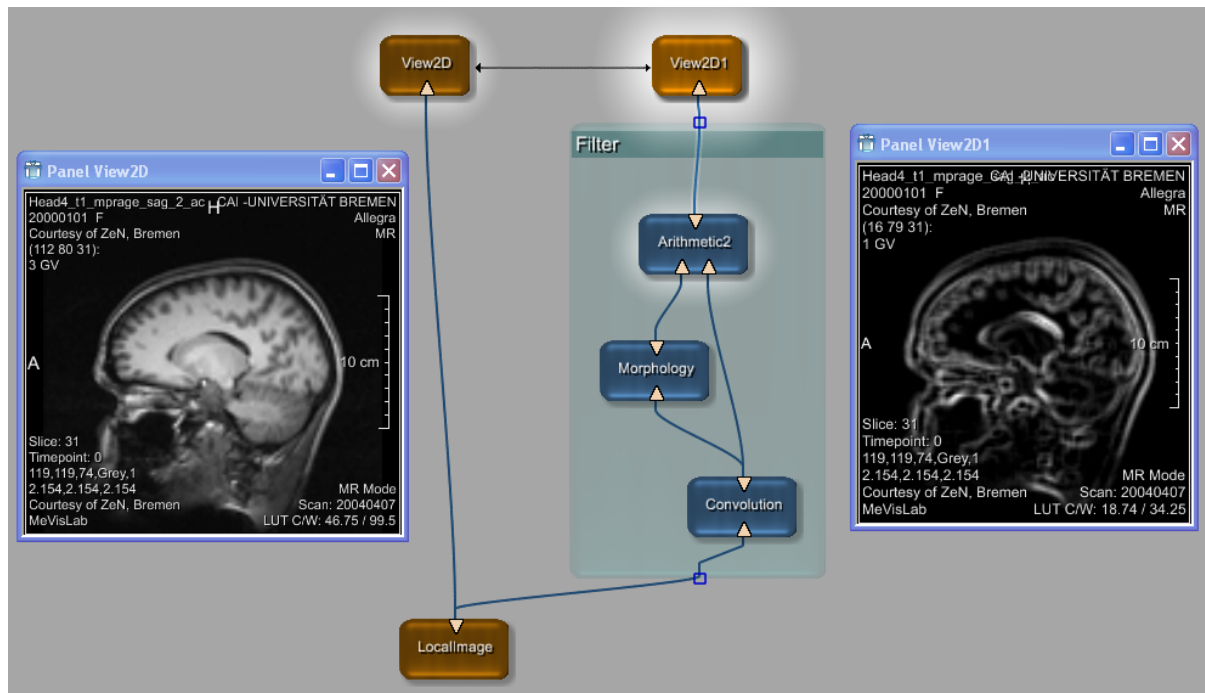


### Tip

Another typical way of notating the fields is “InstanceName.FieldName”, for example `SoView2D.startSlice`. You will find this notation when you right-click the parameter connection to open its context menu, in which you can disconnect single or all parameter connections.

**Figure 4.9. Establishing the Parameter Connections**



**Figure 4.10. Resulting Network**

As a result, moving through the slices with the mouse wheel (“slicing”) in one of the viewers synchronizes the rendered slice in the second viewer.



### Tip

A list of all parameter connections is displayed in the **Parameter Connections Inspector** View (which can be opened via the menu bar, **View** → **Views** → **Parameter Connections Inspector**). Right-click the connections for a context menu with various options.

For further information on parameter connections, please refer to the MeVisLab Reference Manual.

This is the end of this example. The full network is delivered with the demos of MeVisLab (available via **Help** → **Welcome**).

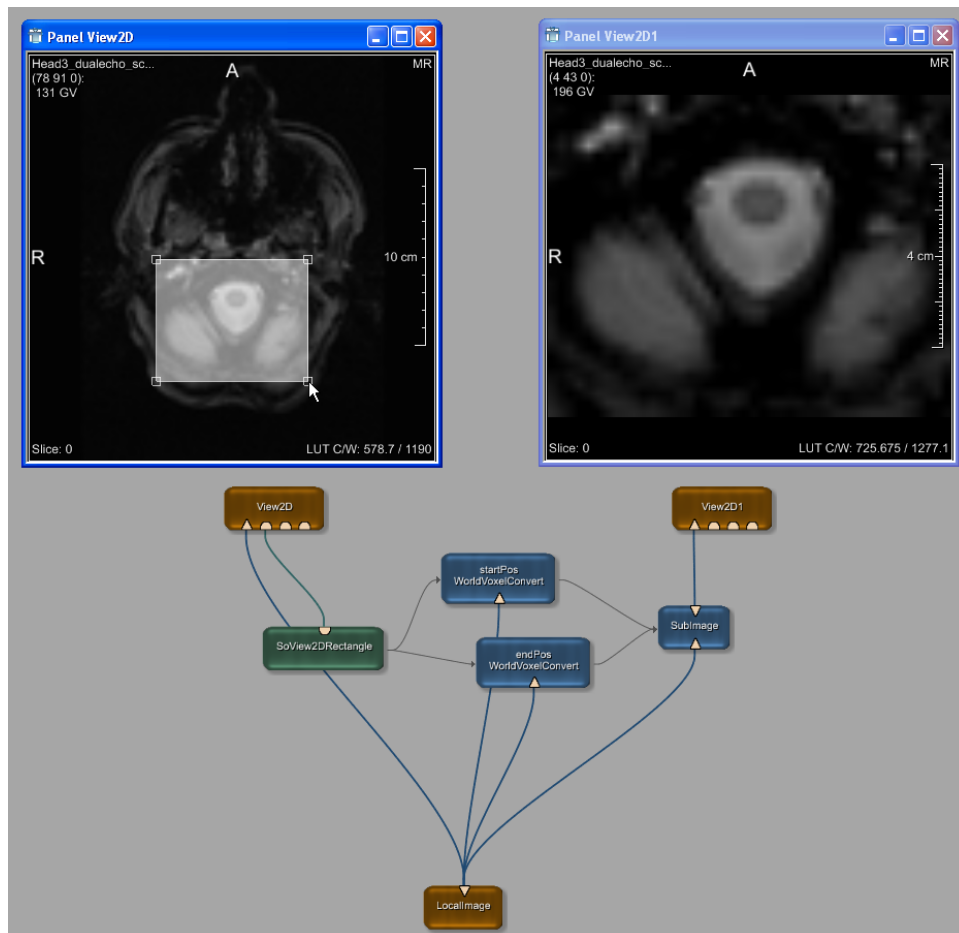
# Chapter 5. Defining a Region of Interest (ROI)

In the following chapter, we will walk through the creation of a network that allows defining a 2D region of interest (ROI), that is by selecting a region of the image in the first viewer, the selected region is displayed as a subimage in a second viewer.

- [Section 5.1, “Creating a Viewer with a Selection Rectangle”](#): adding a first viewer with a selection rectangle
- [Section 5.2, “Adding a Second Viewer for the Subimage”](#): adding a viewer for a subimage
- [Section 5.3, “Adding the Interactivity for the Viewers”](#): adding interaction between the viewers

The resulting network looks as follows:

**Figure 5.1. Example Network ROISelection**



In this chapter, we will use the terms “world position” (absolute) and “voxel position” (relative to the image), which are discussed in detail in the chapter [Chapter 10, Excursion: Image Processing in ML](#).

## 5.1. Creating a Viewer with a Selection Rectangle

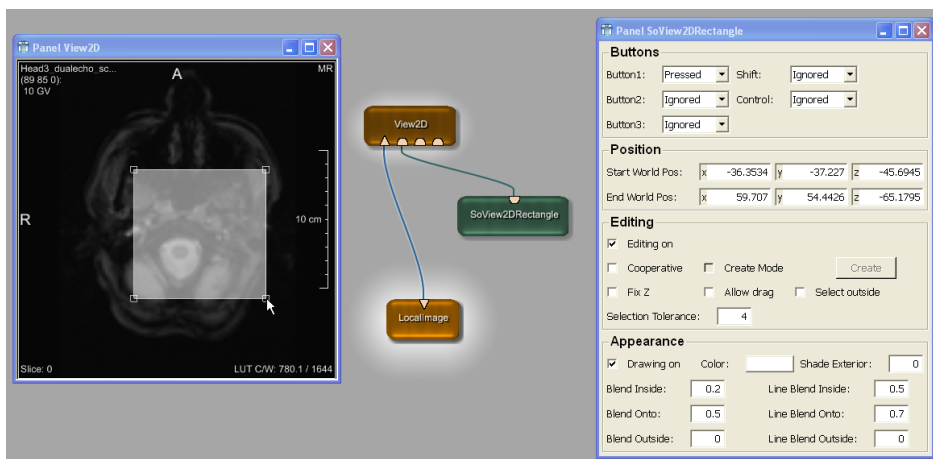
The first part is building a simple network with an image load module, a viewer, and a module that allows for drawing a selection rectangle.

1. Add `LocalImage` and the `View2D` module to the new network and connect their image connectors.
2. To display the usually hidden Inventor inputs of `View2D`, right-click `View2D` and select **View2D Options** → **Show Inventor Inputs**
3. Add the Open Inventor module `SoView2DRectangle` and connect its output to the first `View2D` Open Inventor input connector.

The module help offers the following purpose for the module: “The `SoView2DRectangle` module allows for a drawing and interactive adjustment of a 2D rectangle in a 2D viewer. Note: although this module is called `SoView2DRectangle`, it actually draws a 3D box.” (The latter is the reason why the world positions are given in 3D.)

A double-click on `SoView2DRectangle` opens its panel. For displaying the subimage, the world positions will be crucial.

**Figure 5.2. Viewer with Selection Rectangle**



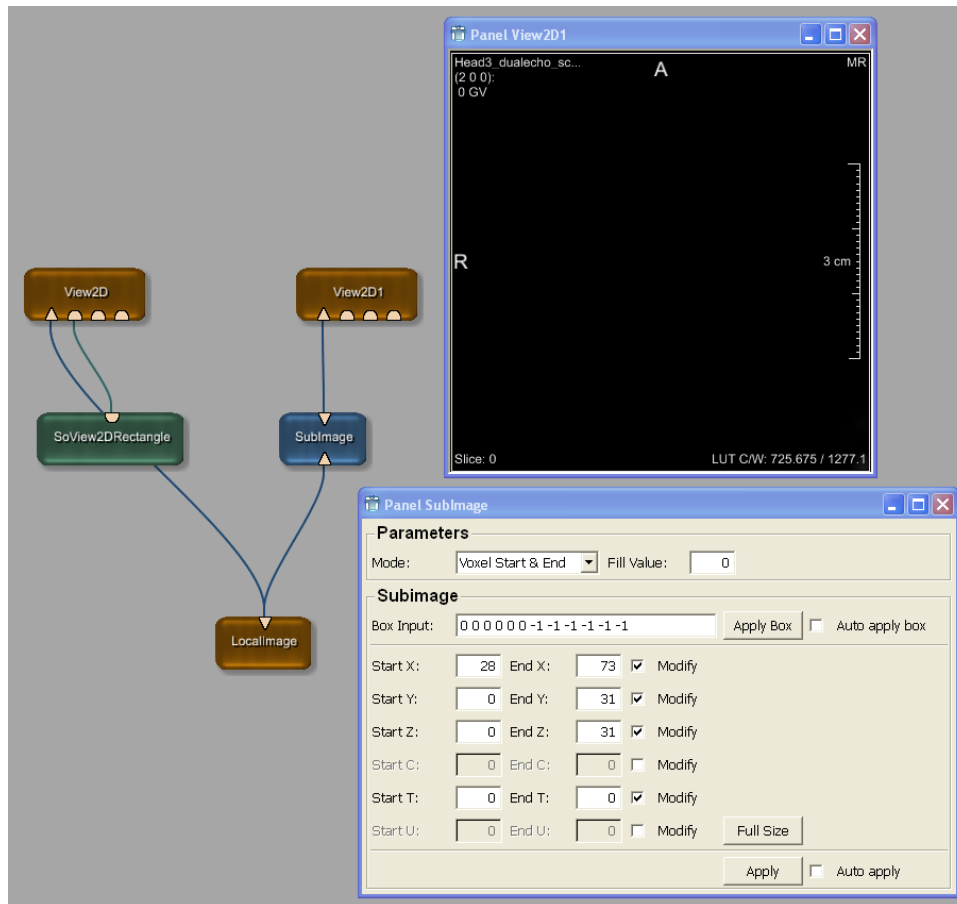
## 5.2. Adding a Second Viewer for the Subimage

Add the second viewer part, which consists of two modules:

- a `SubImage` module for cutting out the selected region
- and another `View2D` module.

The module help of `SubImage` offers the following purpose and usage tips for the module: “This module extracts subimages from its input image. [...] Connect an input image, set the coordinate mode and the size and position of the subimage.”

**Figure 5.3. Viewer for the Subimage**



Of course, since we have not yet defined how the world positions of `SoView2DRectangle` are connected to the subimage, nothing is displayed.

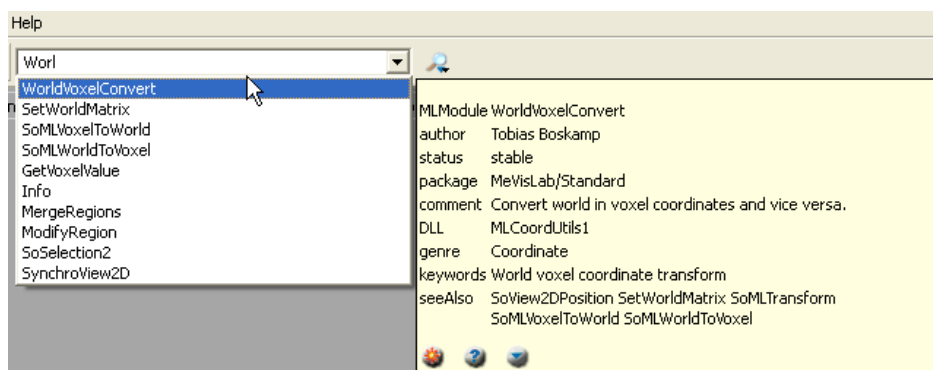
## 5.3. Adding the Interactivity for the Viewers

In the third step, we add the interactivity. The problem in connecting the modules `SoView2DRectangle` and `SubImage` is that the world positions offered by the first modules need to be translated to voxels positions for the latter.

For such translation tasks, there are several modules that convert values from one type to the other.

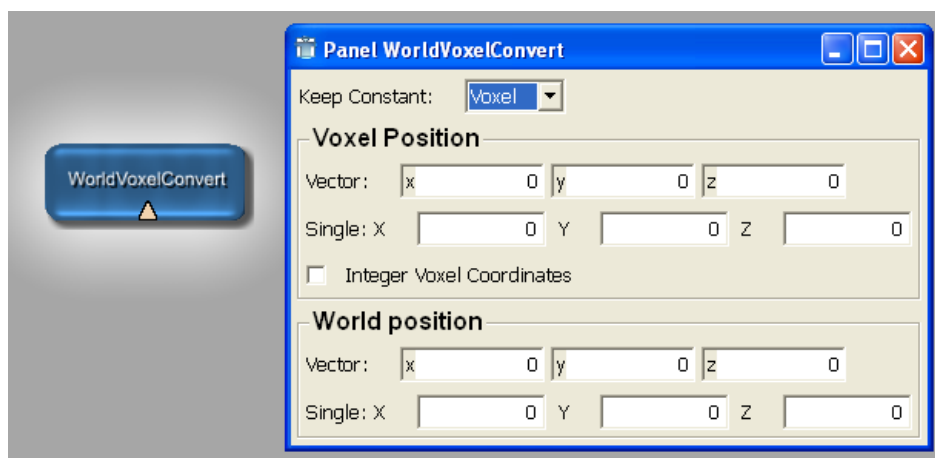
1. As we need world and voxel, enter those words in the quick search to find the module:

**Figure 5.4. Searching for World to Voxel Conversion**



WorldVoxelConvert converts world into voxel positions (or vice versa), either as vector or as single float values.

**Figure 5.5. WorldVoxelConvert Panel**



In our case, we need two conversions, for the start and end positions separately.

2. Add `WorldVoxelConvert` a second time by selecting the module and duplicating it, either via **Edit** → **Duplicate** or by pressing the respective keyboard shortcut.
3. Name the instances accordingly, for example “startPos” and “endPos”. For this, select **Edit Instance Name** in the module's context menu.



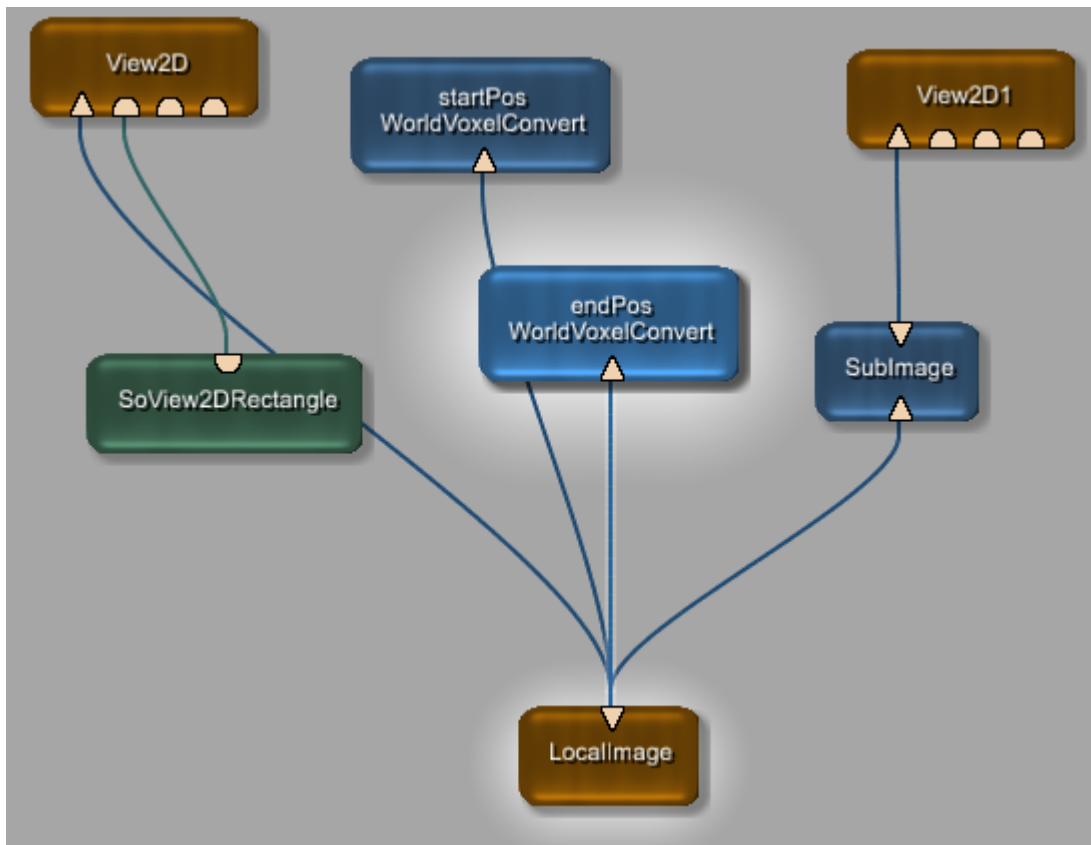
### Tip

Alternatively, use the shortcuts **F2** (Windows and Linux) or **ENTER** (Mac OS X). For a complete list, see the MeVisLab Reference Manual, chapter “Shortcuts”.

4. Both `WorldVoxelConvert` modules need the original image for obtaining the world-to-voxel matrix, so connect them to `LocalImage` (the image output can be connected to an unlimited number of modules).



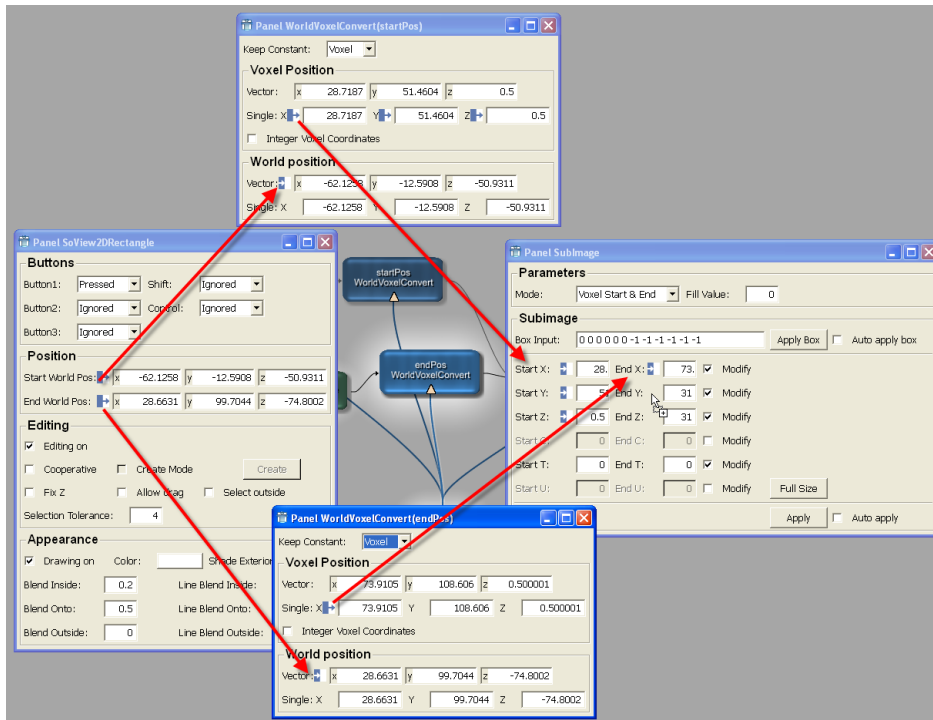
**Figure 5.6. WorldVoxelConvert Modules Added**



5. For the parameter connections, proceed as follows:

- a. Connect the `SoView2DRectangle` **Start World Position** to the `WorldVoxelConvert(startPos)` **Word Position Vector**.
- b. Similarly, connect the `SoView2DRectangle` **End World Position** to the `WorldVoxelConvert(endPos)` **Word Position Vector**.
- c. Connect the converted values from `WorldVoxelConvert(startPos)`, that is the **Single X, Single Y** and **Single Z** values, to the respective `SubImage` **Start X, Start Y** and **Start Z** values.
- d. Similarly, connect the converted values from `WorldVoxelConvert(endPos)`, that is the **Single X, Single Y** and **Single Z** values, to the respective `SubImage` **End X, End Y** and **End Z** values.

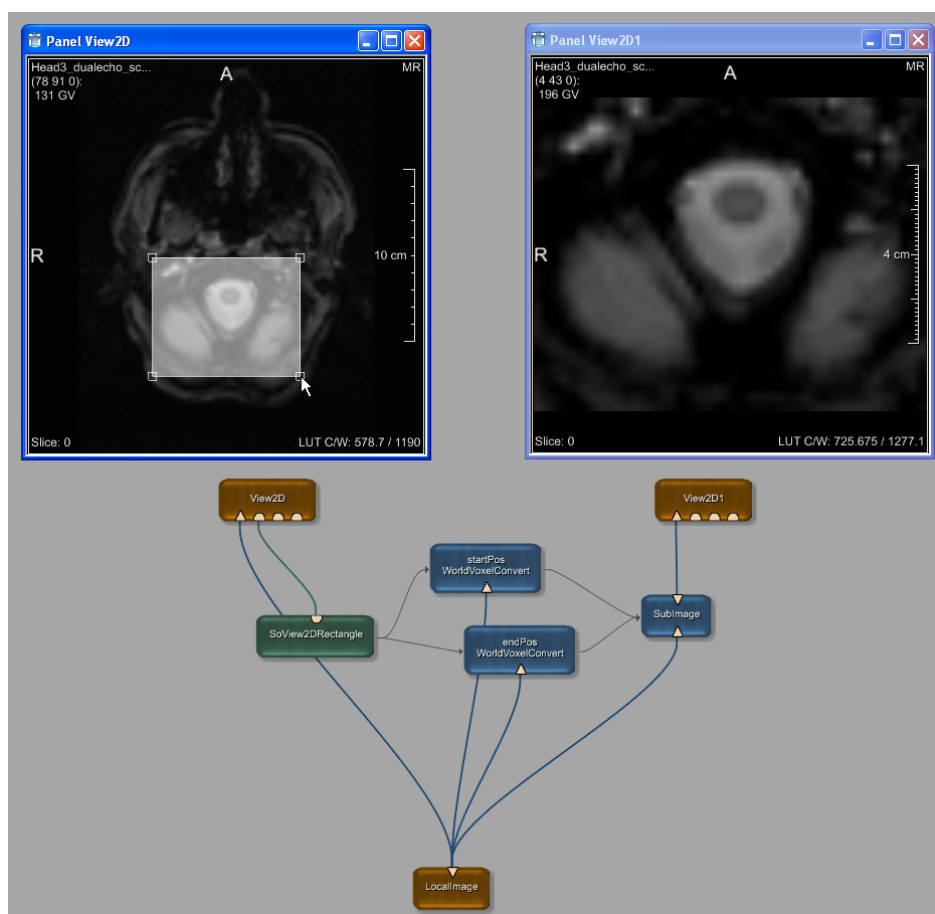
**Figure 5.7. Adding the Parameter Connections**



- At last, check the option **Auto apply** on the SubImage panel (bottom right corner), so that any changes of the selected region in the first viewer are updated automatically in the second viewer.

Now the network is fully functional.

**Figure 5.8. Example Network ROI Selection**



This is the end of this example. The full network is delivered with the demos of MeVisLab (available via **Help** → **Welcome**).

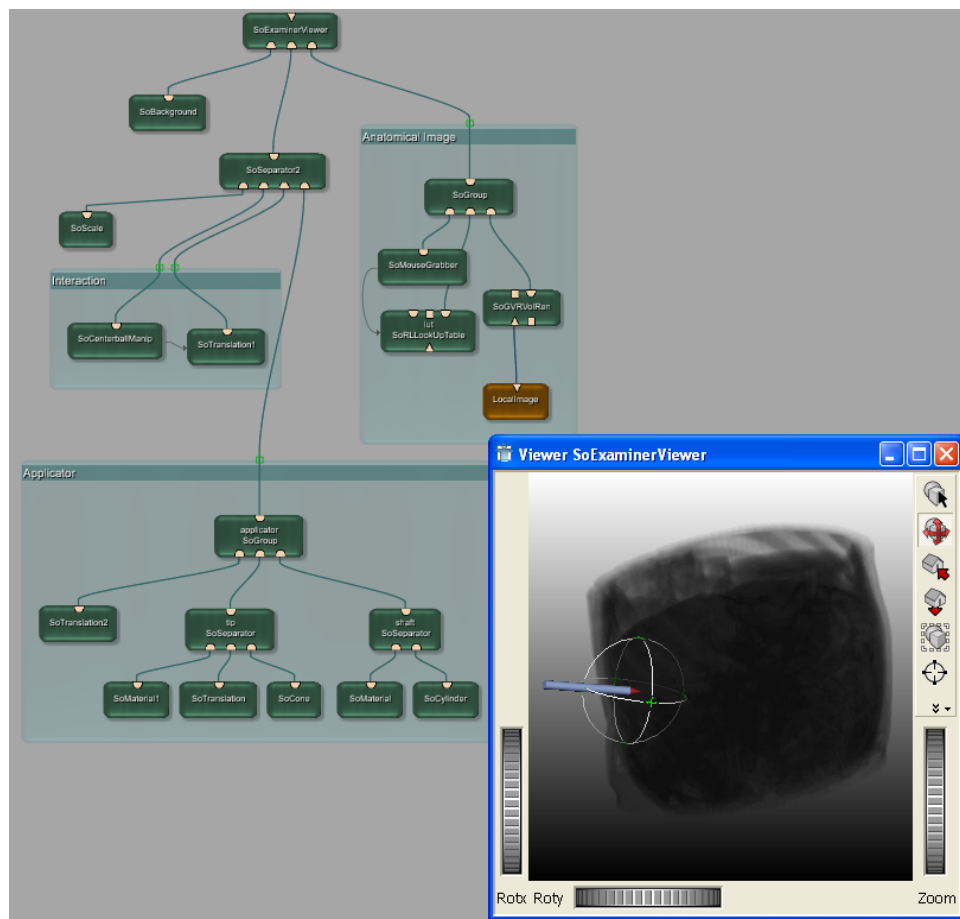
# Chapter 6. Creating an Open Inventor Scene

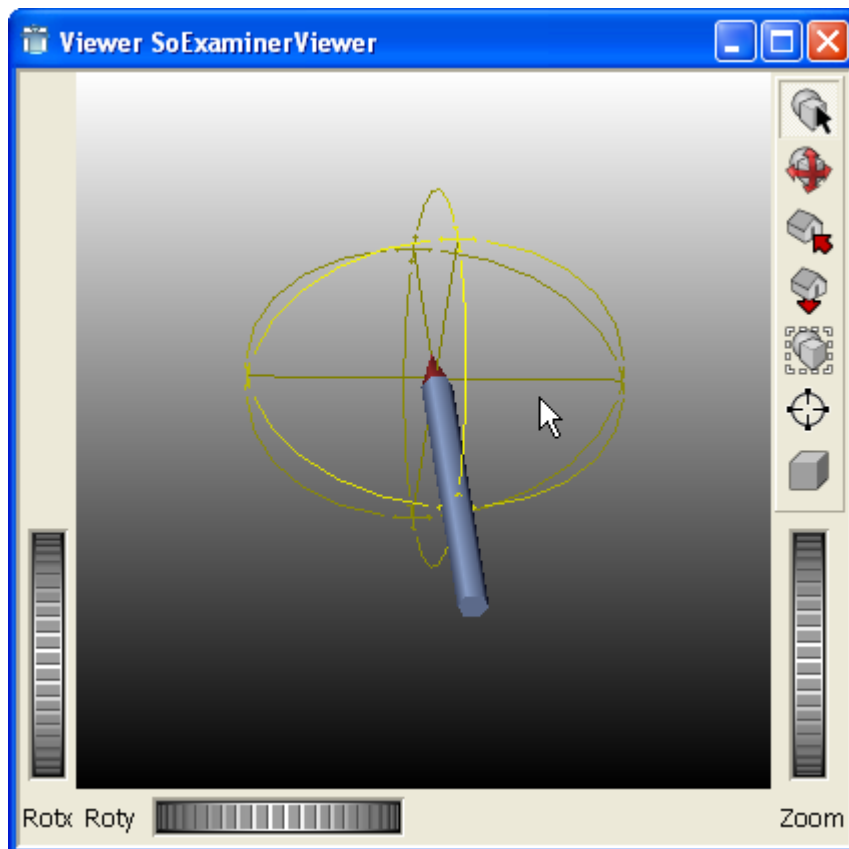
In the following chapter, we will walk through the creation of an Open Inventor scene.

- [Section 6.2, “Creating the Applicator”](#)
- [Section 6.3, “Creating the Interaction”](#)
- [Section 6.4, “Creating the Anatomical Image”](#)
- [Section 6.5, “Finishing the Complete Open Inventor Scene”](#)

Here a look at what we want to accomplish: a dynamically definable applicator shall be placed at a position and an angle relative to the rendering of an anatomical image:

**Figure 6.1. Example Network: Open Inventor Result**



**Figure 6.2. Applicator Only**

The applicator shall be able to be moved within the viewer (navigation) and also be able to be repositioned (interaction) with the tip pointing to the body.

The data shall be displayed in 3D mode. In addition, the output shall have the windowing functionality of the standard Output Inspector.

In the resulting network, modules will be grouped; however, this has no effect on the functionality we will build.

## 6.1. Introduction to Open Inventor

Open Inventor is an object-oriented 3D toolkit developed by Silicon Graphics (SGI) offering a comprehensive solution to interactive graphics programming problems.

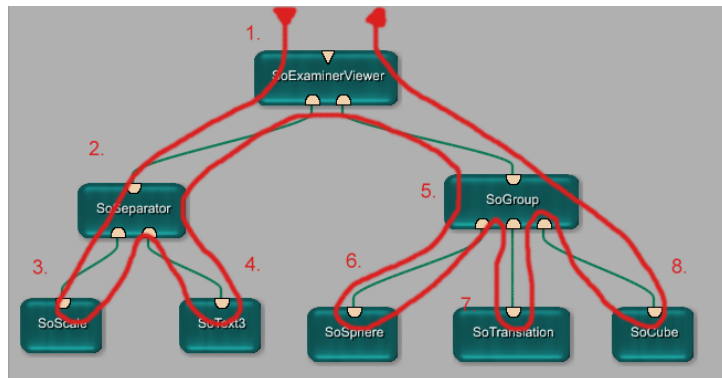
Inventor scenes are organized in structures called scene graphs. A scene graph is made up of nodes, which represent 3D objects to be drawn, properties of the 3D objects, nodes that combine other nodes and are used for hierarchical grouping, and others (cameras, lights, etc). These nodes are accordingly called shape nodes, property nodes, group nodes and so on. Each node contains one or more pieces of information stored in fields. For example, the Sphere node contains only its radius, stored in its radius field.

The MeVisLab implementation of Open Inventor is based on the original SGI source code that was released to the public in 2000. It is suited for use with MeVisLab but can also be used independently. The MeVisLab modules can be used for rendering and viewing both image data and arbitrary Open Inventor objects as well as for interacting with images. Inventor modules function as Inventor nodes, so they may have input connectors to add Inventor child nodes (modules) and output connectors to link themselves to Inventor parent nodes (modules).

Characteristics of an Open Inventor scene graph:

- Scene objects are represented by nodes.
- Size and position is defined by transformation nodes.
- A rendering node represents the root of the scene graph.
- Nodes are rendered in the order of traversal.
- Nodes on the same level are traversed from left to right.
- All modules that are derived from `SoGroup` offer a basically infinite number of input connectors (a new connector is added for every new connection).

**Figure 6.3. Traversing in Open Inventor**



Typical functions of Open Inventor modules are:

- Draggers and manipulators
- Group nodes
- Light sources
- Transformations
- Cameras
- 3D Viewers
- Geometric objects (Cone, 3D Text, Nurbs, Tri.Meshes, etc.)
- Object properties (Textures, Colors, Materials, etc.)

The order of traversal is very important, and its effects will be shown in detail in the following example.

For further information on Open Inventor modules in MeVisLab, please refer to the Inventor Reference and the Inventor Module Help. For information on Open Inventor, we recommend the following literature:

- *The Inventor Mentor* by Josie Wernecke (ISBN 0-201-62495-8): This book provides basic information on programming with Open Inventor. It includes detailed program examples in C++ and describes key aspects of the Open Inventor toolkit, including its 3D scene database, node kits, interactive manipulators, the Inventor Component Library, which contains editors and viewers, and the Open Inventor file format.
- *The Inventor ToolMaker* by Josie Wernecke (ISBN 0-201-62493-1): The Inventor Toolmaker provides advanced information on extending Open Inventor by creating new C++ classes and customizing existing classes. Detailed examples and discussion show how to create new nodes, actions, elements, fields, node kits, draggers, manipulators, engines, and components.



### Tip

For online links to these books and other resources, see the MeVisLab website (<http://www.mevislabs.de>).

## 6.2. Creating the Applicator

1. As a first element, we need the shaft of the applicator. For this, start by adding a `SoCylinder` module.
2. As we want to keep the applicator shaft and tip basically independent, we can already add a `SoSeparator` module here which comes with an in-built viewer. Connect the two modules and set the parameters for the cylinder.



### Tip

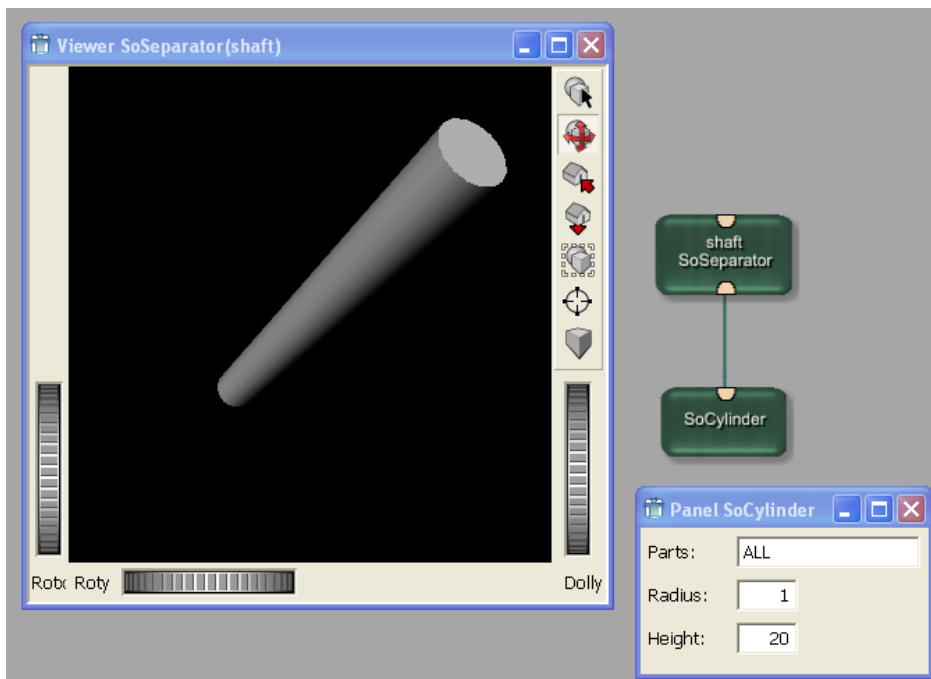
Several Open Inventor modules come with an in-built viewer, like `SoSeparator`, `SoGroup`, `SoRenderArea` and more. For a complete viewer experience, use `SoExaminerViewer` and its associated macro module `SceneInspector`.



### Note

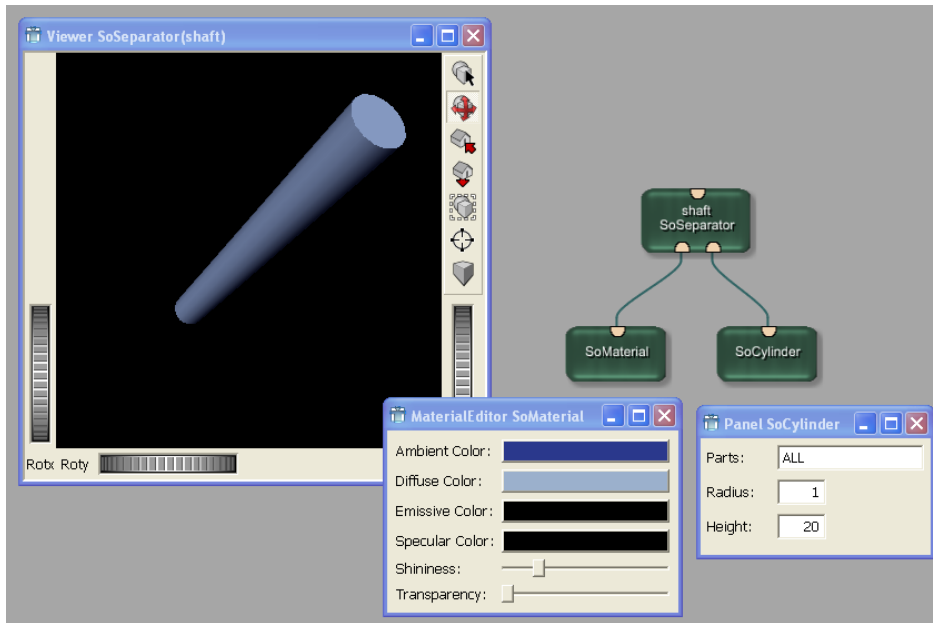
Each of the viewers have their own persistent settings. So if you copy and paste such modules into another network, the zoom settings etc. will be those of the previously used state! If confused, always add fresh modules via the search or the **Modules** menu.

**Figure 6.4. Creating the Applicator Shaft**



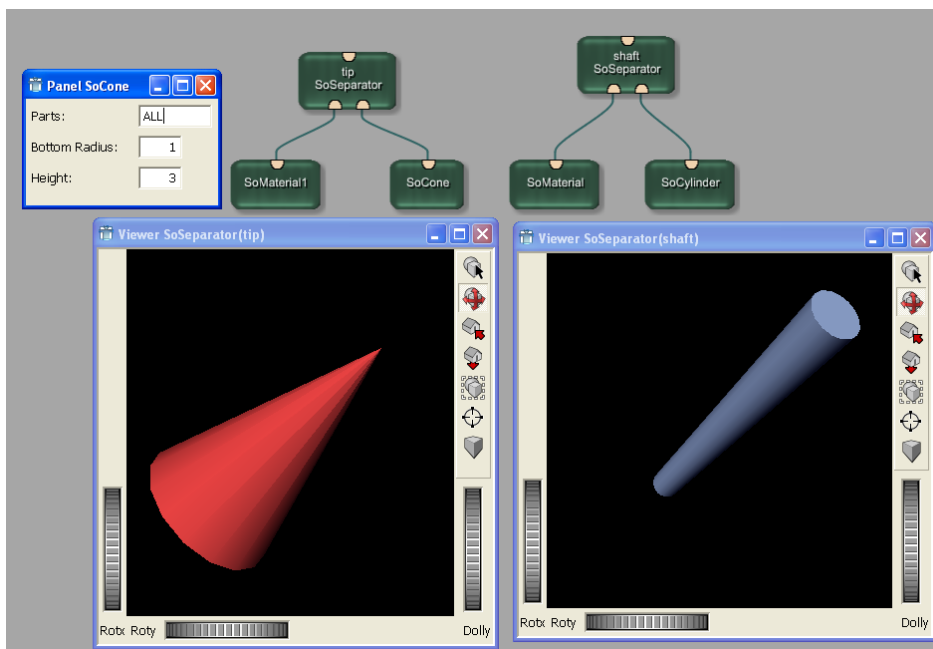
3. Usually, such Open Inventor objects will be colored. Add the `SoMaterial` module before the `SoCylinder` module and edit the material settings. Feel free to play around with the color settings.

**Figure 6.5. Coloring the Applicator Shaft**



4. In a next step, we will create the applicator's tip. For this, add a `SoCone` module and also another `SoMaterial` and `SoSeparator` module to build a construction similar to the shaft.

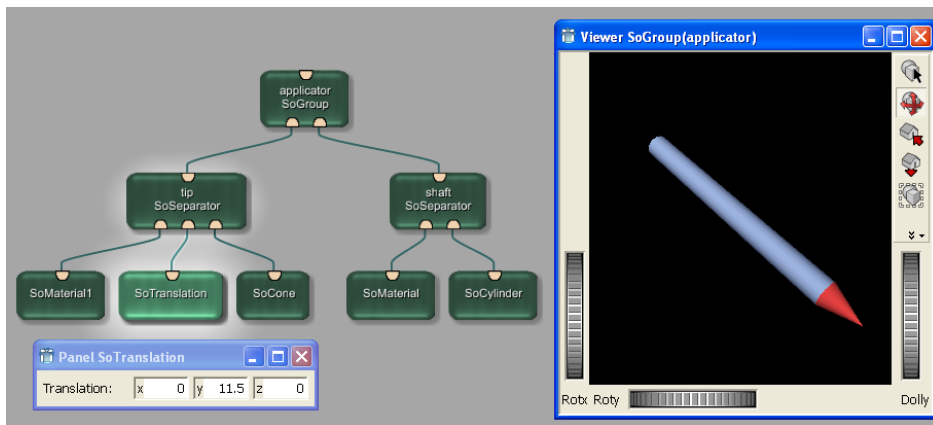
**Figure 6.6. Adding an Applicator Tip**



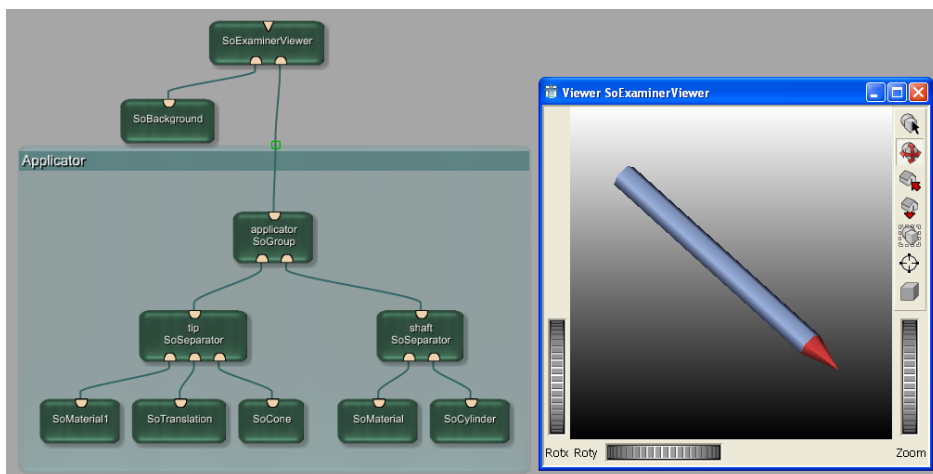
To combine the two independent elements (shaft and tip), we have to a) combine them and b) translate the tip (or shaft) in relation to the other, otherwise the two Open Inventor elements would be placed at the same position, namely the origin of the Inventor's world coordinate system  $[0,0,0]$ . (For more information on coordinate systems, see [Chapter 10, Excursion: Image Processing in ML.](#))

5. For the translation, add a `SoTranslation` module in front of to the cone, and set the translation to (in this case) "11.5". The `SoGroup` module has an in-built viewer, so that you can preview the resulting applicator. It can be rotated in the viewer.



**Figure 6.7. Adding Translation and Grouping**

6. For a finishing touch, add a `SoExaminerViewer` for display and a `SoBackground`. The latter adds a grey gradient background that gives a more 3-dimensional impression of the rendered Open Inventor scene.
7. For easier handling, create a group for the two parts of the applicator. Select the modules that belong to the applicator, right-click them and select **Add to New Group**. Enter an appropriate name like “applicator”. The new group appears in the workspace.

**Figure 6.8. Finishing the Applicator**

## 6.3. Creating the Interaction

Although the applicator created in the last section is complete, it is not yet functional so that you can easily point the tip to a position. For this, some interactivity must be enabled.

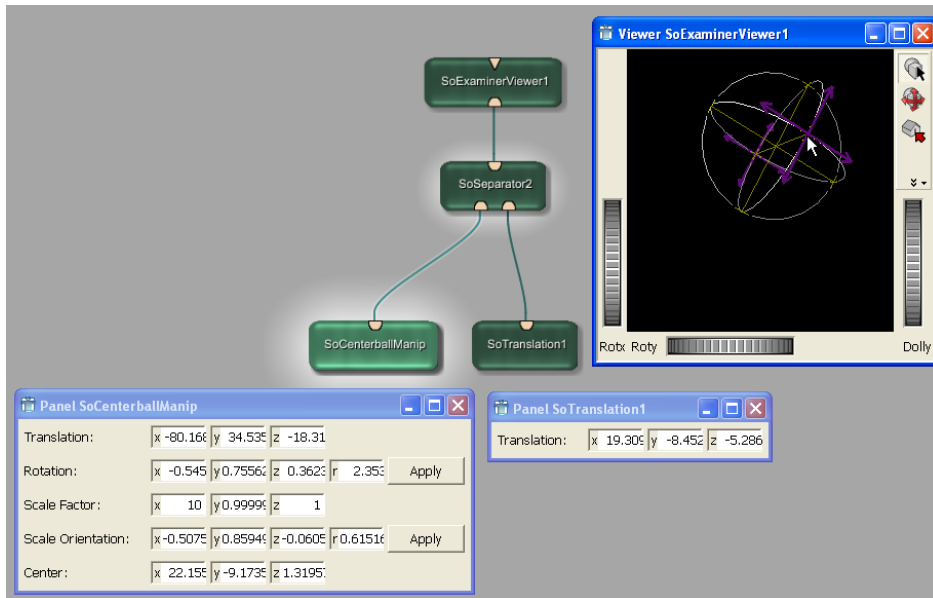
The first module necessary for this is `SoCenterballManip`. In the Inventor Reference, the following information can be found for this module:

“`SoCenterballManip` is derived from `SoTransform` (by way of `SoTransformManip`). When its fields change, nodes following it in the scene graph rotate, scale, and/or translate. [...] On screen, this manipulator will surround the objects influenced by its motion. This is because it turns on the **surroundScale** part of the dragger.”

This means that once we put an object in the middle of the sphere opened by this module, it can be moved around with it.

1. To keep the interaction separate from the applicator, add another separator.
2. Then add the modules `SoCenterballManip` and `SoTranslation`. The translation module is necessary to position the centerball (as the latter is foremost intended for rotation and not perfect for translation).

**Figure 6.9. Using `SoCenterballManip`**



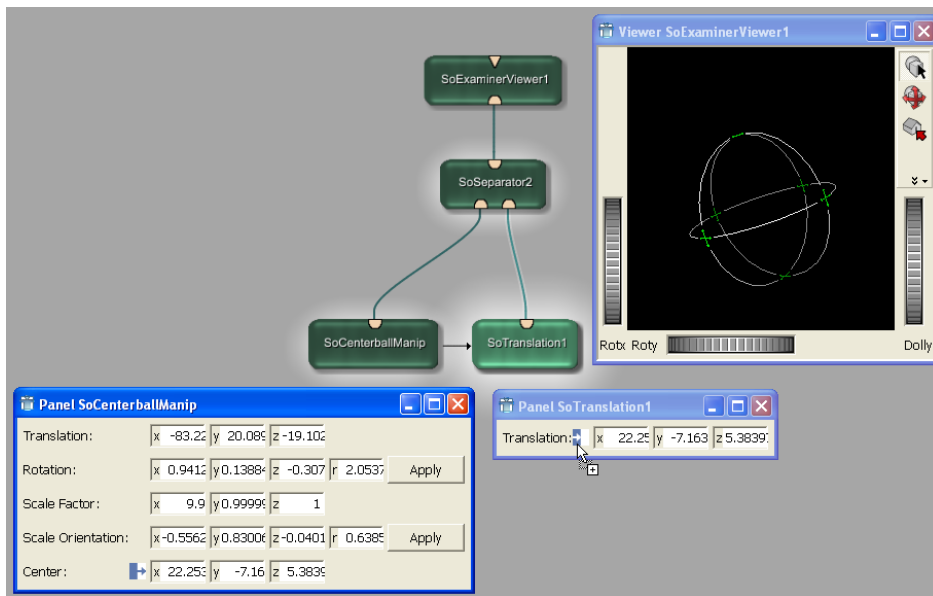
3. To connect the translation of the modules, a parameter connection has to be established between the `Center` field of `SoCenterballManip` and the `Translation` field of `SoTranslation`. This is done by opening the panels, clicking near the `Center` field and dragging it onto the other panel until a little plus sign appears. The parameter connection is drawn as a thin line between the modules, always starting at the modules' side (never on top or bottom, like data connections do).



### Tip

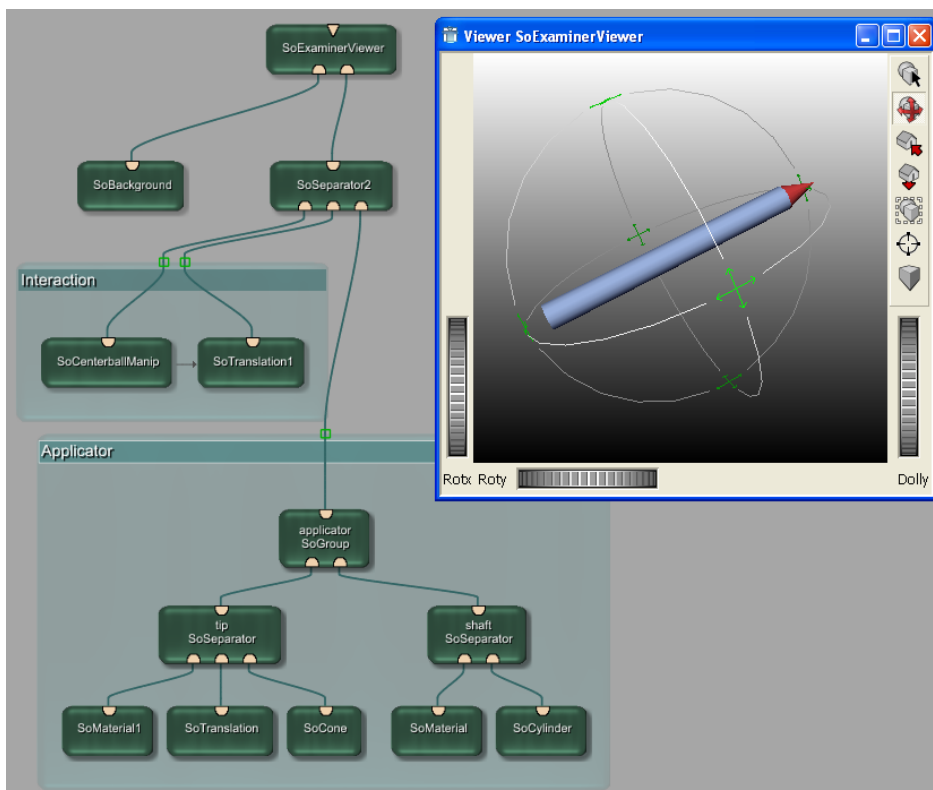
For an overview of all parameter connections, open the **Parameter Connections Inspector** via the menu bar, **View → Views → Parameter Connection Inspector**.

**Figure 6.10. Connecting Parameters**



- Now we can combine the interaction part and the applicator. For this, connect the applicator to the second separator.

**Figure 6.11. Combining Interaction and Applicator**



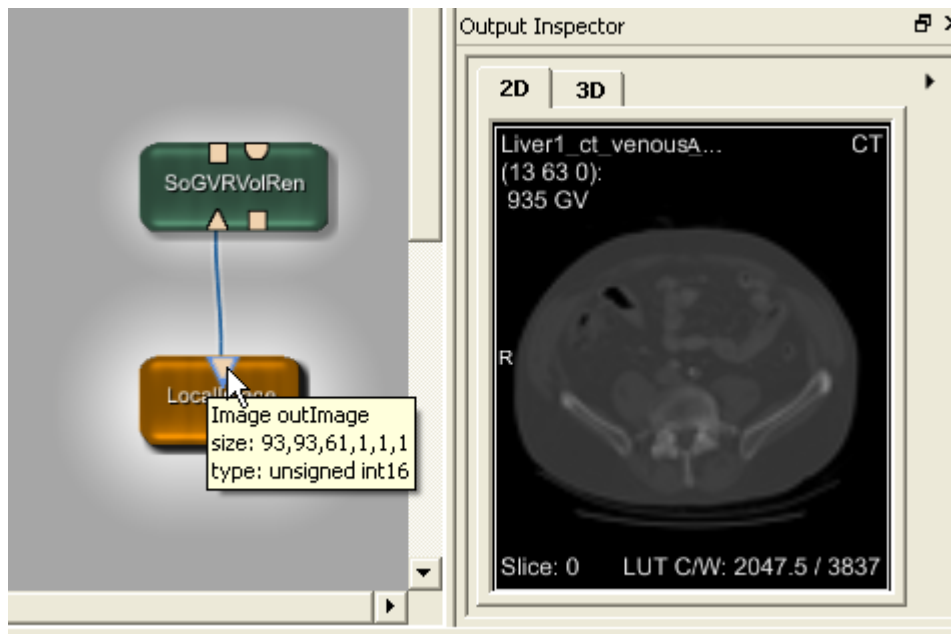
The applicator can now be rotated or dragged into any direction by using the handles on the manipulation sphere.

## 6.4. Creating the Anatomical Image

Last not least we need the 3D image at which the applicator shall be positioned.

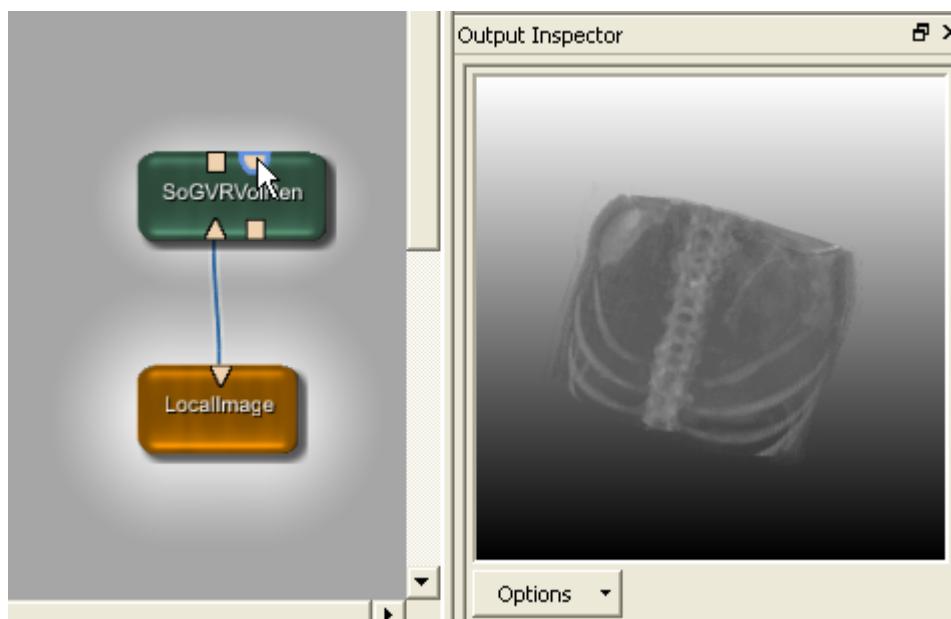
1. As first step, add a `LocalImage` module. Select an image from the demo data folder, for example a liver set. You can view the result in the normal **Output Inspector**.

**Figure 6.12. Loading a Local Image**



2. For the 3D display, add a `SoGVRVolumeRenderer` module. Behind this hides a rather potent module called GigaVoxel Renderer. It comes with many features — open its panel to have a look at the options.

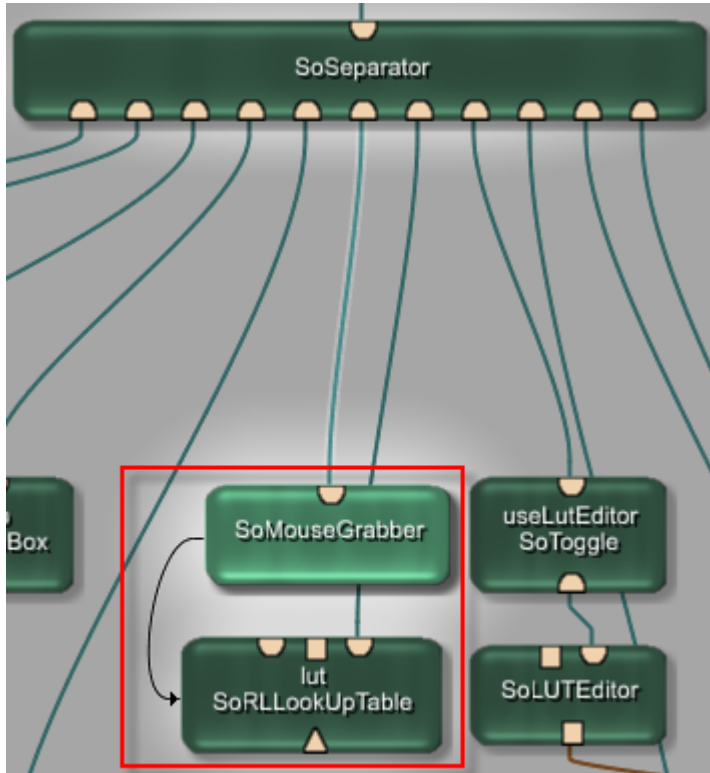
**Figure 6.13. Adding the GigaVoxel Renderer**



For the windowing, we need two modules: `SoMouseGrabber` and `SoRLLookUpTable` module. Instead of building this functionality from scratch, we can take the easy way and copy those modules and their parameter connection from the internal network of the `View3D` module.

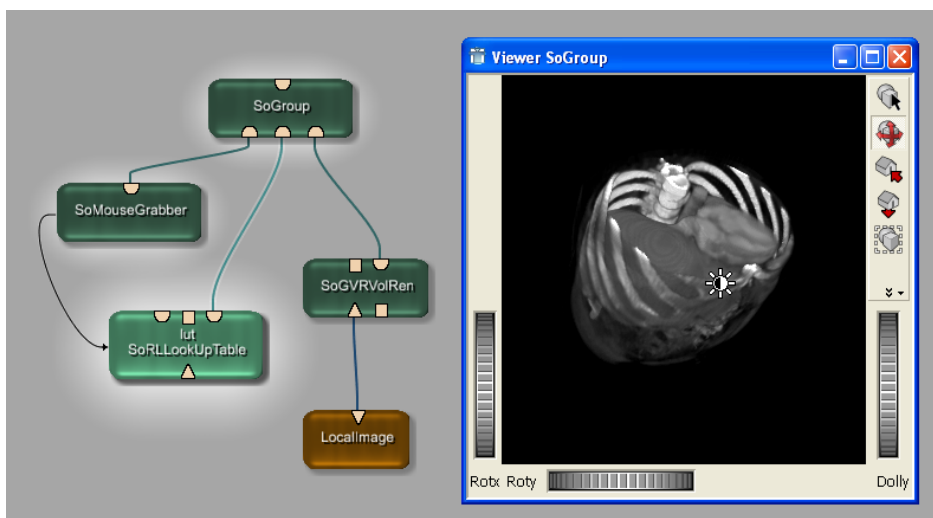
3. Add a `View3D` module via the quick search and open its internal network (via the context menu). Select the two modules and copy them. This will also copy the parameter connection between them.

**Figure 6.14. Copying the Windowing Modules from View3D**



4. Then add the modules to your applicator network and connect them to the `SoGroup` module, in front of the rendering module.

**Figure 6.15. Adding the Windowing to the Applicator**



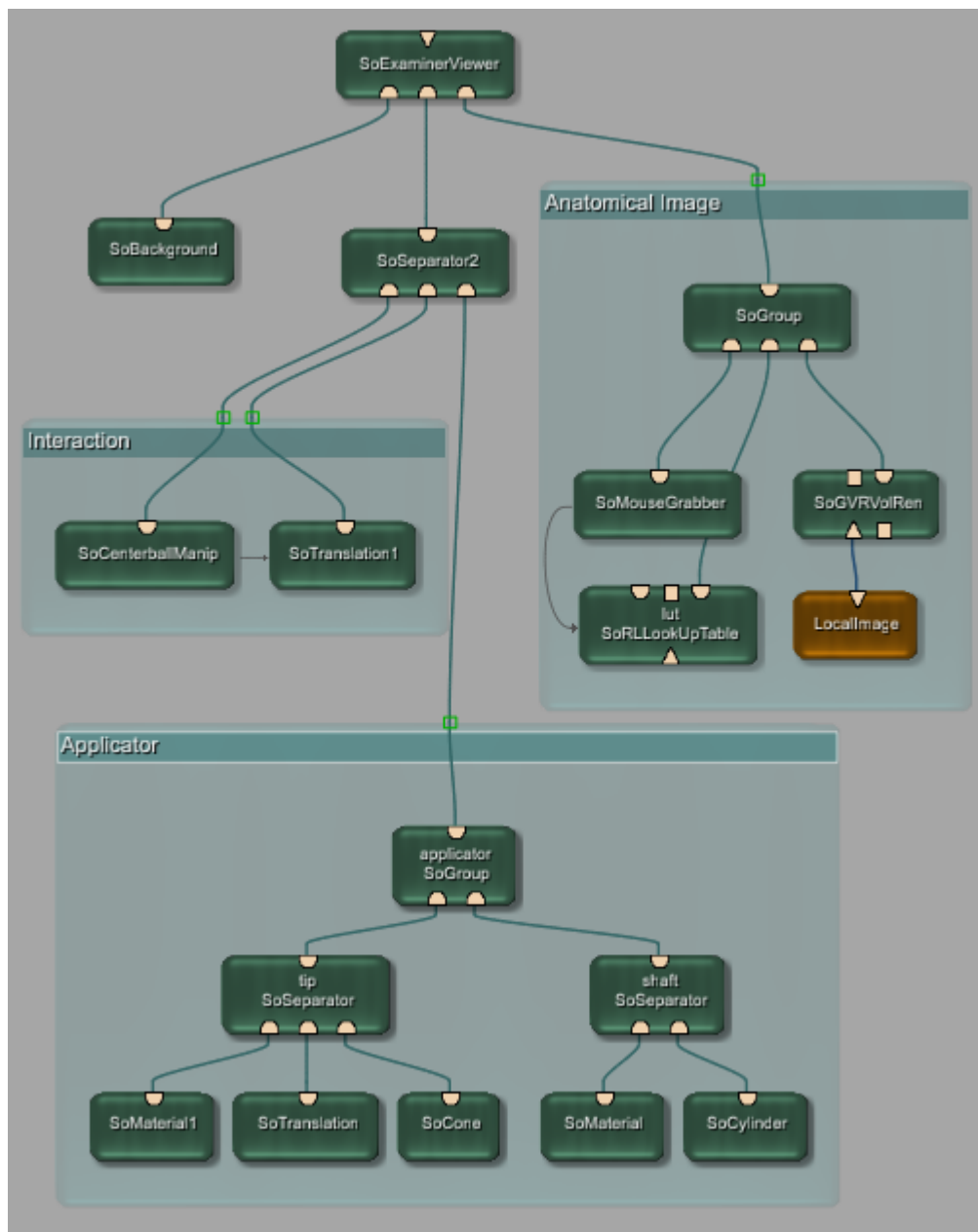
Afterwards, delete the `View3D` module.

## 6.5. Finishing the Complete Open Inventor Scene

The three elements of the scene — applicator, interaction and anatomical image, preferably grouped, now have to be combined to result in one Open Inventor scene.

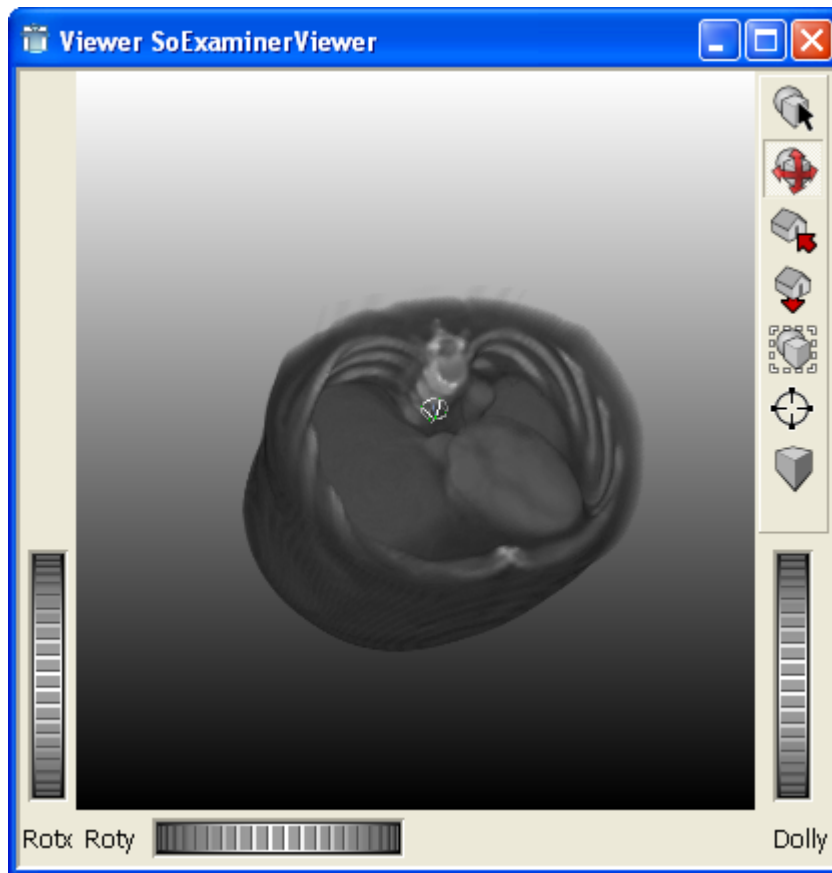
1. First, connect all three groups to the same `SoExaminerViewer`. Make sure that the applicator and its interaction sphere are connected via a separator.

**Figure 6.16. Combining the Groups**



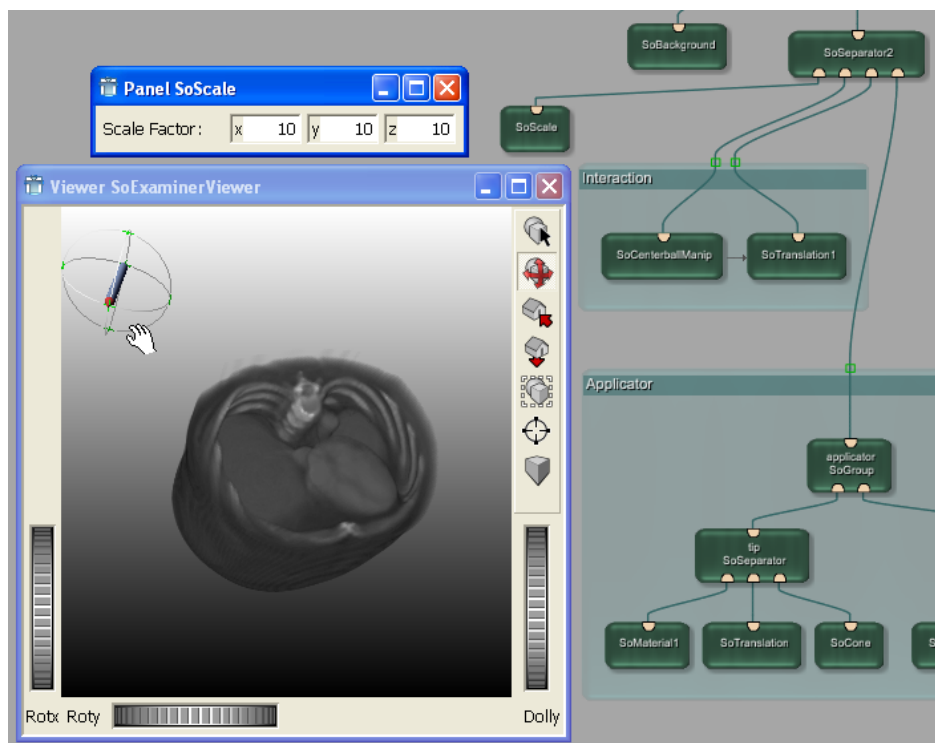
### Note

Because the scene with the anatomical image can be rendered with transparencies, add it right-most to the viewer so it is rendered last.

**Figure 6.17. Combined Graphic Elements**

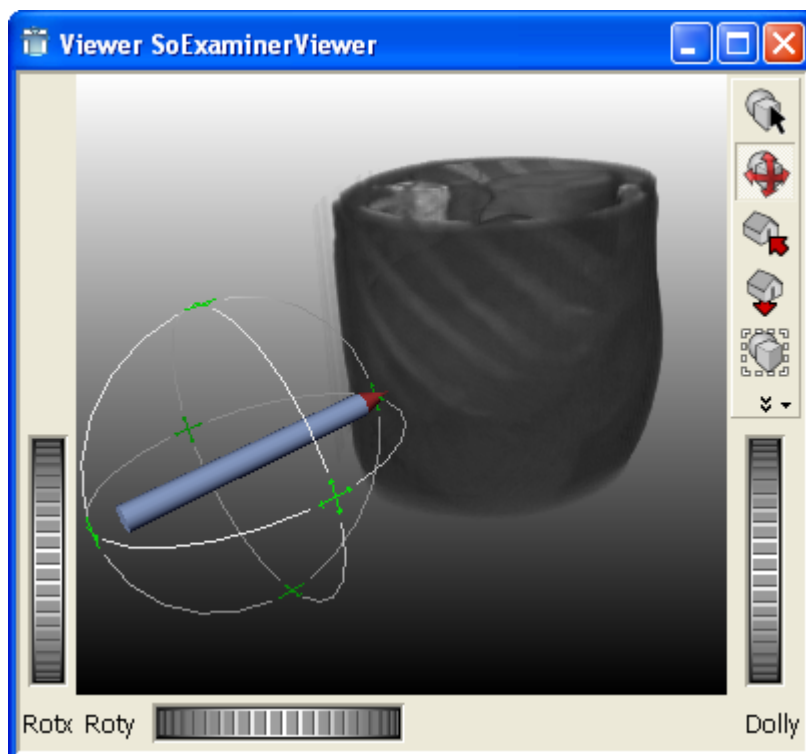
2. A look at the viewer tells us that the relative sizes of the graphic elements need to be aligned. This can be done by adding the scaling module `SoScale`, either to the applicator or the image. In our case, we will add it to the applicator, that means to the `SoSeparator` module. A scale factor of 10 in all directions is sufficient.

**Figure 6.18. Adding the Applicator Scaling**



3. At last, take the applicator and move it to the body to point at whatever spot you want to point at.

**Figure 6.19. Original Applicator/Interaction Arrangement**

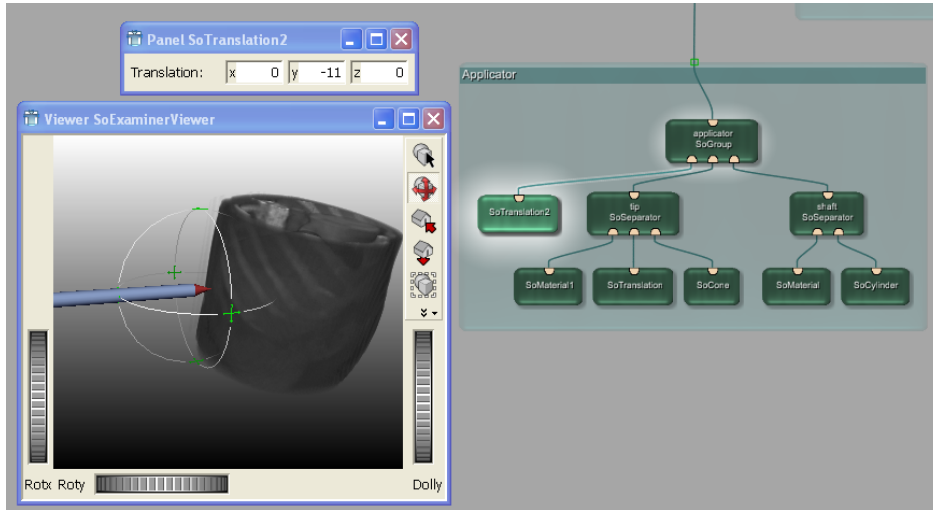




Looking at the result, it might not be the best idea to have the applicator tip at the edge of the sphere which is always aligned by its center. It may be sensible to place the tip into the sphere's center instead.

4. Add another `SoTranslation` module. It needs to have an effect on the applicator, so it needs to be added to the applicator's `SoGroup` module.

**Figure 6.20. Improved Applicator/Interaction Arrangement**



This is the end of this example. The full network is delivered with the demos of MeVisLab (available via **Help** → **Welcome**).



### Tip

In the chapter [Chapter 9, Developing a Macro Module for an Applicator](#), the applicator modules will be used as the starting point for programming a Python macro.

---

# Chapter 7. Starting Development with Package Creation

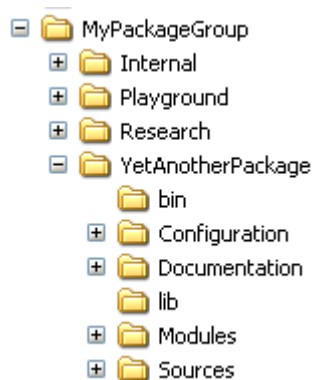
## 7.1. What are Packages

As of MeVisLab 2.0, modules and projects come in a package structure, which offers an improved modularity and granularity.

A package is a self-contained directory structure that contains the following components:

- PackageGroup
  - PackageName
    - Package.def
    - Modules
    - Sources
    - Configuration
    - Documentation
    - lib
    - bin

**Figure 7.1. Example for a Package Tree**



In this example, we have a PackageGroup "MyPackageGroup". Below it, four packages can be found (Internal, Playground, Research, YetAnotherPackage). Below each package, the typical folders can be found. (This example was generated with the Project Wizard in MeVisLab.)

A PackageGroup can contain any number of packages, and of course there can be different PackageGroups.

The PackageIdentifier is defined by "PackageGroup/PackageName", e.g. the MeVisLab Standard Package has the identifier "MeVisLab/Standard".



### Note

For more detailed information on packages, see the Package Structure documentation.

MeVisLab reads packages in the following order:

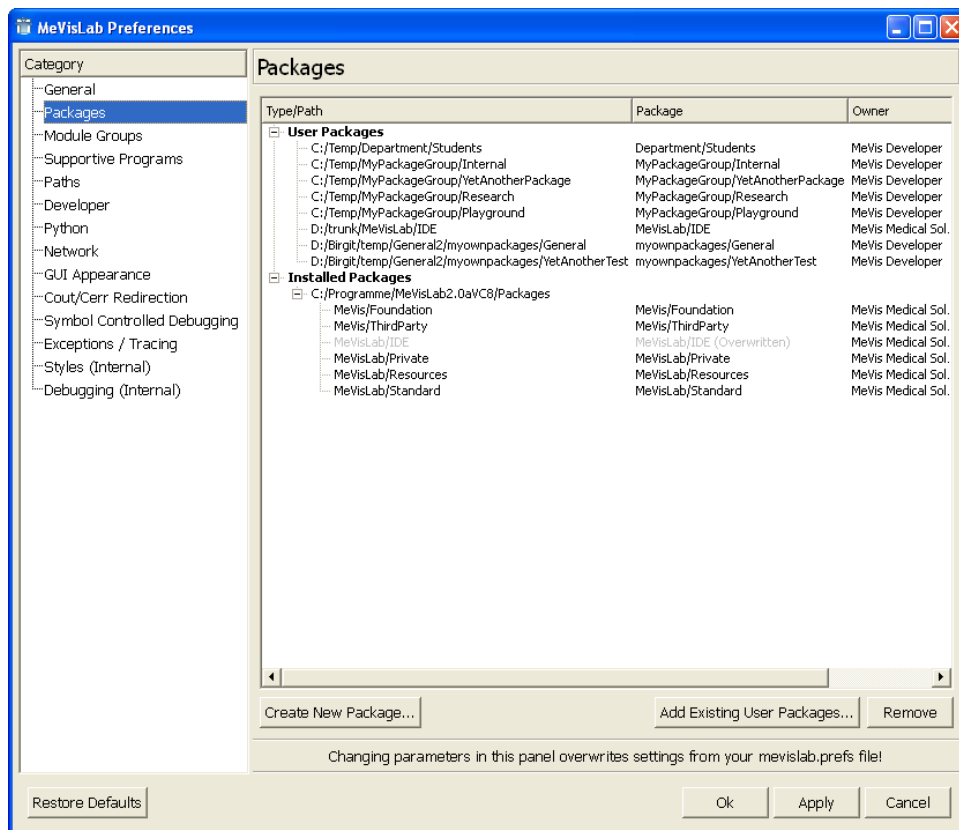
- the Packages directory in which MeVisLab was installed
- the directories given in the PackagePaths settings of the `mevislab.prefs` file
- the UserPackagePath (as set in the MeVisLab Preferences dialog)

Scanning is always two levels deep, never deeper. If a package with the same PackageIdentifier is found more than once, the last package found will overwrite the earlier packages (in the order given above). This way, your packages given by `mevislab.prefs` or your user packages can overwrite installed packages.

You can check your effective package structure in two ways:

- by using the ToolRunner, a meta-tool delivered with MeVisLab 2.0. See the ToolRunner documentation for details.
- by checking the MeVisLab Preferences, section “Packages”.

**Figure 7.2. Preferences — Packages**



In this dialog, the sequence of display is as follows (from top to bottom; higher entries overwrite lower entries):

- **User Packages:** packages found in the user path (packages in other paths can be added manually). These are the default packages for user-defined modules.
- **mevislab.prefs:** packages resulting from the paths given in the `.prefs` file.
- **Installed Packages:** packages resulting from an installation of e.g. MeVisLab SDK.

If a package with the same PackageIdentifier is found more than once, the last package found will overwrite the previously loaded packages. These will be greyed out and labeled “(Overwritten)”.

You can:

**Create New Package:** Opens the Package Wizard (see [Section 7.2, “Creating a User Package for Your Project”](#)).

**Add Existing User Packages:** Opens the default file browser so that you can add a user package. Folders are read recursively and all packages below them are automatically included.

**Remove:** Removes the selected user package from the path of MeVisLab. (Installed packages cannot be removed.) Removed user packages can always be re-added later.

## 7.2. Creating a User Package for Your Project

When you create new modules with the Wizard, you need to enter their package path. For your own modules, you always should have your own user package (and path). This is done as follows:

1. Run the Project Wizard (**File** → **Run Project Wizard**)
2. Select **New Package**. The Package Wizard opens.

**Figure 7.3. Package Wizard**

**Packages/New Package**

**Package Wizard**

General settings for your package

**Package Information**

Package Group: \*

Package Name: \*

Package Owner:

Package Description:

**Target Directory**

Target Directory: \*

**Import MeVisLab 1.6 Projects**

☐ Import MeVisLab 1.6 UserProjectPath into this Package

Import from UserProjectsPath:

**Info**

**Packages** are the way MeVisLab organizes projects. A package can contain any number of C++/Macro Modules, Installers, Documentation etc. The creation of an own package is mandatory for SDK users, all other wizards require a valid target package.

\* : Required fields

3. Create a new package with the Package Wizard. Enter the following:

- **Package Group:** Enter the package group in which your package should be saved. Enter a name, for example your company or site name. For our example, enter "Example".
- **Package Name:** Enter the package name. Select a typical user package name from the list or enter a new package name. For our example, select "General"
- **Package Owner:** Enter a package owner (meta description without actual effect).
- **Target Directory:** Select the target directory below which this package will be created.

4. Click **Create** so that the new package is created.

The new package is added to the User Package Path, including all subdirectories and files. The information entered in the dialog is saved in the `Packages.def` file. As adding a new package group alters the user package path, the module database has to be reloaded.

After reloading, your user package Example/General is ready for saving modules and projects.

---

# Chapter 8. Introduction to Macro Modules

Macro modules are implemented by means of the **MeVisLab Definition Language (MDL)** and the scripting languages Python or JavaScript. A macro module behaves like any other elementary (ML or Inventor) module in MeVisLab. However, no C++ has to be coded to implement a macro module.



## Tip

Based on macro modules, stand-alone applications can be created with MeVisLab. Prerequisite for this is a license for the Application Development Kit (ADK).

Like any other module, a macro module has to be declared within the MeVisLab module database in a module definition file (`*.def`), which has to be located in the `User Module Path`.

The MDL script implementation of a macro module, that is its interface definition (input-, output- and parameter fields) as well as its GUI definition, usually are written in a `*.script` file. The scripting is given in separate `*.py`/`*.js` files which need to be included in the `*.script` module definition file.

The definition of a macro module and the creation of all necessary files is supported by the ML Module Wizard, via **File** → **Run Project Wizard** (see the next chapter [Chapter 9, Developing a Macro Module for an Applicator](#)).

What you should know about macro modules:

- In most cases, macro modules encapsulate the “macro behavior” of an image processing and/or visualization pipeline (realized by a MeVisLab module network). Its functionality is defined by the macro module interface with inputs, outputs and parameters (fields). The interface is built as a combination of the interface elements of the modules in the underlying network, and of eventually new fields. The encapsulated module network is stored in a `<MacroModuleName.mlab>` file, which is also called the macro network of the module.

Why this encapsulation?

- In many cases, a desired module function can be built by connecting some elementary modules or macros that are already implemented.
- Certain processing pipelines may be of common use in a variety of further applications and it is convenient to encapsulate them in macro modules which can then be added easily to any network.
- The interface of an encapsulating macro module is more compact than the sum of all interfaces of the contained modules.
- Macro modules are defined on an abstract level. They can and do exist stand-alone without a corresponding macro network. In those cases, the module's functionality is implemented with scripting only. In most cases those macro modules encapsulate dynamic user interfaces without any image processing or visualization behind it. Examples for those modules are the MDL test modules, for example `TestBoxLayout`. They consist only of `*.def` and `*.script` files without any internal module network.
- Macro modules can also be defined locally to a given network document path, called 'Local Macro Modules'. These are used in complex networks to encapsulate subnetworks as independent functional units with a defined interface to other network components. Such local macros often carry out an application specific function which would not be of common use for any other application, and are therefore not added to the common MeVisLab module database (that is they are not declared in / do not possess a `*.def` file).

Local macros are created and added with respect to the current network via the menu bar, **File** → **Create Local Macro** and **File** → **Add Local Macro**.



### Tip

However, as we will show in our example, local macros can also be promoted to global (normal) macros.

# Chapter 9. Developing a Macro Module for an Applicator

In the following sections, we will create a macro module based on the applicator we have built in the Open Inventor example chapter, adding fields and scripting for dynamic control of length and diameter of the applicator.

- [Section 9.1, “Creating a Basic Global Macro”](#)
- [Section 9.2, “Adding the Macro Parameters and Panel”](#)
- [Section 9.3, “Programming the Python Script”](#)
- [Section 9.4, “Addition: Shifting the Whole Tip”](#)

If you have not followed our tutorial, please open the `ApplicatorExample.mlab` demo (available via **Help** → **Welcome**) and start from there.

## 9.1. Creating a Basic Global Macro

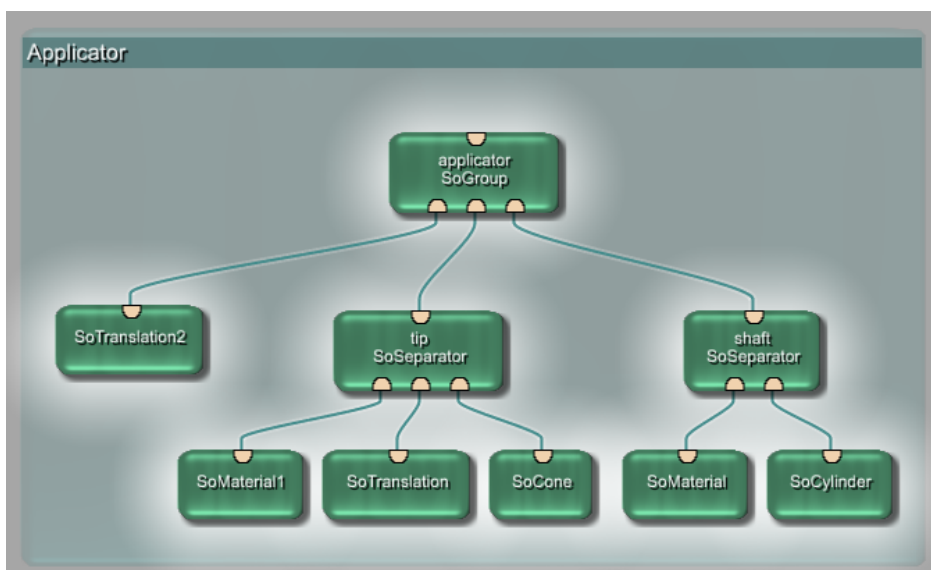
1. For a start, open a new network tab (**File** → **New** or a keyboard shortcut) and copy and paste the applicator modules (**Edit** → **Copy**, **Edit** → **Paste** or the respective keyboard shortcuts) to the new network.



### Tip

You can select the Applicator group with a double-click on its title bar and then press **SHIFT** and click the group title to deselect the group and keep only the modules selected for copying.

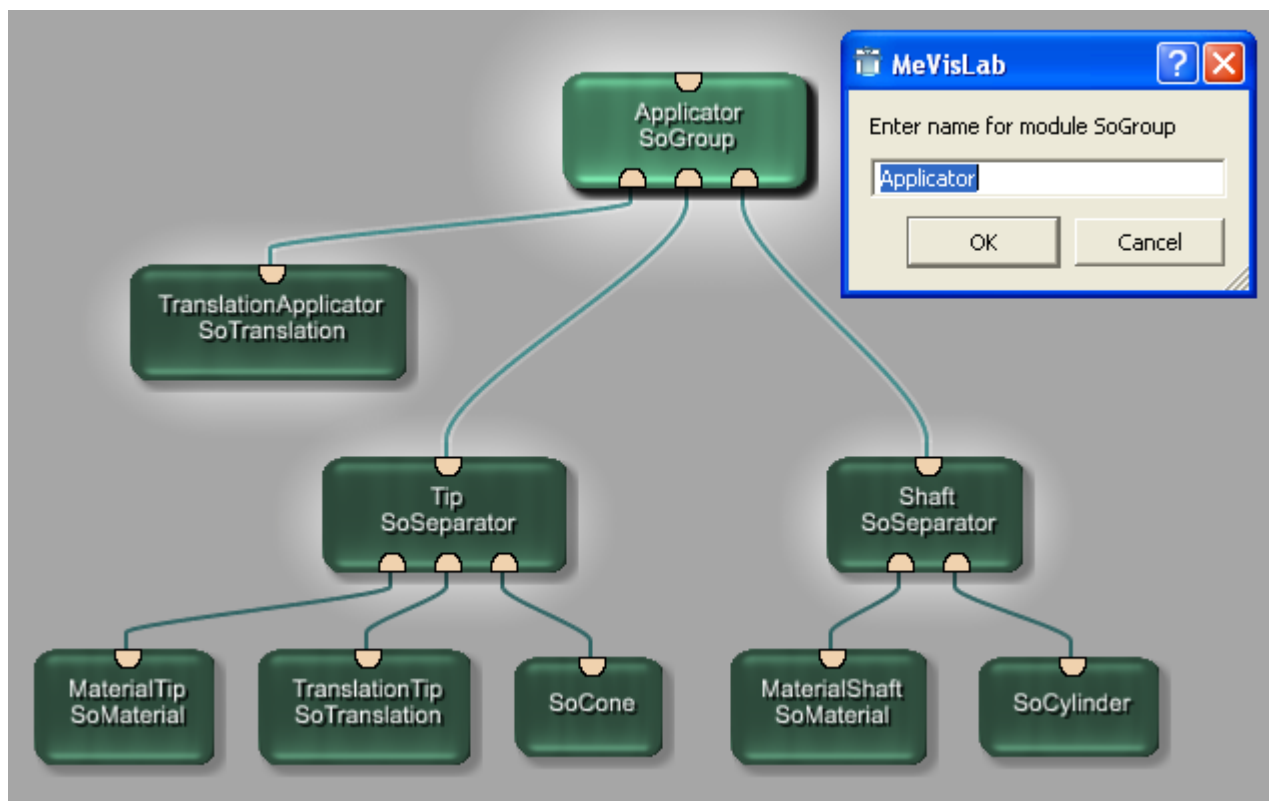
**Figure 9.1. Starting a new Macro from the Existing Applicator**



2. In the next step, clean the instance names of the modules — as they will be used for a new macro, there is no need to have names like “SoTranslation2”. Remove all numbers and write all module instance names starting with capital letters (if you want to) by right-clicking the module and selecting **Edit Instance Name** from the context menu.

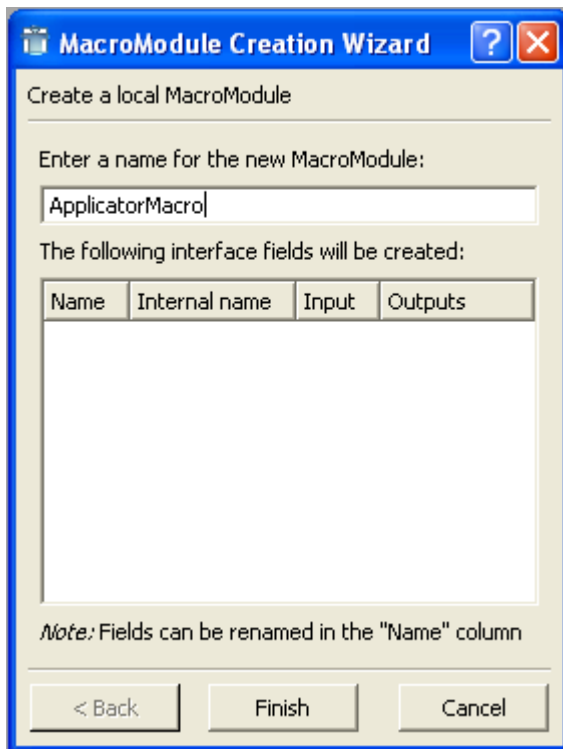


**Figure 9.2. Renaming Instance Names**



As we already have the modules for our macro, it is easiest to create a new local macro from them first. For this, select **File** → **Create Local Macro**. The local macro can be promoted to a global macro in the creation process.

**Figure 9.3. Creating a Local Macro**



When you promote the macro to a global macro, the Macro Module Wizard starts.

3. Enter the properties for your new module.

- **Name:**

Enter the module name `ApplicatorMacro` here. It has to be a unique name within the MeVisLab module database (including the SDK module database).

- **Author**

Enter your name or initials. The author entry is mandatory and will be used in module searches.

- **Comment**

Enter a short description for the module. The comment entry is mandatory.

- **Keywords**

The optional keywords should be the terms other users might search for, e.g. “applicator” in this case.

- **See Also**

The optional See Also entries should list other, related modules that might be of interest for a user.

- **Genre**

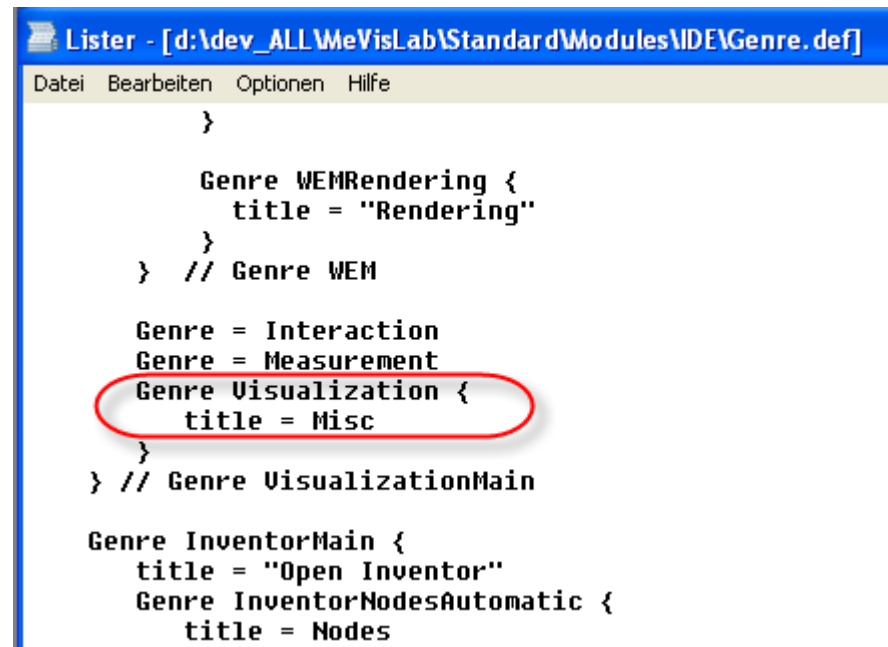
Enter the genre. Genre entries are mandatory; they defines the place of the module in the **Modules** menu and the **Module Browser**. For suggestions, check out similar modules in the database.



## Tip

To find a fitting genre, you might have a look at the `Genre.def` file in the Standard/IDE package. In our case, Visualization/Misc might be a good choice, which is (slightly confusing) the genre “Visualization” in the genre definition file.

**Figure 9.4. Selecting a Genre**



```
}  
  
Genre WEMRendering {  
  title = "Rendering"  
}  
} // Genre WEM  
  
Genre = Interaction  
Genre = Measurement  
Genre Visualization {  
  title = Misc  
}  
} // Genre VisualizationMain  
  
Genre InventorMain {  
  title = "Open Inventor"  
  Genre InventorNodesAutomatic {  
    title = Nodes  
  }  
}
```

The genres are not carved in stone but developed over time, so there might be more than one fitting choice for your module. You may even want to add a new genre in `Genre.def` or define an own user genre.

- **Add reference to example network:**

Each module should be completed by an example network to explain its function and usage in an exemplary application. Check to create an empty example network `ExampleModuleName.mlab` which may be edited later (optional).

- **Project:**

User defined modules are grouped in projects. Enter a new project name here: “ApplicatorMacro”. The module will be installed in the `Project Path` in the subdirectory `ProjectName`.

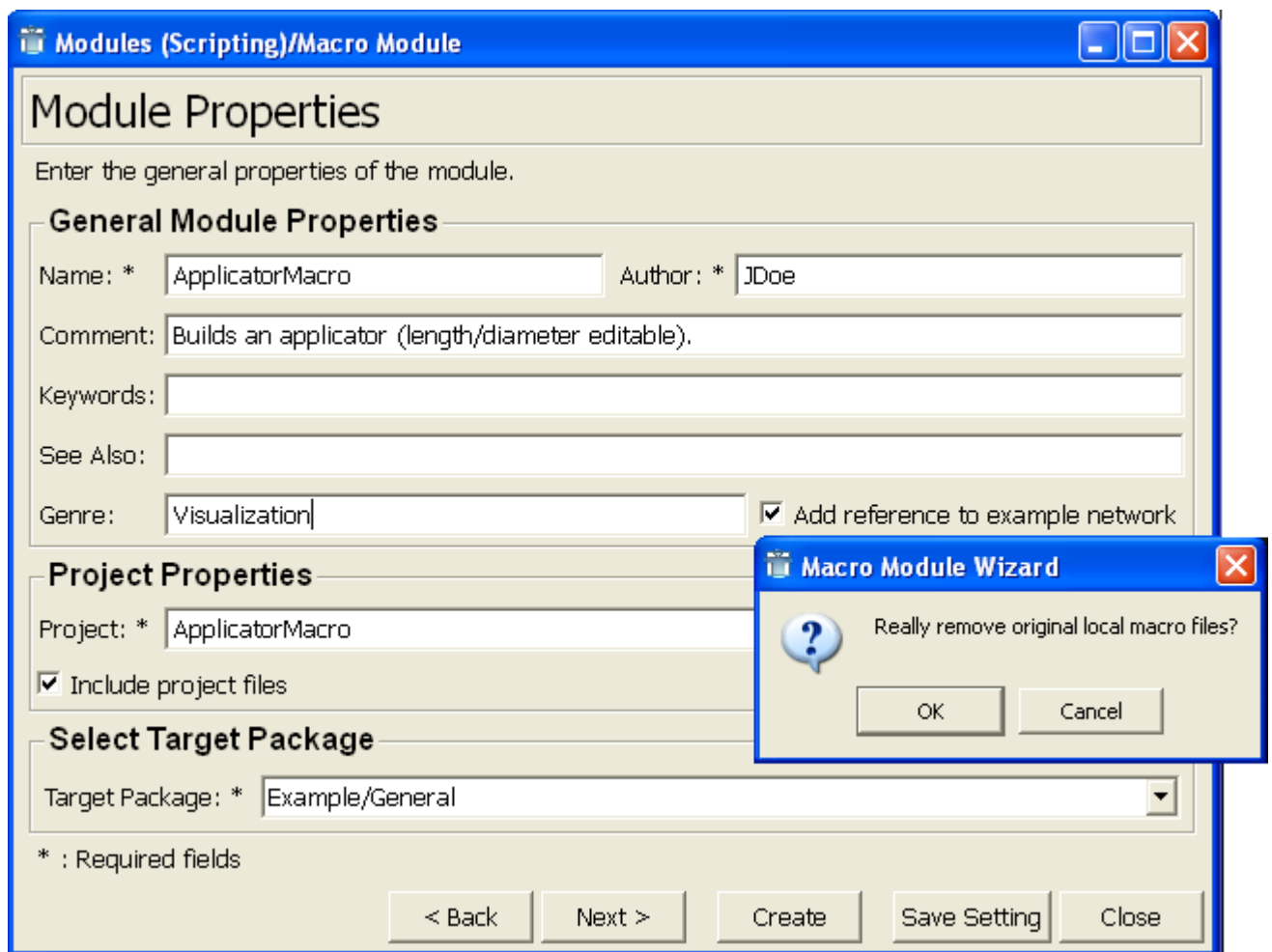
- **Target Package:**

Select a Target Package from the list, for this example “MyPackageGroup/Research” ??.

Click **Next**.

4. Click **Create**. You are asked whether the original local macro files should be removed. Accept with **OK**, because the local macro files are obsolete with the promotion to global.

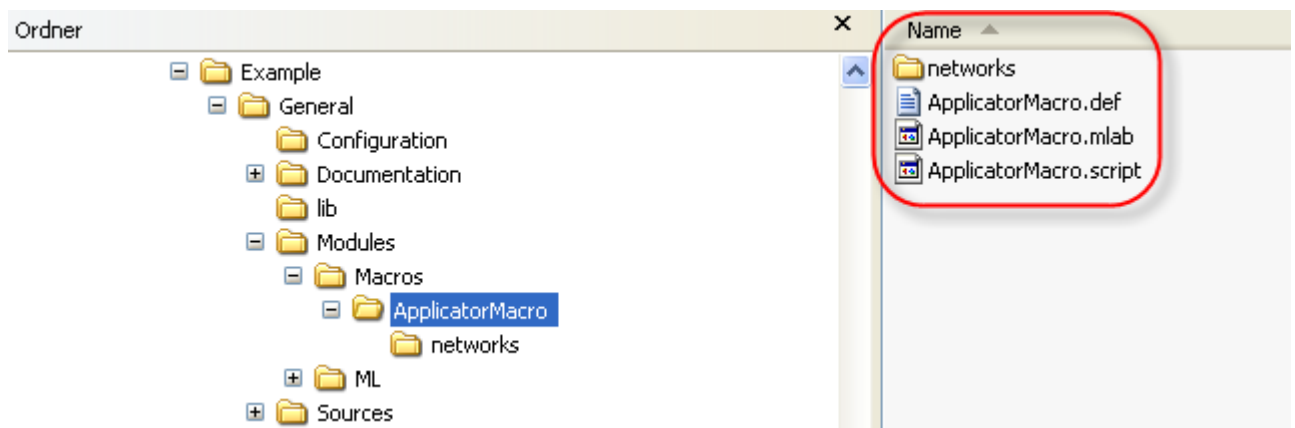
**Figure 9.5. Module Properties**



Now that the macro module and its necessary files are created, the file browser (depending on your system) will open and display the folders and files. In our example, we have a package group "Example" with the package "General" and in the folder Modules/Macros the new `ApplicatorMacro` with the files

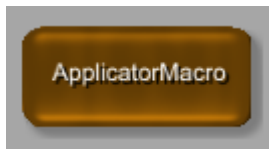
- `.def`: module definition file, for registering the module(s) to the MeVisLab module database.
- `.mlab`: network file which includes the modules and their settings.
- `.script`: MDL script file for the panel and from which other scripts (Python or JavaScript) may get called.

**Figure 9.6. File Browser with the New Macro Module Files**



On the workspace, the previously visible network is now displayed as a macro module.

**Figure 9.7. ApplicatorMacro as Macro Module**



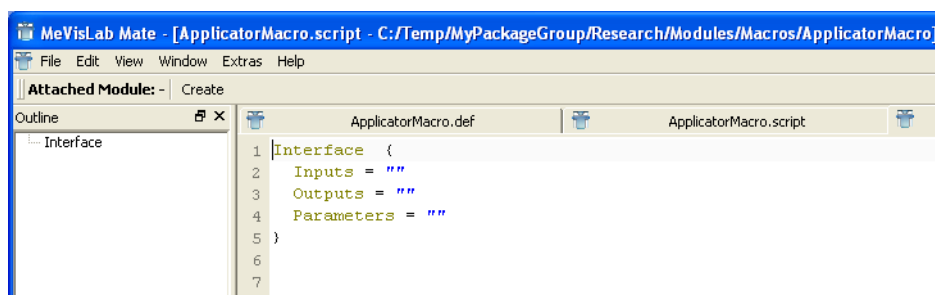
5. To display the internal network on a second tab, right-click the module and select **Show Internal Network** from the context menu. Alternatively, you can hold **Shift** and double-click the macro module.

## 9.2. Adding the Macro Parameters and Panel

So far, the macro module has no points of interaction. Therefore, the input/output, the parameters/fields and the scripting need to be added.

1. To edit the panel and its underlying scripting, right-click the `ApplicatorMacro` module and select **Related Files** → **ApplicatorMacro.script** to open the file in the in-built text editor Mate. Since we just defined this macro module, the script file is basically empty except for some placeholders.

**Figure 9.8. ApplicatorMacro.script in Mate**



### Tip

Mate comes with some special features like autocompletion, syntax highlighting, indentation, etc. for MDL, Python and JavaScript. For an extensive list, see the MeVisLab Reference Manual.

We want three sections in the `.script` file:

- a. **Interface:** defines the inputs and outputs of data connections for the macro. In our case, the macro has no inputs from other modules, but one output which is the Inventor scene.
- b. **Commands:** defines the scripting file to be executed upon the activity of defined fields.
- c. **Window:** defines the panel of the macro to set the parameters. In our case, length and diameter. This is an optional entry; if not defined, only the automatic panel is available.



## Note

The window section of the GUI could also be implemented in the `.def` file. If you want to implement an enhanced GUI and add more fields that only exist for scripting, use the `.script` file and reference that from your `.def` file. The advantage of splitting the GUI definition from the module announcement is a faster MeVisLab startup (because only the `.def` file is read). Further information on this subject can be found in the MDL Reference. .

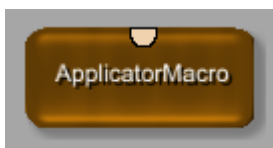
2. First we will define the interface. As no inputs are needed, keep this line as it is. For the output, we address the output of the `SoGroup` module named `ApplicatorMacro`. The following lines will result in an output field that will "deliver" the applicator.

```
Interface {  
  Inputs = ""  
  Outputs {  
    Field Scene { internalName = "Applicator.self" }  
  }  
  Parameters = ""  
}
```

Enter the lines in Mate and save the script file.

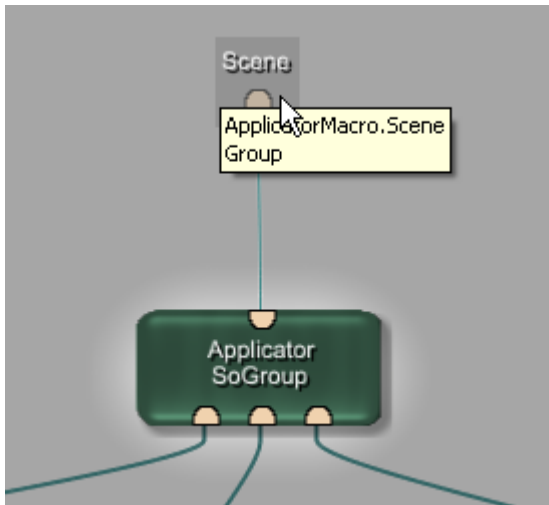
3. Then reload the module by right-clicking the macro module and selecting **Reload Definition** to apply the changes. The `ApplicatorMacro` module now shows an Open Inventor output connector.

**Figure 9.9. ApplicatorMacro Module with Output Connector**



The internal network of the macro shows the output placeholder. In the mouse-over, the output field name is displayed.

**Figure 9.10. Internal Network of the ApplicatorMacro Module**



4. As next step, we will define the parameters for our interface. In this example, we want to have two parameters:

- **Length**: this shall be the overall length of the applicator.
- **Diameter**: this shall be the diameter of the applicator.

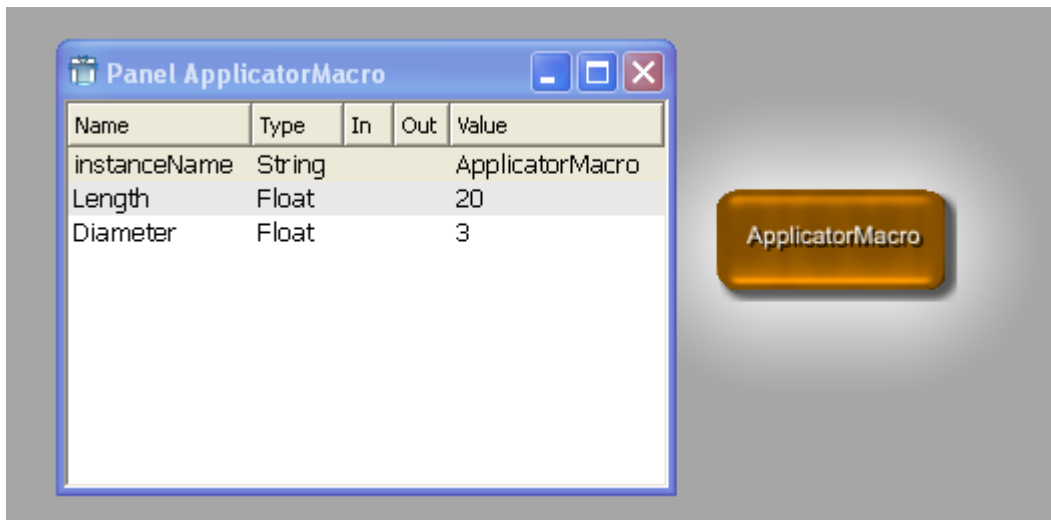
These two parameters need to be added to the `Interface` part of the script file. Besides setting the parameter type (`type`) and the default value (`value`), you can also add a minimum and a maximum value to limit the range to sensible values.

```
Interface {
  Inputs = ""
  Outputs {
    Field Scene { internalName = "Applicator.self" }
  }
  Parameters {
    Field Length {
      type = float
      value = 20
      min = 1
      max = 50
    }
    Field Diameter {
      type = float
      value = 3
      min = 0.1
      max = 10
    }
  }
}
```

Once again, save the script and reload the macro module.

5. Open the automatic panel, either by double-clicking the module, by holding **ALT** and double-clicking the module, or by right-clicking the module and selecting **Show Window** → **Automatic Panel** from the context menu. The new parameters are visible in the automatic panel. They can also be edited there by clicking on each value field and editing the value.

**Figure 9.11. Automatic Panel of the ApplicatorMacro Module**



In principle, this would be enough to enter the values. However, usually a more user-friendly panel should be offered. In the panel, values can be sorted by correlation or importance and distributed on various tabs. It is also possible to leave rarely used parameters out of the panel to make it slimmer; as the automatic panel of a module is always available, the user can always view and edit all parameters there.

6. To create a panel for the two parameters, the new section `Window` is added at the end of the script file. Besides defining the fields in `Category`, you can also add a step value which will regulate how large the step is when moving through the values with the spin box arrows or the mouse wheel (with the mouse cursor over the field). As the diameter is smaller than the length, it makes sense to set a smaller step size here.

```
Interface {
  Inputs = ""
  Outputs {
    Field Scene { internalName = "Applicator.self" }
  }
  Parameters {
    Field Length {
      type = float
      value = 20
      min = 1
      max = 50
    }
    Field Diameter {
      type = float
      value = 3
      min = 0.1
      max = 10
    }
  }
}

Commands {
}

Window {
  Category {
    Field Length { step = 1 }
  }
}
```

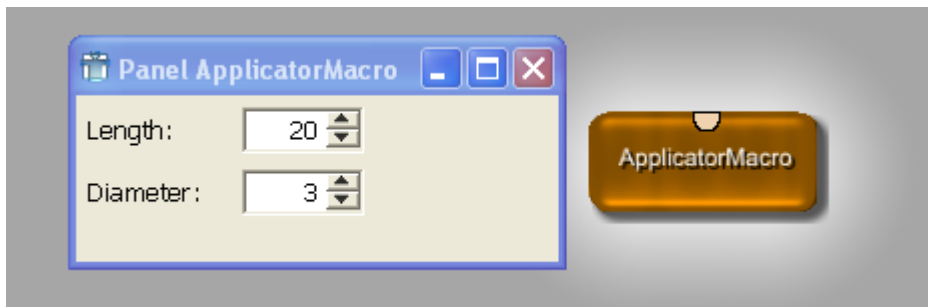


```
Field Diameter { step = 0.1 }  
}  
}
```

Save the script and reload the macro module.

7. Now open the panel, either by double-clicking the module (because the panel is the new default panel) or by right-clicking the module and selecting **Show Window** → **Panel** from the context menu. The new parameters are visible in the panel and can be edited manually (or by using the spin arrows or the mouse wheel).

**Figure 9.12. Panel of the ApplicatorMacro Module**



All parameters are defined and the panel is ready for entering values — however, we still do not have any interaction. So the last section `Command` needs to be added, in which the respective scripting file (in our case, a Python file) and the fields this scripting file should “look at” need to be entered

The source will be a local file which we will add manually, with the name `ApplicatorMacro.py` by convention.

To relate to the scripting, we need two field listeners that listen to fields and call the script command given in the `command` tag when the field changes. The functions `AdjustLength` and `AdjustDiameter` used in the code do not exist yet but will be defined by us in the Python file.

```
Interface {  
  Inputs = ""  
  Outputs {  
    Field Scene { internalName = "Applicator.self" }  
  }  
  Parameters {  
    Field Length {  
      type = float  
      value = 20  
      min = 1  
      max = 50  
    }  
    Field Diameter {  
      type = float  
      value = 3  
      min = 0.1  
      max = 10  
    }  
  }  
}  
  
Commands {  
  source = $(LOCAL)/ApplicatorMacro.py  
  
  FieldListener Length { command = AdjustLength }  
  FieldListener Diameter { command = AdjustDiameter }  
}
```

```
Window {
  Category {
    Field Length { step = 1 }
    Field Diameter { step = 0.1 }
  }
}
```

8. Save the script and reload the macro module. If the Python file or the scripting commands do not exist yet, errors messages will appear in the Debug Output of Mate. Do not be concerned — we will add everything we need for real interactivity in the next section.



### Tip

Panels can have a more complex design; for the possibilities, see the MDL Reference and the MDL panel example modules (search for modules starting with “Test...”).

## 9.3. Programming the Python Script

1. If not yet existing, create the Python file. For this, select **File** → **New** in the Mate menu bar and save the new file as `ApplicatorMacro.py` in the same folder as the other module files.
2. For the header of the file, take a look at other existing macro modules. What we need, besides the comment lines in `#`, is a line for importing the MeVis Python modules.

```
# -----
## This file implements scripting functions for the ApplicatorMacro module
#
# \file    ApplicatorMacro.py
# \author  JDoe
# \date    01/2009
#
# -----

# MeVis module import
from mevis import *
```

3. Then we need to add two functions, one for each scripting command

```
def AdjustLength():
    return

def AdjustDiameter():
    return
```



### Note

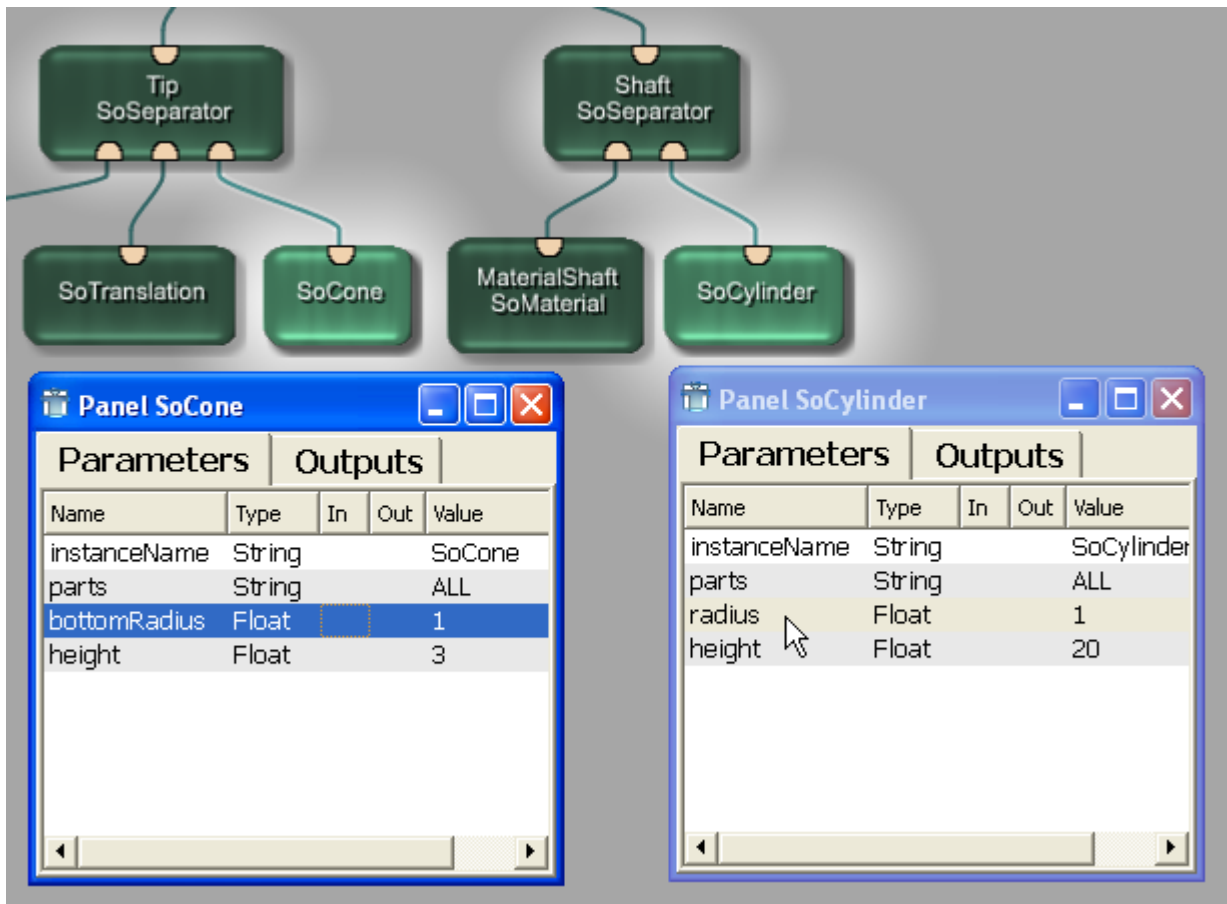
In Python, block structure is defined by indentation. Therefore it is important to indent the lines as shown in the code examples. In the Mate editor, this will happen automatically.

4. Let us have a look at the diameter adjustment. The diameter is given by the `Diameter` field. This is written as follows:

```
def AdjustDiameter():
    diameter = ctx.field ("Diameter").value
    return
```

To have both an effect on shaft and tip likewise, the diameter parameter of both must be set to the value of the `Diameter` field. A look at the automatic panels of `SoCone` and `SoCylinder` shows that both modules offer a radius parameter.

**Figure 9.13. Parameters for Diameter Setting**



These radius parameters need to be set to diameter:

```
ctx.field("SoCone.bottomRadius").value = diameter
ctx.field("SoCylinder.radius").value = diameter
```

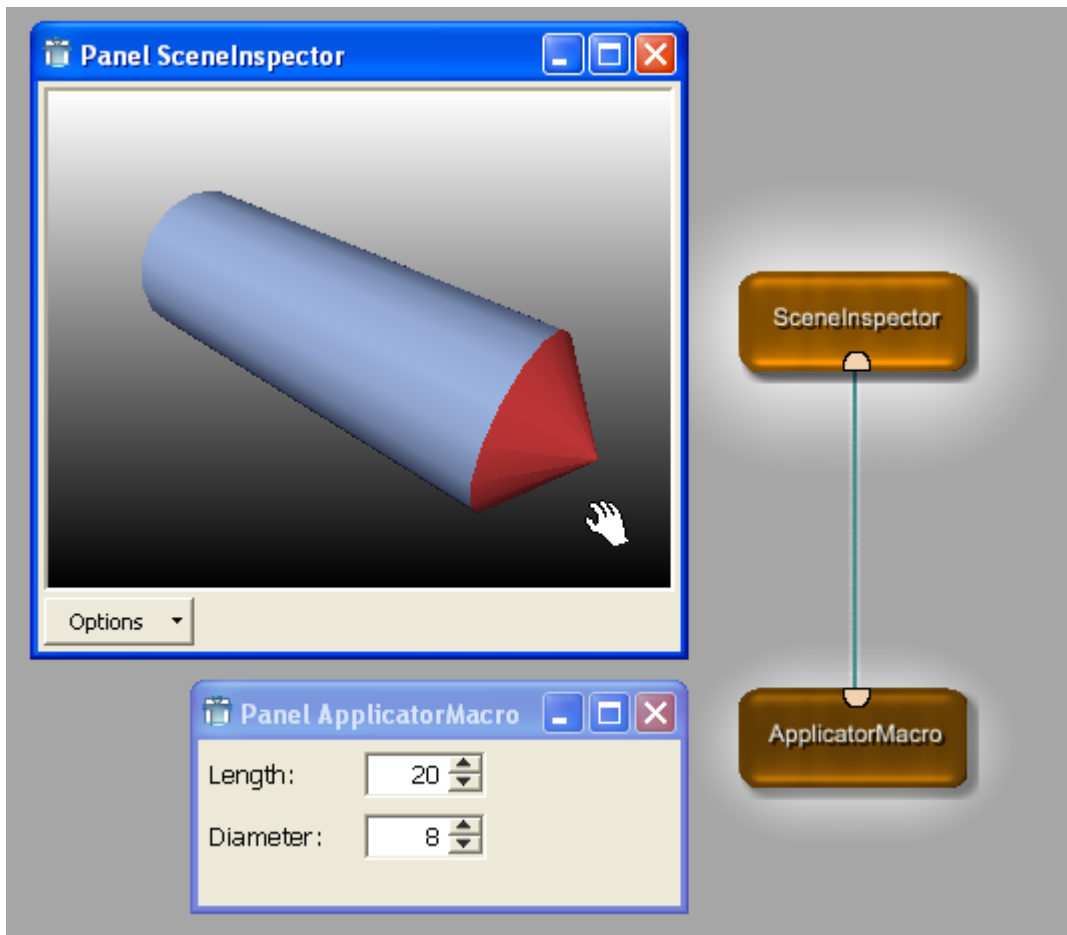
As the radius is half the diameter, a correcting factor of 0.5 has to be added to the diameter equation.

```
def AdjustDiameter():
    diameter = ctx.field("Diameter").value * 0.5

    ctx.field("SoCone.bottomRadius").value = diameter
    ctx.field("SoCylinder.radius").value = diameter
    return
```

5. To test if the diameter adjusting works, add a `SceneInspector` module to the network and connect its input to the output of your `ApplicatorMacro` module. Double-click the `SceneInspector` to open its viewer. When you change the diameter setting of the macro, the diameter of the applicator is changed accordingly.

**Figure 9.14. Changing the Diameter of the Applicator**



6. Adjusting the length is a bit more complicated. The length change should have the following effects:

- The `Length` parameter gives the overall length.
- Only the shaft should be extended, not the tip.
- The adjustment should be done in a way that the point of the tip is not translated, that is that the tip points to the same position as before. Therefore, we need to increase the applicator length in the direction away from the tip.

We can define an overall length, a tip length and a shaft length. They can be calculated as follows:

```
def AdjustLength():
    overallLength = ctx.field("Length").value
    tipLength      = ctx.field("SoCone.height").value

    shaftLength    = overallLength - tipLength
    return
```

The original translation factor for the tip (which is the relevant factor) was given by half the shaft length ("10") plus half the tip length ("1.5"). This can be written in a general way.

```
tipTranslation = shaftLength*0.5 + tipLength*0.5
```

The `shaftLength` defines the height of the `SoCylinder` cone to

```
ctx.field("SoCylinder.height").value = shaftLength
```

The resulting code lines for the length adjustment look as follows:

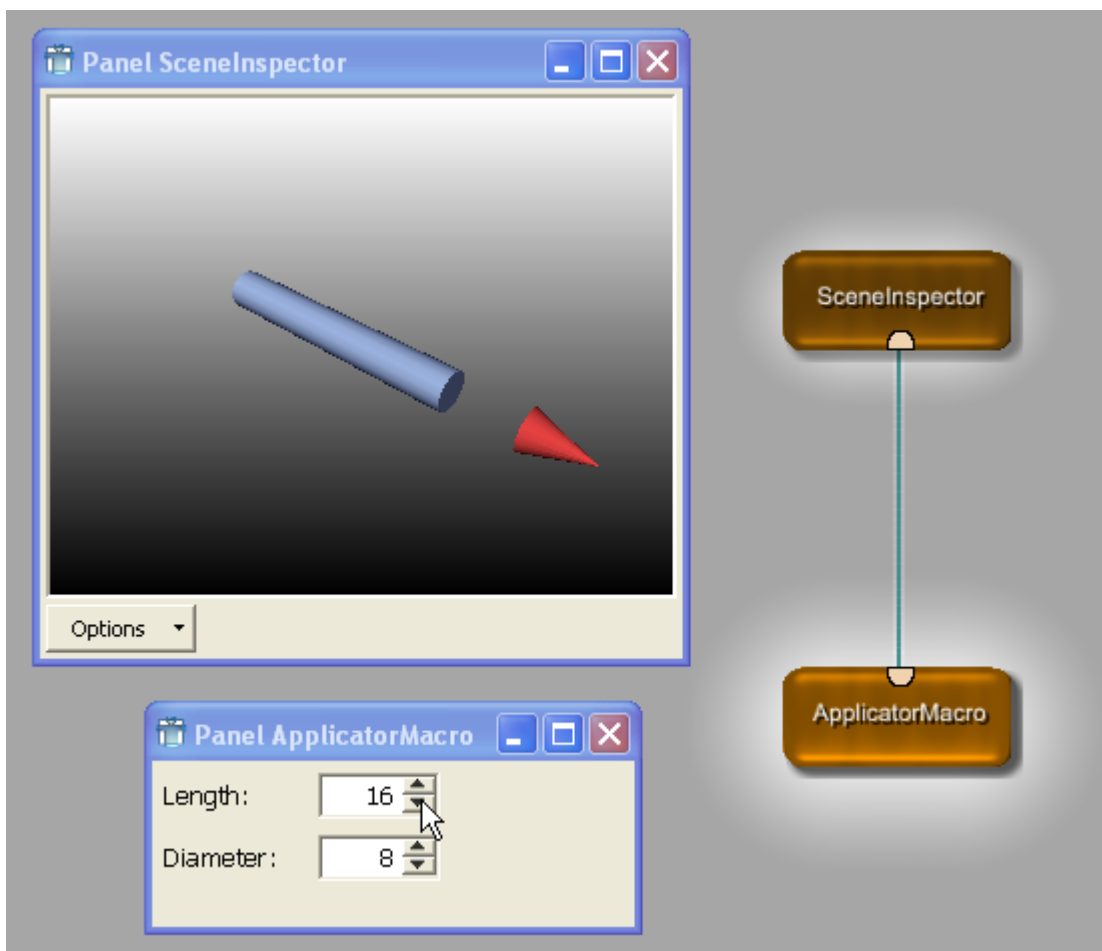
```
def AdjustLength():
    overallLength = ctx.field("Length").value
    tipLength      = ctx.field("SoCone.height").value

    shaftLength    = overallLength - tipLength
    tipTranslation = shaftLength*0.5 + tipLength*0.5

    ctx.field("SoCylinder.height").value = shaftLength
    return
```

Add this code to the Python script, save, and reload the definition. A test shows a funny effect: the shaft length is changed independently of the tip.

**Figure 9.15. Strange Behavior of the ApplicatorMacro**



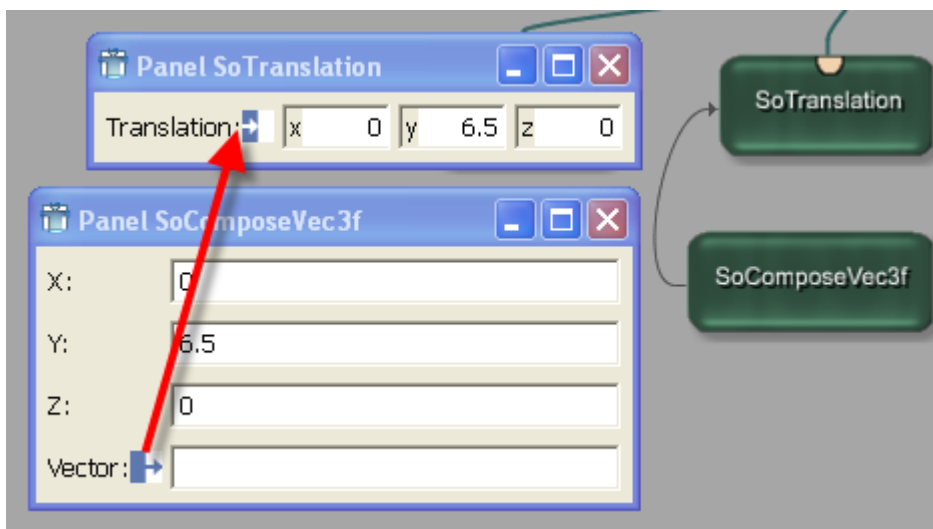
This is due to not having connected the calculated `tipTranslation` with the `TranslationTip` module yet.

7. To solve this problem, add the `SoComposeVec3f` module to the internal network of the macro and assign to its translation in y direction the calculated value `tipTranslation`.

```
ctx.field("SoComposeVec3f.y").value = tipTranslation
```

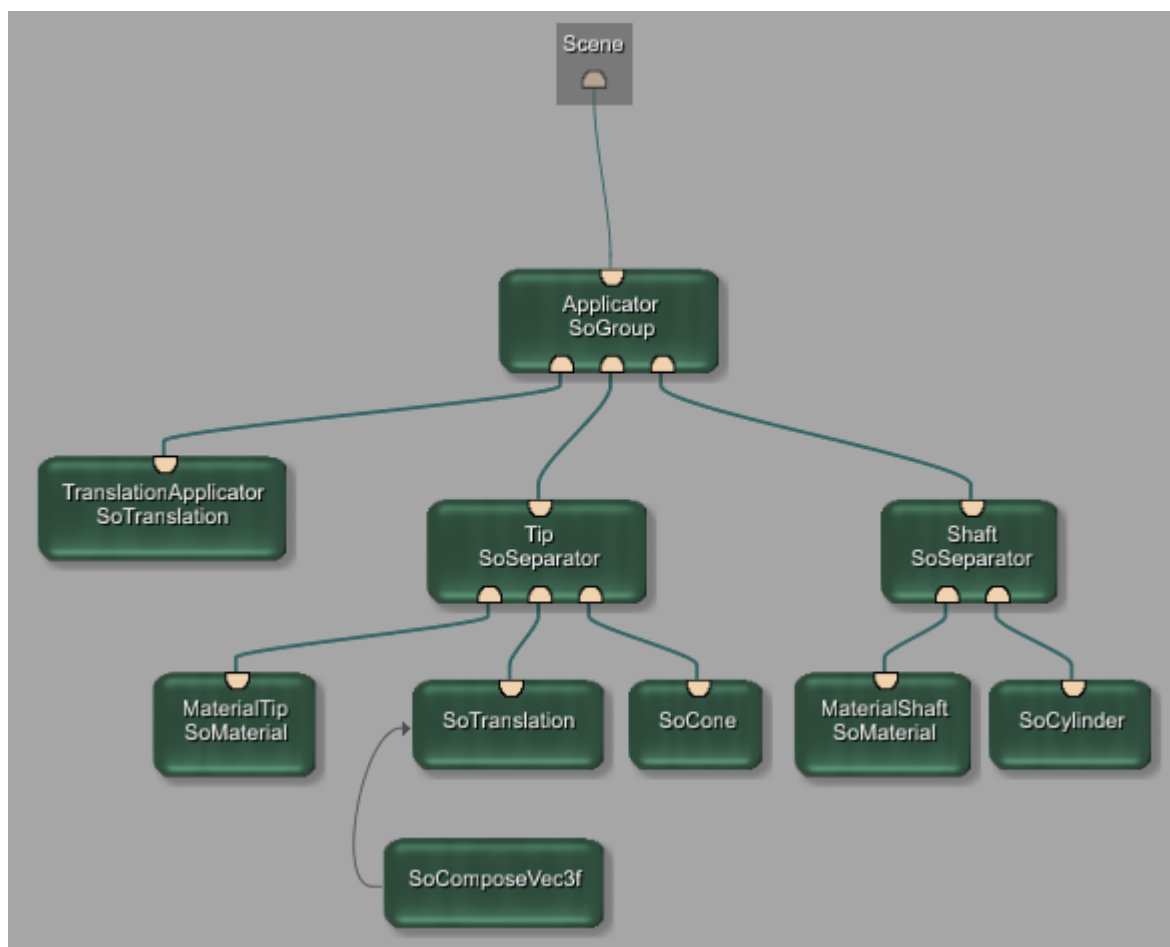
8. In a last step, this translation needs to be connected to the tip's `SoTranslation` module via a parameter connection in the network.

**Figure 9.16. Adding the Correct Tip Translation**



Here the network and complete Python script of the ApplicatorMacro example:

**Figure 9.17. Complete ApplicatorMacro**



```
# -----  
## This file implements scripting functions for the LocalFileName module  
#
```

```
# \file      ApplicatorMacro.py
# \author    JDoe
# \date      01/2009
#
# -----

# MeVis module import
from mevis import *

def AdjustDiameter():
    diameter = ctx.field("Diameter").value * 0.5

    ctx.field("SoCone.bottomRadius").value = diameter
    ctx.field("SoCylinder.radius") .value = diameter
    return

def AdjustLength():
    overallLength = ctx.field("Length").value
    tipLength      = ctx.field("SoCone.height").value

    shaftLength    = overallLength - tipLength
    tipTranslation = shaftLength*0.5 + tipLength*0.5

    ctx.field("SoCylinder.height").value = shaftLength
    ctx.field("SoComposeVec3f.y") .value = tipTranslation
    return
```

## 9.4. Addition: Shifting the Whole Tip

In the example above, the change in length will be translated into an overall change with the center of rotation as overall center. However, it might be preferable to keep the tip in place and change the length of the shaft into the other direction.

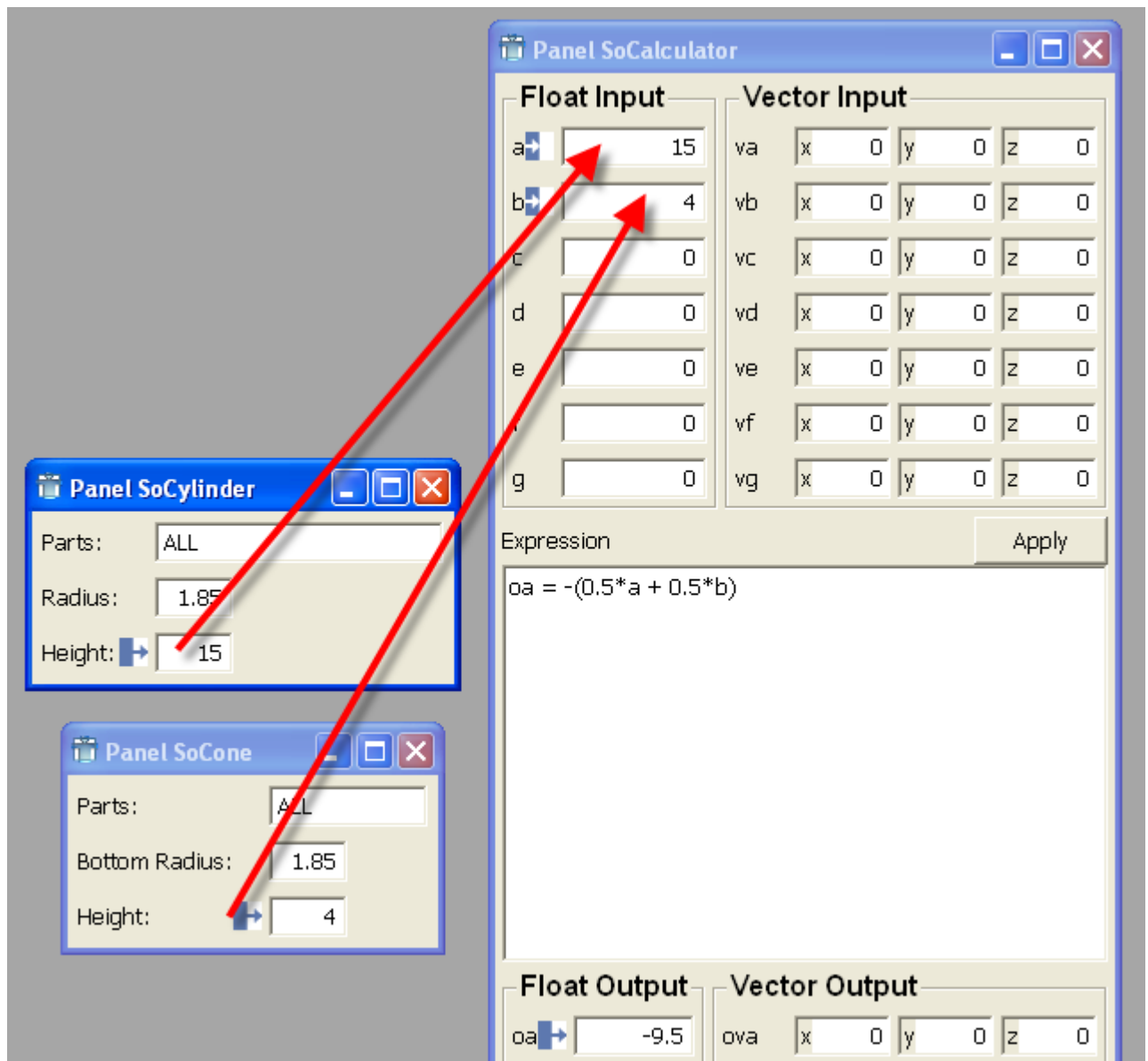
Basically, this is the same problem as the length calculation we made in the Python script. However, instead of calculating it in the macro scripting, we can also use a module for the calculation.

For this, the following modules need to be added:

- `SoCalculator`: For calculating the length of the shaft.
- `SoComposeVec3f`: For applying the translation of the float value to the vector of the overall translation in `TranslationApplicator`.

The `SoCalculator` module offers input and output of floating values and vectors.

**Figure 9.18. Feeding the SoCalculator Module**



Some important points:

- In the **Expression** field, mathematic formulas can be entered; the name of the input values and the name of the output have to be given.
- More than one expression can be entered. For that, end each line with a semicolon ;
- For the expression to be calculated, you need to click **Apply**.

For calculating the translation from the input values of cone and shaft height, use the **SoCalculator** module and set up parameter connections

1. Connect **SoCylinder.height** to **SoCalculator.a**
2. Connect **SoCone.height** to **SoCalculator.b**
3. Enter the calculation:  $oa = - (0.5*a + 0.5*b)$  (a negative sign needs to be added; otherwise, the end of the applicator is fixed and the tip side grows).



To apply the new translation, we need another `SoComposeVec3f` module. It allows for converting the float value `y` into a vector translation in `y` direction. For this, it needs to receive the output of `SoCalculator` and deliver the input for the `SoTranslation` module.

1. Connect `SoCalculator.oa` to `SoComposeVec3f1.y`
2. Connect `SoComposeVec3f1.vector` to `SoTranslation.translation`

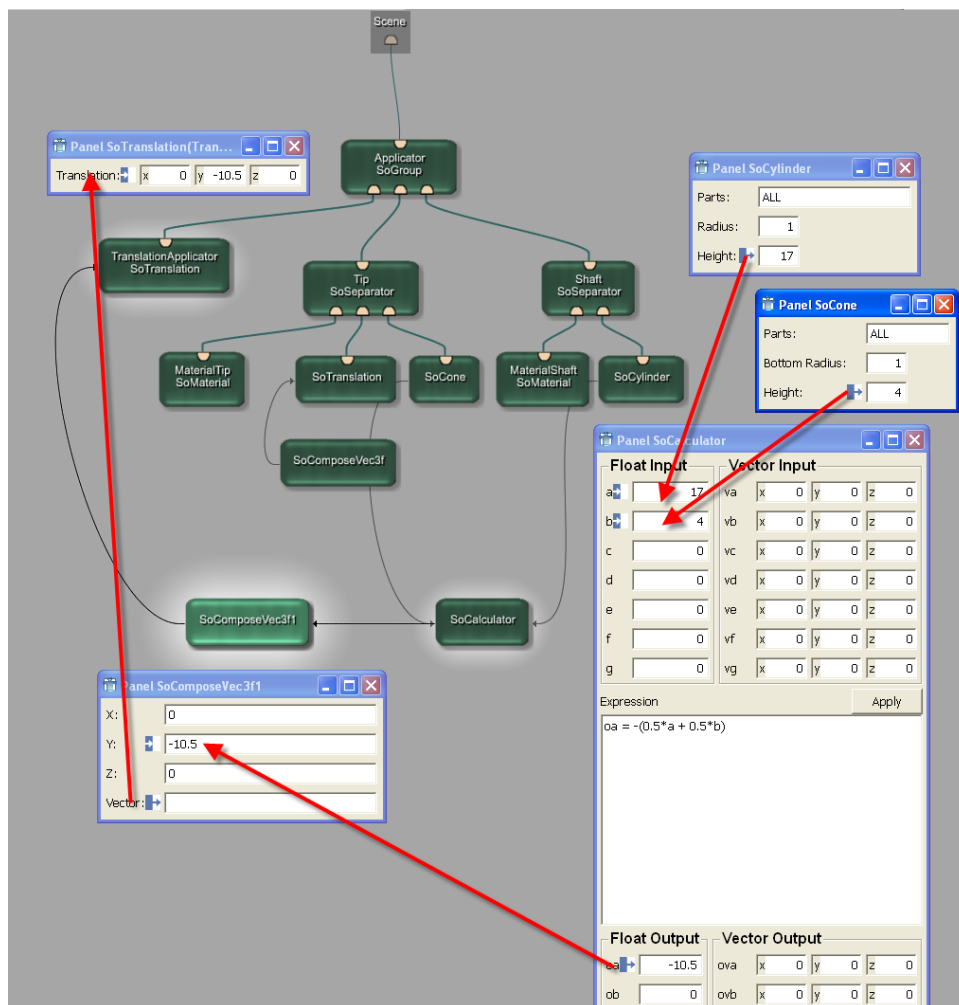


### Tip

You can find the names of the connected parameters by right-clicking the parameter connections. For an overview of all parameter connections in a network, use the **Parameter Connections Inspector** View.

The resulting macro network looks as follows:

**Figure 9.19. Improved Applicator Macro Module**



When to choose calculating values in scripts and when via modules? This is not an easy question.

- The advantage of the script is that it is easily changed and extended. This might be harder with modules
- The advantage of the modules is that the connections between modules are visible as parameter connections (which can be changed and removed).

In the end, it comes down to your current network and your design decisions which way to choose. Or you might combine them, like we did in our `ApplicatorMacro` network.

What else could you do now? You could, for example, make sure that the shaft length cannot be shorter than the tip length (which looks strange in the Open Inventor scene). You could also make the colors parametrizable, or add new features for the applicator.

This is the end of this example. The full network is delivered as example (`$(InstallDir)Packages/MeVisLab/Standard/Modules/Examples/GettingStarted/ApplicatorMacro`), so feel free to check it out and play around with it.

---

# Chapter 10. Excursion: Image Processing in ML

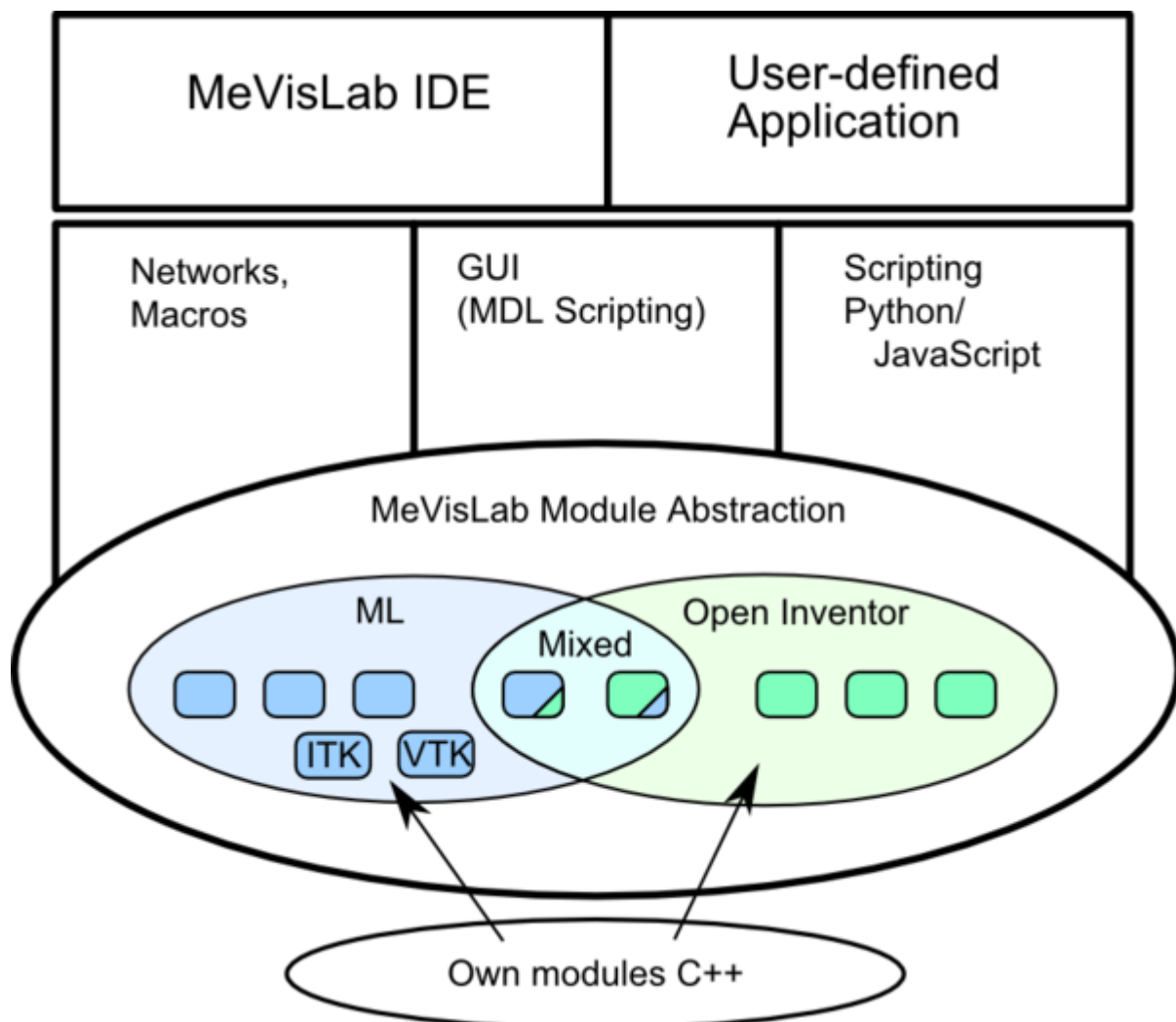
## 10.1. Some Advanced Information on Image Processing

In this chapter you will find a brief survey of some more advanced image processing concepts used in MeVisLab. Many of them are also discussed in the MLGuide, chapter 5 “Image Processing Concepts”. Please refer to this document for further information.

## 10.2. Structure of MeVisLab

In the following figure, the basic structure MeVisLab is shown:

**Figure 10.1. MeVisLab Structure**



MeVisLab is based on C++ objects called modules which either belong to the ML type system developed at MeVis or to the Open Inventor type system from SGI. Both module types offer a generic parameter field system for parametrization and change notification. Open Inventor modules together form a scene

graph for interaction and rendering in OpenGL, while the ML modules can be connected to form an image processing pipeline.

Image processing in the ML is demand-driven (in that only the required parts of an image output are calculated) and tile-based (this is used for caching of results). As an additional benefit, many classes from the ITK and VTK libraries are provided in the ML type system through code-generated wrapper modules.

Mixed modules belong to either system but can take input from the other system, thereby serving as a bridge between systems.

MeVisLab unifies these two module systems with another internal layer that abstracts away the differences between these systems. Stacked upon that layer is

- a system to turn whole module networks into new macro modules with an interface of their own. Macro modules may be built upon other macro modules.
- a GUI system where the elements are generated from a hierarchical description file, automatically providing access to the parameter fields of the modules if desired.
- an interface to the scripting languages Python and JavaScript with full access to the modules and GUI widgets, including the ability to generate new modules or widgets.

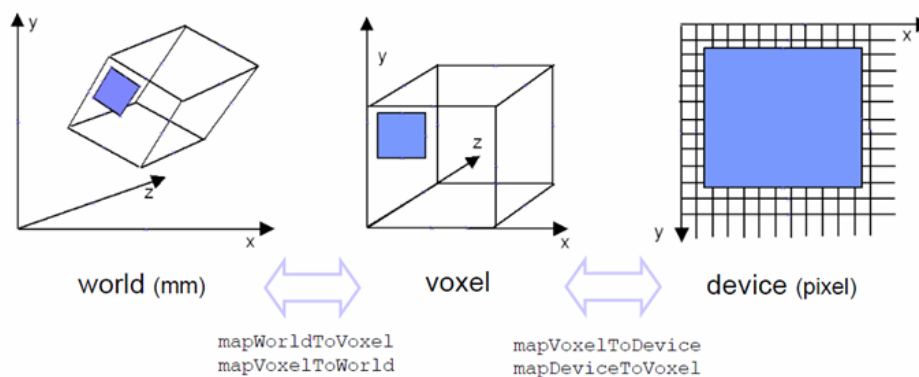
Based on these functionality one can build, test and evaluate own applications with the integrated development environment and — with the proper license — generate own installers with standalone applications.

## 10.3. Coordinate Systems

In MeVisLab, three coordinate systems exist next to each other:

- World coordinates
- Voxel coordinates
- Device coordinates

**Figure 10.2. Coordinate Systems**



The blue rectangle shows the same region in the three coordinate systems.

World coordinates are:

- Global: Combine several objects in a view
- Absolute: Measure distances and angles

- Isotropic: All directions are equivalent
- Orthogonal: Coordinate axes are orthogonal to each other

Voxel coordinates are:

- Relative to an image
- Dependent on voxel spacing
- Continuous from  $[0..x, 0..y, 0..z]$ , voxel center at 0.5
- Often non-isotropic, sometimes non-orthogonal
- Direct relation to voxel location in memory

Device coordinates are:

- 2D coordinates in OpenGL viewport
- Measured in pixel
- Have their origin (0,0) in the top left corner of the device (with x-coordinates increasing to the right and y-coordinates increasing downwards)

## 10.4. Affine Transformations

For mapping e.g. world to voxel coordinates, or device to world coordinates, affine transformations have to be applied. This is done with homogeneous coordinates:

- Extend the (x,y,z) triple by an artificial coordinate with a fixed value 1.
- Affine transforms can then be represented by a single matrix multiplication.

Why not a 3x3 matrix? Two reasons:

1. One cannot construct a 3x3 matrix that will translate the point (0,0,0). The zeroes in the coordinate vector cancel out all the coefficients.
2. Transformations could not be combined by multiplying the matrices.

Affine transformations have these elementary transforms:

- Translation (moves an object along a direction vector)
- Rotation (rotates the object around an axis vector)
- Scaling (shrinks/grows the object size)
- Shearing (deforms the object; rare in medical image data)

### Figure 10.3. Matrix Multiplication

$$\begin{pmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{pmatrix} = \begin{pmatrix} & t_x \\ M & t_y \\ & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 1 \end{pmatrix}$$



### Tip

Look at the example [Chapter 5, Defining a Region of Interest \(ROI\)](#) for the module `WorldToVoxel` in action.

The voxel coordinate system is a continuous coordinate system. Voxel boundaries are at integer values, voxel centers are 0.5 off. To transform integer voxel indices to voxel centers in world coordinates, either add the value “0.5” to voxel indices or check the option **Integer Voxel Coordinates** in the modules `WorldVoxelConvert`, `SoMLTransform`, and others.

#### Common pitfalls

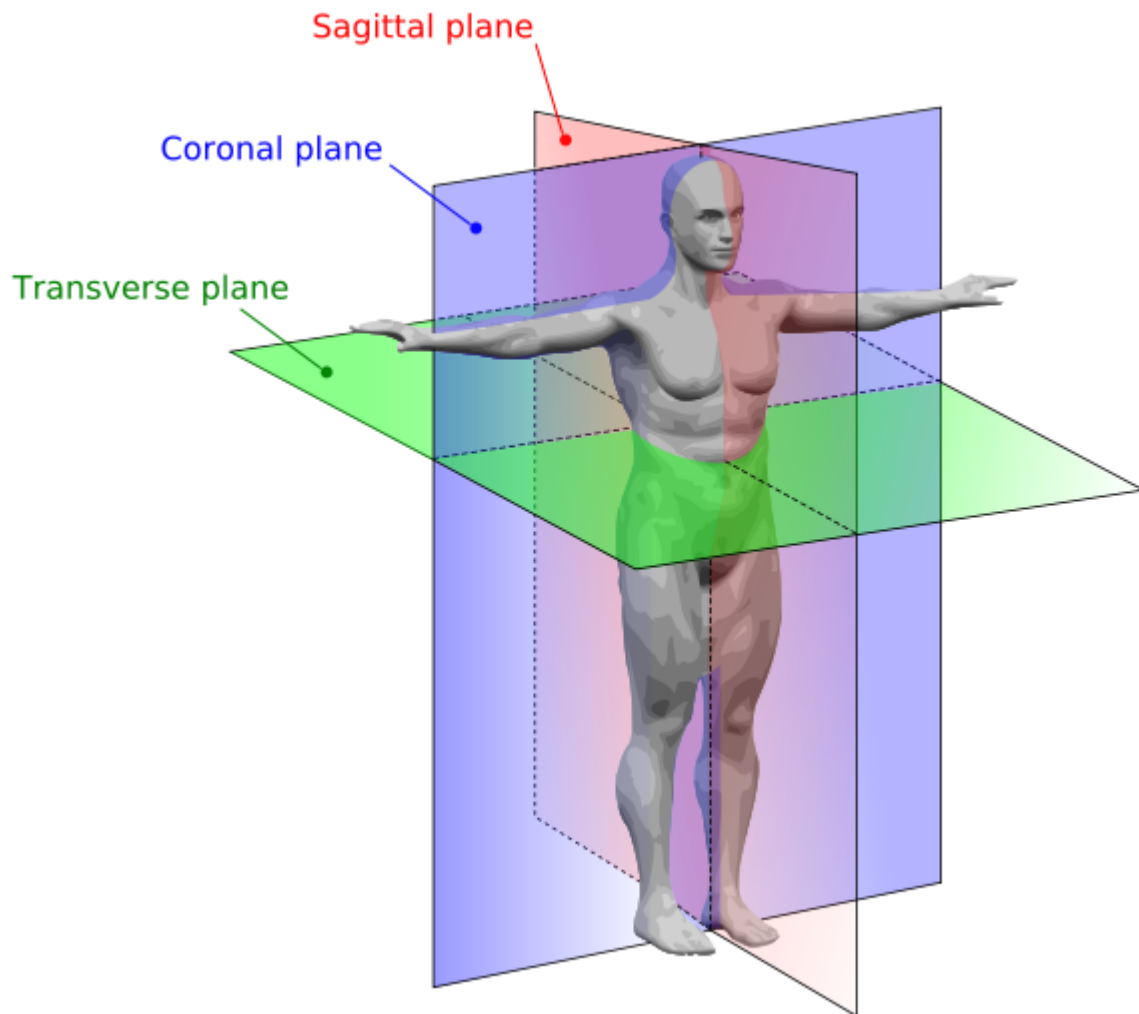
- Computing the voxel volume: `getVoxelSize()` returns voxel spacing in x, y and z. The product of these values is not the voxel volume if the voxel-to-world-matrix is not orthogonal. Solution: Use the absolute value of the matrix determinant instead.
- Inventor using row vector conventions: ML and MeVisLab use the widespread column vector conventions, that is vectors are written as columns and matrices are applied by left-multiplication. Open Inventor, in contrast, uses row vector conventions, that is vectors are written as rows and matrices are applied by right-multiplication. Solution: Use the matrix transposition to convert a matrix from one convention to the other.

## 10.5. DICOM Data and Coordinates

A mixed type are DICOM "coordinates". They are mostly world coordinates but refer to the patient axes.

- Based on the patient's main body axes (axial/transverse, coronal, sagittal)
- Measured as 1 coordinate unit = 1 millimeter
- Right-handed
- Not standardized regarding their origin

**Figure 10.4. World Coordinates in Context of the Human Body**



The DICOM (Digital Imaging and Communications in Medicine) standard is a data format that groups information into data sets. This way, the image data is always kept together with all meta information like patient ID, study time, series time, acquisition data etc. The image slice itself is essentially just another tag with pixel information.

DICOM tags have unique numbers, encoded as 2x4 numbers in hexadecimal notation (0000,0000). The first four numbers are the data group, the second four numbers the data set/tag.



### Note

Although DICOM is a standard, often the data that is received / recorded does not follow the standard. Wrongly used tags or missing mandatory tags may cause problems in data processing.

Some typical modules for DICOM handling:

- With `DicomImport` you import DICOM files and convert them into a 4D-TIFF image and a DICOM header file for the use in MeVisLab.
- In addition, `DicomImport` offers features for sorting; click the help button for an overview of possible options.
- You can view the image-wide DICOM tags with the module `DicomTagViewer`.

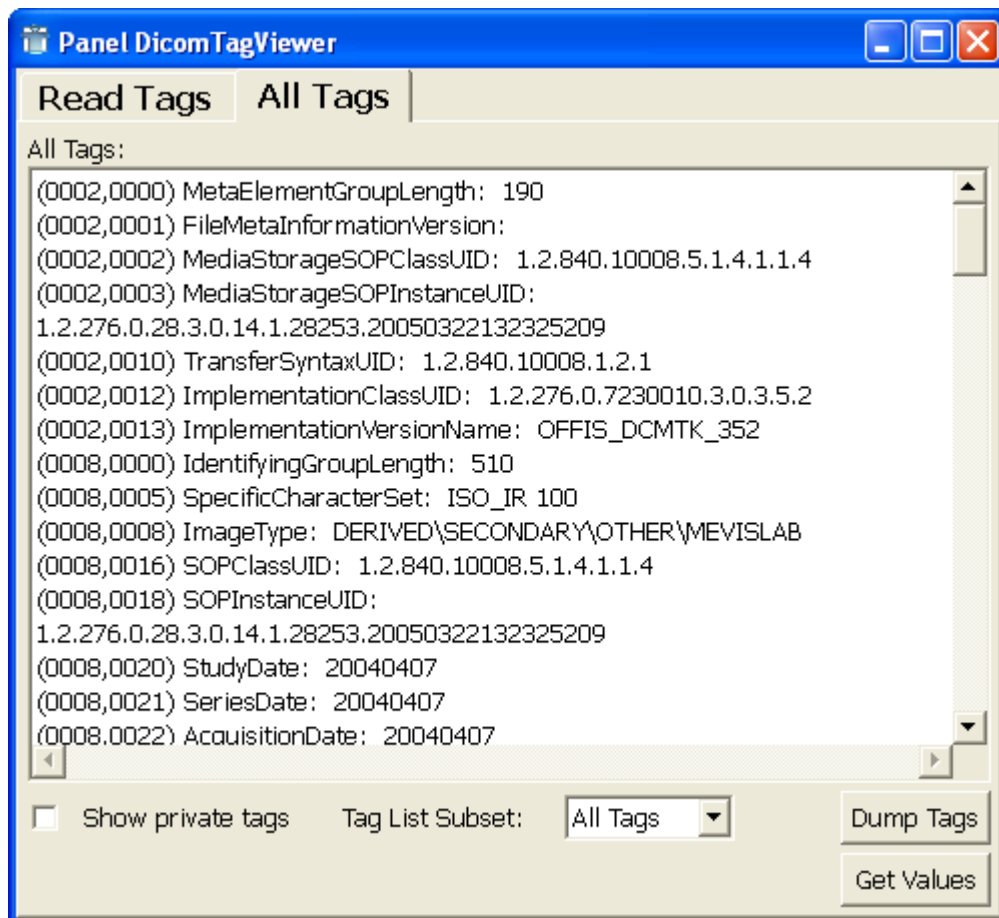
- You can view and cut out frame-specific tags with the module `DicomFrameSelect`.
- You can modify DICOM tags with the module `DicomTagModify`.
- You can also create a new DICOM header for an image file with the `ImageSave` module, tab **Options**, **Save DICOM header file only**.



### Tip

For handling and manipulating DICOM data, the DICOM toolkit “DCMTK” (DICOM@offis) is recommended. Parts of this toolkit are also used in MeVisLab.

**Figure 10.5. The DICOM Tag Viewer**



## 10.6. Coordinate Systems in the MeVisLab GUI

You can find information about the voxel and world matrix in the image properties on the **Output Inspector** View.

The easiest (ideal) image is when the world and the voxel matrix correspond, so that one voxel is one world unit, and the world matrix is coronal (not tilted in any way). In case of an image taken in the sagittal position, voxel sizes may be different and the world matrix may be tilted.



Figure 10.6. Image Properties for an Ideal Image

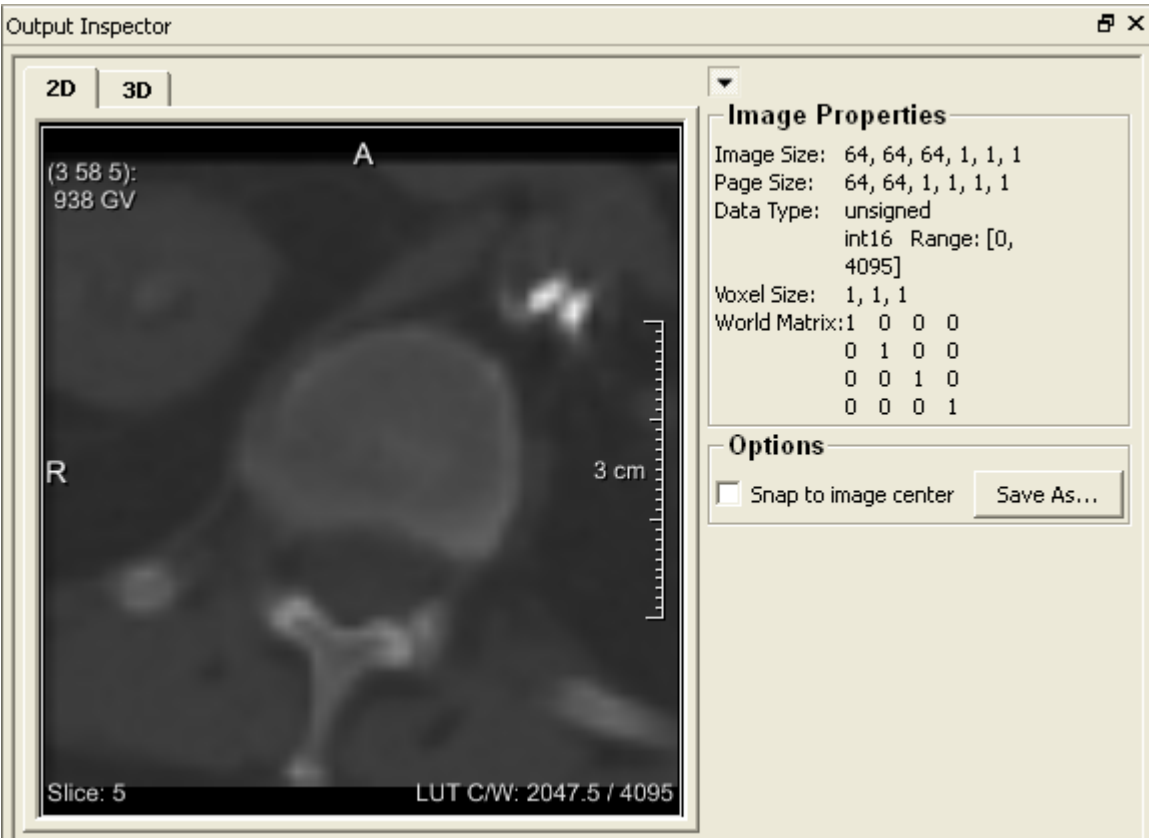
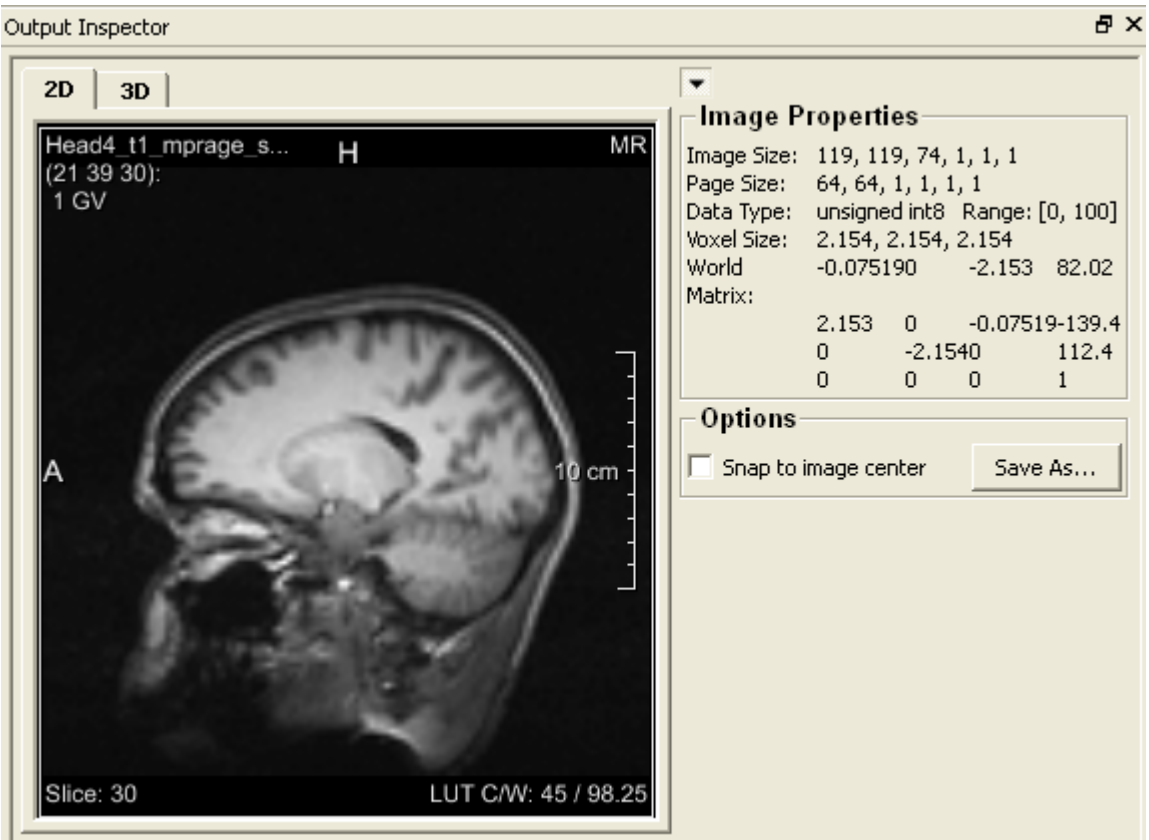


Figure 10.7. Image Properties for a Sagittal Image





## Note

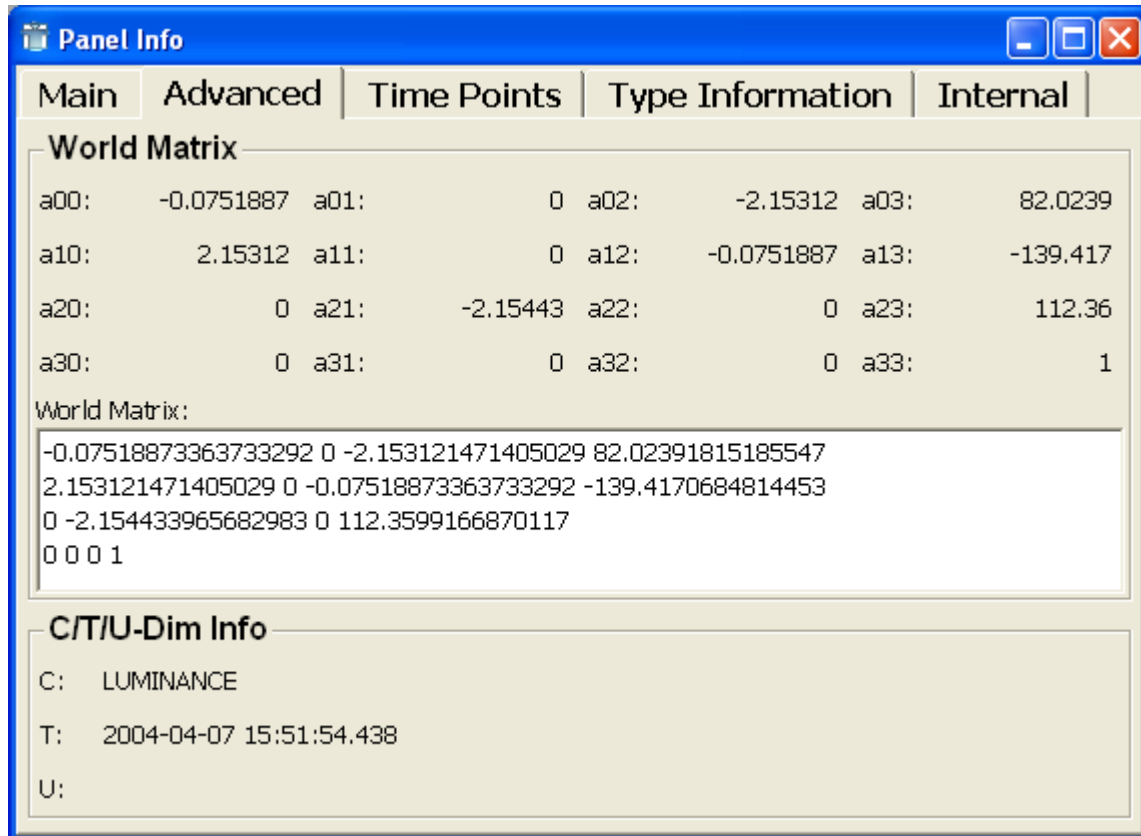
In DICOM, the voxel thickness does not necessarily correspond to the distance between slices. In MeVisLab however, the calculated voxels close the slice distance.



## Tip

Also see the `Info` module and its help for further information on the displayed data, especially the calculation of the slice thickness `z`.

**Figure 10.8. Image Properties in the `Info` Module**



## 10.7. Data Types for DICOM and TIFF

The DICOM standard does not support pixel data types other than signed and unsigned integer, and the maximum bit depth is 16. This is the reason why in MeVisLab, the data is saved as float and (u)int32 data in DCM/TIFF format. This data type is correctly encoded in the TIFF format, and the DICOM file is written as if it was an (u)int16 image.

The data is saved as follows:

- The TIFF file stored as part of a DCM/TIFF pair is a fairly standard TIFF file. For storing 3D images, the SGI 3D TIFF extension is used. 4D images are stored as 3D, the time dimension being unfold into the z-dimension.
- The DCM file in a DCM/TIFF pair is a fairly standard DICOM file, except that it does not contain the pixel data tag. The contents of such a file can be read with the `dcmddump` tool by DICOM@offis, for example. Some information gathered during the original DICOM import, such as the individual time points in a 4D data set and the values of frame specific tags, are stored in private DICOM tags. There is no official documentation of these private tags.

In MeVisLab, the libraries `libtiff` and `dcmTk` (by DICOM@offis) are used to read these files. The following applies:

- When opening such a DCM/TIFF pair, the data type stored in the TIFF file has precedence over the one in the DCM file. This mechanism is described in the help pages of the `ImageSave` and `ImageLoad` modules.
- If a DICOM file contains illegal values, the data is not regarded as valid DICOM and is completely ignored. The TIFF file is handled as if the DICOM file did not exist.

The MeVisLab binding (e.g. as used in `ImageSave` and `ImageLoad`) does not support the double image data type for TIFF.

As consequence, images with data of the type `double` cannot be saved as TIFF by `ImageSave`. As a workaround, you can either convert the data type to `float` or use `MLImageFormatSave` and `MLImageFormatLoad`.

However, the images can be saved as RAW images with double data type (not long double).



### Tip

For loading several TIFF files, use the module `ImageLoadMulti`. This should not be confused with loading a multi-page TIFF file (in which several images are saved); that format is not supported by MeVisLab.



### Tip

The page size delivered by the `ImageLoad` module is actually not determined by the `pageSizeHint` field, but by the file format module reading the image data. Only if the file format allows reading the image data in different (or even arbitrary) pages, the `pageSizeHint` is used. (That is why it is called page size *hint* and not page size.) For the TIFF format, the page size is fixed by the size of the tiles in the TIFF file holding the image data. To change the page size for successive modules, `ImagePropertyConvert` needs to be used. For RAW images, the page size hint can be set.

## 10.8. Image Processing Concepts: Pages, Slices, VirtualVolumes and more

In MeVisLab, a variety of image processing concepts is available. They differ in scope:

Page-based approaches:

- Page-based
- Voxel-based
- Slice-based
- Kernel-based

Semi-global approaches:

- Random Access (Tile requesting)
- Sequential Image Processing
- Virtual Volume

Global approaches:

- Temporary Global
- Global
- Memory Image

All those concepts are discussed in detail in the MLGuide, chapter 5 “Image Processing Concepts”.

When choosing your approach, keep in mind that some of the concepts are not scaling well for larger images. For example, the page-based approach can only be beneficial if the pages are of a size so that they actually fit into memory, or can be administered by the internal ML host / cache. Always try to set the page sizes to some reasonable values, like 128x128x1x1x1. You can do this with `ImagePropertyConvert` modules (insert them right after the loading modules in your network).



### Tip

The ITK modules frequently produce memory allocation problems for large images because they try to load the entire image at once. You can find out about the memory management in the ITK module help. Look for something like `PageExt=ImgExt` or global “memory management”. If you find these, the module cannot work page-based.

---

# Chapter 11. Introduction to C++ Modules

There are different types of modules that may be developed by the user of MeVisLab:

- Macro modules
- Image processing (ML) modules
- Open Inventor modules

There are several noticeable characteristics for all these modules types, and it is not always easy to choose the best way of implementing a new project.

## 11.1. Module and Connection Specifics on the C++ Level

ML modules on the C++-level:

- Image processing modules are objects derived from class `BaseOp` defined in the ML library and therefore are also called ML modules.
- Image inputs and outputs are connectors to objects of class `SubImage`, which are defined in the ML library.
- Inputs and outputs for abstract data structures are connectors to pointers of objects derived from class `Base` and are called Base objects.

Inventor modules on the C++-level:

- Most Inventor modules are objects derived from class `SoNode` defined in the Open Inventor library.
- Inventor inputs and outputs are connectors to objects derived from class `SoNode` defined in the Open Inventor library. Many Inventor modules will return themselves as outputs (“self”). On inputs, they may have connectors to child Inventor modules.
- Some Inventor modules are objects derived from class `SoEngine`. They are used for calculations and return their output not via output connectors but via fields.
- Inventor modules may also have input and output connectors to Base objects and Image objects.
- All standard Inventor nodes defined in the Open Inventor library are available in MeVisLab as Inventor modules.

### Modules

In [Section 2.3, “MeVisLab Modules”](#), we introduced modules by their functions and looks. Here a brief look at their programming basis:

**1 Inventor Modules:** green. Objects derived from class `SoNode` or `SoEngine` defined in the Open Inventor library.

**2 ML Modules:** blue. Objects derived from class `BaseOp` defined in the ML library.

**3 Macro Modules:** brown. MeVisLab intern objects of the type `MLABMacroModule`.

There is no special module type for MLBase objects.

### Module Inputs/Outputs

**1 Inventor:** Inputs/Outputs: half-circles. Connectors from/to objects derived from class SoNode defined in the Open Inventor library .

**2 Image:** Inputs/Outputs: triangles. Connectors from/to Image objects of type SubImage defined in the ML library.

**3 Base:** Inputs/Outputs: squares. Connectors from/to objects derived from class Base defined in the ML library.

## 11.2. Some Tips for Module Design

### 11.2.1. Macro Modules or C++ Modules?

Advantages of macros:

1. Macros are useful for creating a layer of abstraction by hierarchical grouping of existing modules.
2. Scripts can be edited on the fly:
  - no compilation and reload of the module database necessary
  - scripting possible on the module or network level
  - scripting supported by the **Scripting Assistant** View (basically a recorder for actions performed on the network)

Disadvantages:

With macros, only existing functionalities and algorithms can be used.

Conclusion:

- For rapid prototyping based on existing image processing algorithms, use macros.
- For implementing new image processing, write new ML or Open Inventor modules.

### 11.2.2. Combining Functionalities

It is possible to have ML and Open Inventor connectors in the same module. Two cases are possible:

- Type 1: ML -> visualization: Image data or properties are displayed by a visualization module. Usually a `SoSFXVImage` field gets random access to an ML image by `getTile()`. Examples: `SoView2D`, `GlobalStatistics`.
- Type 2: visualization -> ML: Modules generate an ML image from a pixmap (sequence). Examples: `SoExaminerViewer`, `SoShadowViewer`.

Generally, however, it is not always a good solution to combine that, as the processes of image processing and image visualization are usually separated.

Therefore, rather separate the ML and Open Inventor functionalities into two modules. This way,

- functionality is encapsulated and can be reused as module
- modules for the single steps may already be available in MeVisLab and spare you a new development

## 11.2.3. Tips for Module Testing

After being done with the usual module and macro tests, make sure to stress your network's algorithms and processing speed by testing with

- large data sets
- images with anisotropic voxels
- images with non-trivial world matrix (translated or rotated)

Many of the possible problems will only occur with these kinds of data.

In addition, keep in mind that modules

- need to run platform-independent
- should work on 32 and 64 bit
- should offer a well-designed panel for future users
- should come with a useful help and example network

## 11.3. Programming Examples

Besides the examples in the next chapters, several programming examples are available in the MeVisLab software development kit.

For these modules to be available, the module group “Module Examples” has to be enabled, see **Preferences** → **Module Groups**.

The module data can be found at

- Sources: Packages\MeVisLab\Standard\Sources\Examples\ML\...
- Modules: Packages\MeVisLab\Standard\Modules\Examples\ML\...

Some modules are combined in one DLL, like the MLExample modules.



### Tip

See the chapter [Section 12.3, “Combining Two Modules in One Project”](#) on how to combine modules into one DLL.

Here is an overview of the most important example modules, listed by module name.

- **AddExample** (Class: mlAddExample; DLL: MLExample)

Startup example for ML module programming.

- **BitImageExample** (Class: mlBitImageExample; DLL: MLExample)

This module demonstrates the BitImage class of the ML Tools project.

- **FieldExample** (Class: mlFieldExample; DLL: MLExample)

An example module which simply creates most ML fields and adds them to a module interface. It also uses the new Vec8Field also derived in this library.

- **GlobalPagedImageExample** (Class: mlGlobalPagedImageExample; DLL: MLExample)

This module demonstrates how a `VirtualVolume` and/or a `TVirtualVolume` instance can be used to get a random read/write access to an input image during page-based processing and to demand driven image processing.

- **Kernel3In2OutExample** (Class: `mlKernel3In2OutExample`; DLL: `MLKernelExamples`)

Example class to demonstrate the implementation of a kernel-based algorithm with three inputs and two outputs in the ML.

- **KernelExample** (Class: `mlKernelExample`; DLL: `MLKernelExamples`)

Example class to demonstrate the implementation of a kernel-based algorithm in the ML.

- **MarkerListExample** (Class: `mlMarkerListExample`; DLL: `MExample`)

Example module generating an equally spaced linear set of `XMarker` objects.

- **ObjVolume** (Class: `MObjVolume`; DLL: `MObjVolume`)

Example module to store and retrieve volume information in a hard-coded `ObjMgr` information cell. For details see the MeVisLab SDK.

- **ProcessAllPagesExample** (Class: `mlProcessAllPagesExample`; DLL: `MExample`)

This is an example module to demonstrate how to process all pages of one or more (input) images.

- **SeparableKernelExample** (Class: `mlSeparableKernelExample`; DLL: `MLKernelExamples`)

Example class of the implementation of a kernel-based algorithm in the ML which implements separable kernel filtering.

- **SmallImageInterfaceExample1**, **SmallImageInterfaceExample2** (Class: `mlSmallImageInterfaceExample`; DLL: `MLSmallImageInterfaceExamples`)

Example modules to demonstrate the class `SmallImageInterface` which provides a very simplified image processing interface for educational use. See the MeVisLab SDK for details.

- **SparseImageExample** (Class: `mlSparseImageExample`; DLL: `MExample`)

Defines an example module which uses a `VirtualVolume` as a sparse image.

- **TypeAddExample** (Class `MLTypeAddExample`)

Example class to demonstrate the integration of a new voxel data type in the ML.



### Tip

Similar examples are available for MDL panels; for those, search for modules starting with "Test..."



---

# Chapter 12. Developing ML Modules

In the following chapter, the development of ML modules will be shown in three examples.

1. An ML module that allows adding a user-defined constant value to image voxels, see [Section 12.1, “Creating a New ML Module for Adding Values”](#).
2. A more complex ML module that calculates a simple average over voxel values of an entire image, see [Section 12.2, “Creating an ML Module For Simple Average”](#).
3. Combining the two ML modules in one project (which results in one DLL), with a discussion of the pros and cons of such combinations, see [Section 12.3, “Combining Two Modules in One Project”](#).

The following examples are developed very explicitly to give you some insight into the ML, the MeVis image processing library. Another useful way to start with module development is to copy the source code of an existing module that might already have some of the wanted functionality and adapt it to your needs. For further information, please refer to the MLGuide.



## Note

Developing C++ modules requires a C++ development environment being available on your computer, e.g. Visual C++ on Windows and Xcode on Mac OS X.

## 12.1. Creating a New ML Module for Adding Values

In the following chapter, we will create a new ML module with the functionality of adding a value to all voxels, in the following steps:

- [Section 12.1.1, “Creating the Basic ML Module with the Project Wizard”](#)
- [Section 12.1.2, “Preparing the Project”](#)
- [Section 12.1.3, “Programming the Functions of the ML Module”](#)
- [Section 12.1.4, “GUI Creation/Optimizing”](#)
- [Section 12.1.5, “Creating an Example Network and Help File”](#)



## Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Standard/Modules/Examples/GettingStarted/MLSimpleAdd`, source files in `$(InstallDir)Packages/MeVisLab/Standard/Sources/Examples/GettingStarted/MLSimpleAdd`). The module can be added via quick search. As module names have to be unique, choose another name when trying to recreate this example, e.g. `MLMySimpleAdd`.

### 12.1.1. Creating the Basic ML Module with the Project Wizard

1. First of all, make sure that you have a user package defined as described in [Section 7.2, “Creating a User Package for Your Project”](#) or create it now.
2. Then run the Project Wizard and select the link **ML Module**. This starts the Wizard for C++/ML Modules. Enter the following:

- **Name:** SimpleAdd
- **Comment:** Adds a constant double value to each voxel.
- **See Also:** Arithmetic1
- **Project:** SimpleAdd
- **Target Package:** Example/General

Click **Next** to proceed.

**Figure 12.1. Entering the ML Module Properties I**

**Module Properties**

Enter the general properties of the module.

**General Module Properties**

Name: \* SimpleAdd Author: \* JDoe

Comment: Adds a constant double value to each voxel.

Keywords:

See Also: Arithmetic1

Genre: ☐ Add reference to example network

**Project Properties**

Project: \* SimpleAdd Prefix: ML

☒ Include project files

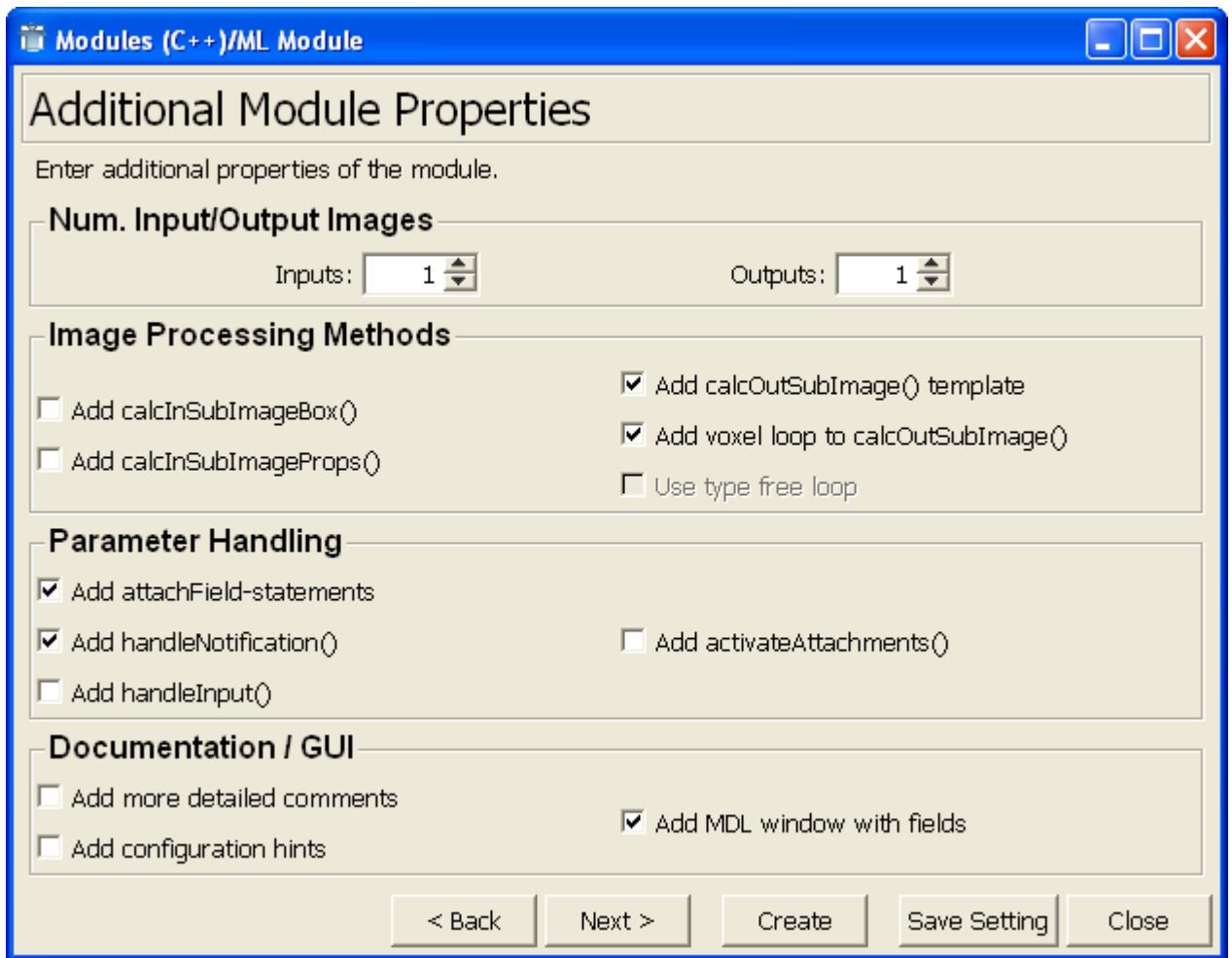
**Select Target Package**

Target Package: \* Example/General

\* : Required fields

< Back Next > Create Save Setting Close

3. On the dialog **Additional Module Properties**, the inputs and outputs as well as possible sample code can be added to the ML module.

**Figure 12.2. Entering the ML Module Properties II**

Most of the settings can be kept. Enter/change the following:

- **Inputs:** 1
  - **Outputs:** 1
  - **Add configuration hints:** Uncheck (otherwise your code will be full of text).
  - **Add calcInSubImageBox:** Uncheck (as we will not work with subimages).
4. On the dialog **Module Field Interface**, the fields of the module can be defined (more fields can be added later but this is the easiest way to add fields).

**Figure 12.3. Entering the ML Module Properties — Fields**

Modules (C++)/ML Module

### Module Field Interface

Add fields to the interface of the module.

Name	Type	Comment	Value	Enum Values
constantValue	Double	This constant value is added to each voxel.		

Field Name: 
 Field Type:

Field Comment:

Field Value:

Enum Values:

Click **New** to create a new field, then enter the following:

- **Field Name:** constantValue
- **Field Type:** Double
- **Field Comment:** This constant value is added to each voxel.
- **Field Value:** 0.

5. Click **Create** to create the module.

In the default file browser of your system, two folders are opened:

- folder with the source code: path \Example\General\Sources\ML\MLSimpleAdd
- folder with the module's GUI definition: path \Example\General\Modules\ML\MLSimpleAdd



### Note

For a full list of all created files and their contents, refer to the MLGuide, chapter “B.2. Files in an ML Project”.

The foundation of the module has been created with the Wizard. From here on, the programming starts.



## Tip

The Wizard will not close automatically. This way, you can change settings or fields and create the module once more.

After module creation, the module database needs to be reloaded.

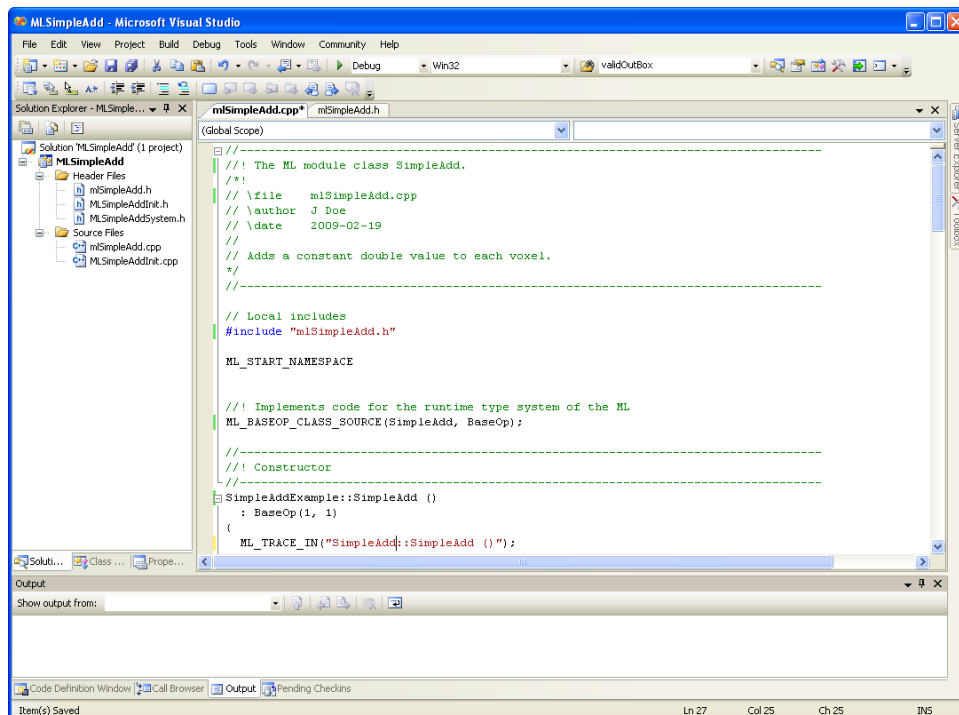
## 12.1.2. Preparing the Project

Out of the MeVisLab .pro files, the system-dependent project files (release and debug) have to be created. How this is done depends on your operating system.

On Windows, the .vcproj file should be created automatically. (If this does not happen, double-click the <ModuleName>.bat file to create one <ModuleName>.vcproj project file (the debug/release status is set in the Visual C++ environment).

Double-click the project file. Visual Studio starts, displaying a list of all project files.

**Figure 12.4. Project in Visual C++ 2005**

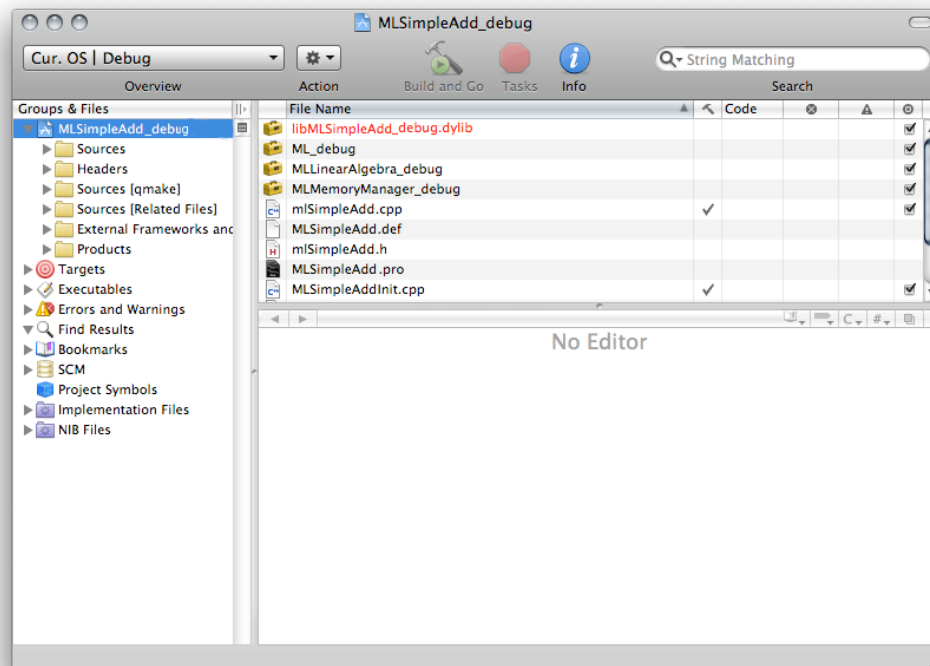


## Note

If you are encountering problems with MeVisLab on Visual C++ 2005, make sure that the Service Pack 1 (SP1) is installed. You can find all version-specific information on the MeVisLab website (<http://www.mevalab.de/>), section "Download".

On Mac:

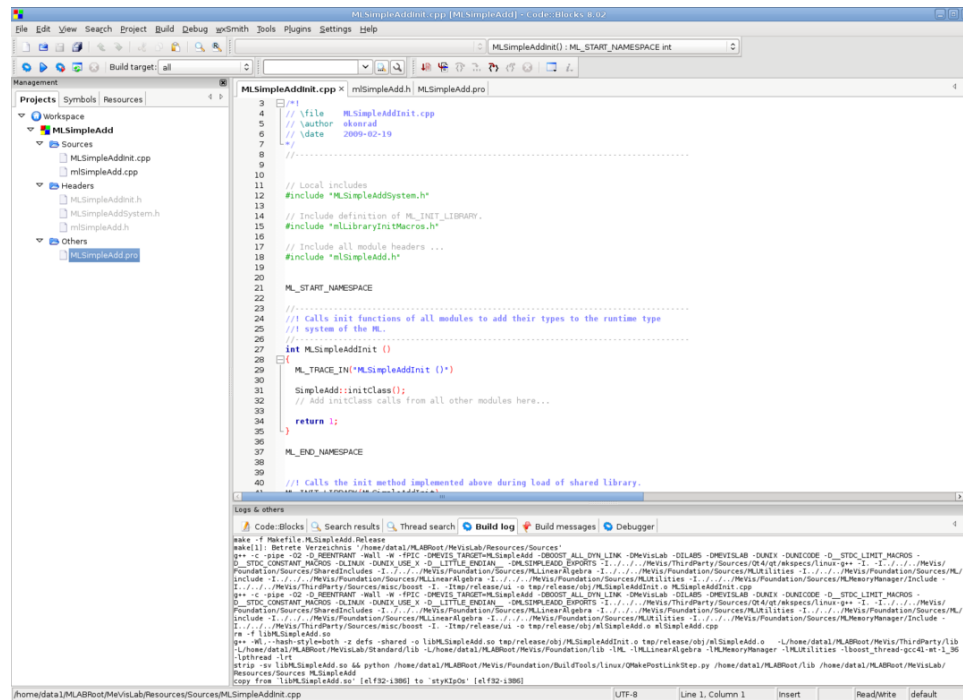
- Restart MeVisLab.
- Double-click the <ModuleName>.pro file. The application MeVisLabProjectGenerator starts which creates the two files <ModuleName>.xcodeproj and <ModuleName>\_debug.xcodeproj.
- Double-click one of the project files (debug or release). The application Xcode starts, displaying a list of all project files.

**Figure 12.5. Project in Xcode**

On Linux:

- Restart MeVisLab.
- Open a MeVisLab console, for example `$ /home/.../MeVisLab/MeVisLab2.0aGCC4.1.3/bin.`
- In the console, switch to the project folder and run the `<ModuleName>.sh` shell script. This results in two files, `Makefile.<ModuleName>` and `<ModuleName>.cbp`. The `.cbp` file is a Code::Blocks project (see <http://www.codeblocks.org/> for more information).

Figure 12.6. Project in Code::Blocks

**Note**

It is recommended to open and compile the debug versions for development.

## 12.1.3. Programming the Functions of the ML Module

Open the file `mSimpleAdd.cpp`.

**Note**

In the following code examples, the comment lines already available in the created `.cpp` file are added for better overview.

### 12.1.3.1. Implementing `calcOutImageProps`

As we add a constant value to each voxel, we need to adjust the value range of the output image, which results in:

```
outMin = inMin + constValue
outMax = inMax + constValue
```

In code, this is:

```
//-----
//!! Sets properties of the output image at output outIndex.
//-----
void SimpleAdd::calcOutImageProps (int outIndex)
{
    ML_TRACE_IN("SimpleAdd::calcOutImageProps ( )");

    // get the constant add value
    const double constValue = _constValueFld->getDoubleValue();

    // get input image's min and max values
    const double inMinValue = getInImg(0)->getMinVoxelValue();
    const double inMaxValue = getInImg(0)->getMaxVoxelValue();
```

```
// set the output image's min and max values
getOutImg(outIndex)->setMinVoxelValue(inMinValue + constantValue);
getOutImg(outIndex)->setMaxVoxelValue(inMaxValue + constantValue);
}
```

*outindex is the index number of the output connector.*

### 12.1.3.2. Implementing calcOutSubImage

1. Loop over all voxels of the output page and add the constant value. The loop is already generated by the wizard, so only the following line has to be added at the start of the method, to obtain the constant value in the correct data type:

```
// Compute subimage of output image outIndex from input subimages.
const T constantValue = static_cast<T>(_constantValueFld->getDoubleValue());
```

That is the datatype of the output image which is the data type of the input image.

2. Then change the inner line of the following loop:

```
// Process all row voxels.
for (; p.x <= rowEnd; ++p.x, ++in0Voxel, ++outVoxel){
    *outVoxel = *in0Voxel;
}
```

Change the line

```
*outVoxel = *in0Voxel;
```

to

```
*outVoxel = *in0Voxel + constantValue;
```

so that the constant value is added to the value of the input voxel.

3. Compile the project (this includes all module files) in the development environment.
4. (Re)start MeVisLab.



#### Note

If the module was edited in the debug version, MeVisLab must be run in the debug mode.

The restart is necessary

- so that the `ModuleName.def` file can be found and parsed by MeVisLab.
- so that the module DLL is copied to the correct location, from a temporary source folder to the lib folder. (If a `.def` file exists but no DLL is found, the module is displayed in red in MeVisLab.)

The module is now available in the (quick) search. Add it to the network.

### 12.1.4. GUI Creation/Optimizing

1. For optimizing the GUI of the module — that is the panel — open the `.def` file. You can do that in two ways:
  - Open the `.def` file in your development environment. The downside is that the development environment does not support the MDL language of the `.def` file.



- Open the `.def` file in the inbuilt text editor Mate, by right-clicking the module in MeVisLab and selecting **Related Files** → **MLSimpleAdd.def** from the context menu. The advantage is that Mate supports MDL (and Python and JavaScript). Therefore, it is recommended to edit MDL files primarily with Mate. (More information on Mate can be found in the MeVisLab Reference Manual.)
2. Add the line `step = 100` to the definition of the field `constantValue` in order to adjust the constant value conveniently. (Smaller steps are barely visible in the output.)

```
Window {
  Vertical {
    Field constantValue {
      tooltip = "This constant value is added to each voxel."
      step = 100    // big change for big effect
    }
  }
}
```

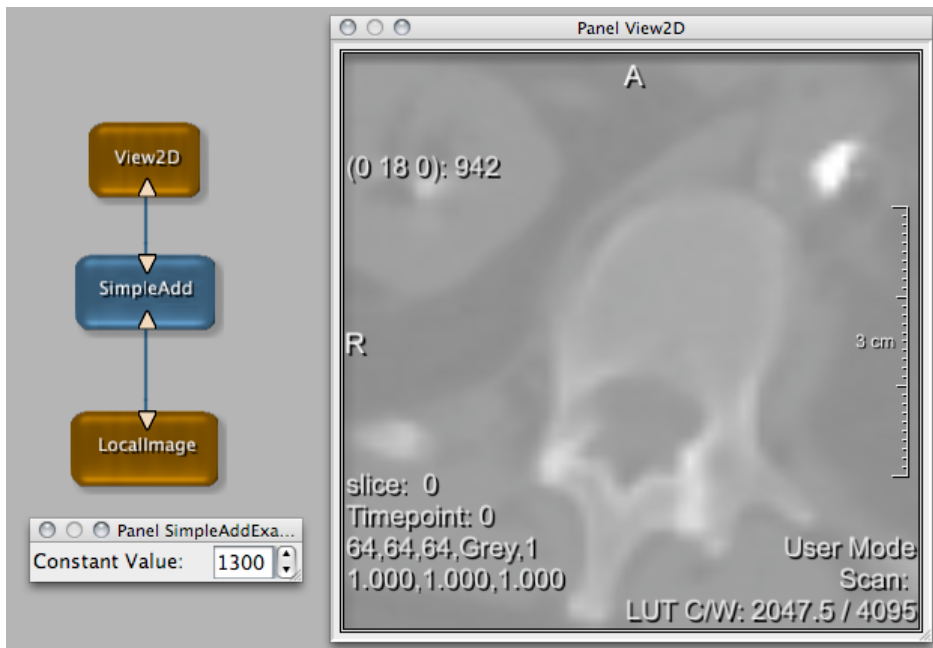
3. Reload the module definition by right-clicking the module and selecting **Reload Definition** from the context menu. This will only reload the GUI definition, not the module DLL.
4. To check if everything worked, double-click the module to open the panel and test

Congratulations, you have now implemented your first page-based and demand-driven ML image processing module!

As last step, we will create a little example network.

## 12.1.5. Creating an Example Network and Help File

1. Load the example network of the module via **File** → **Open**. Its name is automatically constructed as `<ModuleName>Example.mlab`. So far, the example network only includes the module itself.
2. Add two modules to the network, namely `LocalImage` and `View2D`. Connect the image input to the bottom connector and the image output to the top connector of `SimpleAdd`.
3. Double-click `SimpleAdd` to open its panel and `View2D` to open the viewer. When you now change the steps, the image display changes.

**Figure 12.7. Example Network for SimpleAdd**

4. To create the help, right-click the new module and select **Create Help** from the context menu. The default HTML editor (as set in the MeVisLab Preferences) opens and displays a template HTML file. Add the module-specific contents and save them.

Now the module is ready for usage.

The module including the example network and help file are delivered with the examples of MeVisLab, so feel free to check it out and play around with it.

## 12.2. Creating an ML Module For Simple Average

In the following chapter, we will create a new ML module that calculates an average over voxel values, in the following steps:

- [Section 12.2.1, “Creating the Basic ML Module with the Project Wizard”](#)
- [Section 12.2.2, “Editing the Header File of SimpleAverage”](#)
- [Section 12.2.3, “Editing the CPP File of SimpleAverage”](#)



### Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Standard/Modules/Examples/GettingStarted/MLSimpleAverage`, source files in `$(InstallDir)Packages/MeVisLab/Standard/Sources/Examples/GettingStarted/MLSimpleAverage`). The module can be added via quick search. As module names have to be unique, choose another name when trying to recreate this example, e.g. `MLMySimpleAverage`.

## 12.2.1. Creating the Basic ML Module with the Project Wizard

For the following example, we expect the user package `Example/General` to be available, see [Section 12.1.1, “Creating the Basic ML Module with the Project Wizard”](#).

1. Run the Project Wizard and select the link **ML Module**. This starts the Wizard for C++/ML Modules. Enter the following:

- a. **Name:** SimpleAverage
- b. **Comment:** Computes the average voxel value of an image.
- c. **Keywords:** Statistics Average
- d. **See Also:** ImageStatistics
- e. **Project:** SimpleAverage
- f. **Target Package:** Example/General

Click **Next** to proceed.

2. On the dialog **Additional Module Properties**, the inputs and outputs as well as possible sample code can be added to the ML module.

Most of the settings can be kept. Enter/change the following:

- **Inputs:** 1
- **Outputs:** 1
- **Add configuration hints:** Uncheck (otherwise your code will be full of text).
- **Add attachField-statements:** Uncheck (as no entry field will be used).
- **Add calcInSubImageBox:** Uncheck (as we will not work with subimages).



### Note

Although we will have no real "output" of the module, it is helpful to create an output here, as this will add some of the ML methods necessary for the module functionality. It is easier to exchange or delete some code than to add new code sections manually.

Click **Next** to proceed.

3. On the dialog **Module Field Interface**, create two new fields:

One field to keep the calculated value:

- **Field Name:** voxelValueAverage
- **Field Type:** Double
- **Field Value:** 0.

One field that will function as **Update** button:

- **Field Name:** update
- **Field Type:** Notify

- Click **Create** to create the module.

In the default file browser of your system, two folders are opened:

- folder with the source code: path \Example\General\Sources\ML\MLSimpleAverage
- folder with the module's GUI definition: path \Example\General\Modules\ML\MLSimpleAverage



### Note

For a full list of all created files and their contents, refer to the MLGuide, chapter “B.2. Files in an ML Project”.

- Reload the module database.
- Prepare the project, as described in [Section 12.1.2, “Preparing the Project”](#).

## 12.2.2. Editing the Header File of `simpleAverage`

- Open the file `mlSimpleAverage.cpp`.
- Add the following two lines to the private section

```
size_t _numVoxels;
double _sumVoxelValues;
```

They will be used as follows: All voxel values are added (`_sumVoxelValues`) and divided by the number of counted voxels (`_numVoxels`). Voxel values usually define brightness or color.

- Remove the following lines.

```
//! Sets properties of the output image at output outIndex.
virtual void calcOutImageProps (int outIndex);
```

The virtual function calling `calcOutImageProps` has to be removed because there will be no image output. If the line is not removed, a warning will be generated by the compiler. (However, the `calcOutSubImage` template is necessary.)

## 12.2.3. Editing the CPP File of `simpleAverage`

Open the file `mlSimpleAverage.cpp`.



### Note

In the following code examples, the comment lines already available in the created `.cpp` file are added for better overview, when necessary.

- Change the constructor call of the superclass from `BaseOp(1,1)` to `BaseOp(1,0)`.

This leaves our module with one input and no output image.

- Add the following code in the method `handleNotification(Field* field)`.

```
// Handle changes of module parameters and connectors here.
if (field == _updateFld) {

    _numVoxels      = 0;
    _sumVoxelValues = 0;

    processAllPages();
}
```

```
double result = 0;

if (_numVoxels > 0) {
    result = _sumVoxelValues / static_cast<double>(_numVoxels);
}

_voxelValueAverageFld->setDoubleValue(result);
}
```

The code includes the important ML BaseOp method `processAllPages()`. This method can be used in algorithms that only extract information from an image (but do not modify it). As the extraction of information is not driven by demand, the loop over all pages has to be implemented with `processAllPages()`. For further information, see the ML Guide.

3. Remove the following lines, as no image will be output by this module.

```
//-----
//! Sets properties of the output image at output outIndex.
//-----
void SimpleAverage::calcOutImageProps (int outIndex)
{
    ML_TRACE_IN("SimpleAverage::calcOutImageProps (");

    // Change properties of output image outIndex here whose
    // defaults are inherited from the input image (if there is one).
}
```

4. In the method `calcOutSubImage(...)`, remove `outSubImg` and `outIndex` from the method's signature. Result:

```
template <typename T>
void SimpleAverage::calcOutSubImage (TSubImg<T>* ,
                                     int ,
                                     TSubImg<T>* inSubImg0
                                     )
```

`outIndex` would reference an output image of the module which we do not have.

5. Replace the line:

```
const SubImgBox validOutBox(outSubImg->getBox().intersect(...
```

with the line:

```
const SubImgBox inBox = inSubImg0->getBox();
```

6. Remove the line

```
T *outVoxel = outSubImg->getImgPos(p);
```

7. Replace all occurrences of `validOutBox` with `inBox`.

8. Replace the line

```
*outVoxel = *in0Voxel;
```

with the lines:

```
_sumVoxelValues += static_cast<double>(*in0Voxel);
++_numVoxels;
```

9. At last, compile the project. Then restart MeVisLab so that the new module is registered and added to the module database.

## 12.2.4. Testing the Module

Now you can use the new module in MeVisLab.

1. Add your new module `SimpleAverage` and a `LocalImage` module to a new network. Connect them and load an image.
2. Then double-click `SimpleAverage` to open its automatic panel and click the **Update** button on the module panel. The calculated output of `SimpleAverage` is displayed.

A module with a similar functionality is available in MeVisLab, called `ImageStatistics`.

Add `ImageStatistics` via the quick search and compare its mean value with the displayed value of `SimpleAverage`. You will find that the results are almost the same apart from the rendering error in the display.



### Tip

This test network is delivered as the example network for `SimpleAverage`.

## 12.3. Combining Two Modules in One Project

In the following chapter, we will merge our two modules (`SimpleAdd` and `SimpleAverage`) into one project, in the following steps:

- [Section 12.3.1, “Copying the Souce Files”](#)
- [Section 12.3.2, “Editing and Recompiling the .pro File”](#)
- [Section 12.3.3, “Editing the Project in the Development Environment”](#)
- [Section 12.3.4, “Editing the Module Definition \(.def\)”](#)
- [Section 12.3.1, “Copying the Souce Files”](#)

Per project, one DLL (`.dynlib/.so`) file is created and transferred, and the modules might share common includes etc. within one project.

Therefore, this example is a showcase on how to build a larger library by augmenting an existing project.

In this example, we will merge the `SimpleAverage` module into the `SimpleAdd` project. For two modules, this is an arbitrary decision; for larger projects, always merge into the existing project.



### Tip

The source code of this example is delivered with MeVisLab (source files in `$(InstallDir)Packages/MeVisLab/Standard/Sources/Examples/GettingStarted/MLSimpleMerged`). However, as module names have to be unique, no `.def` file is delivered (so the module is not available in MeVisLab), to avoid collisions with the examples above.

### 12.3.1. Copying the Souce Files

Copy the `mlSimpleAverage.cpp` and `mlSimpleAverage.h` files to the source folder of `SimpleAdd`.

### 12.3.2. Editing and Recompiling the .pro File

1. Open `mlSimpleAdd.pro` in any text editor.
2. Add `mlSimpleAverage.h` to the `HEADERS` section.

3. Add `mlSimpleAverage.cpp` to the `SOURCES` section. Make sure that the previous line is terminated with a backslash with NO whitespaces behind it. The last line does not need to be terminated by a backslash.
4. Recompile the `.pro` file (run `.bat` on Windows, `.sh` on Linux, double-click `.pro` on Mac).

For the resulting `.pro` file, see `$(InstallDir)Packages/MeVisLab/Standard/Sources/Examples/GettingStarted/MLSimpleMerge`.

### 12.3.3. Editing the Project in the Development Environment

1. Open the `SimpleAdd` project in your development environment.
2. Open `SimpleAverage.h`.
3. Exchange the line

```
#include "MLSimpleAverageSystem.h"
```

by

```
#include "MLSimpleAddSystem.h"
```

4. Exchange the macro in the class definition (this handles exporting symbols under Windows)

```
MLSIMPLEAVERAGE_EXPORT
```

by

```
MLSIMPLEADD_EXPORT
```

The new module in this project (i.e. `SimpleAdd`) needs to be initialized for the runtime-type system.

5. Open `MLSimpleAddInit.cpp`.
6. Add the line

```
#include "mlSimpleAverage.h"
```

below the line

```
#include "mlSimpleAdd.h"
```

7. Add the line

```
SimpleAverage::initClass();
```

below the line

```
SimpleAdd::initClass();
```

This registers the classes to the ML runtime type system.

8. Recompile the project.



#### Note

`mlSimpleAverage.cpp` does not have to be edited.

For the resulting sources, see `$(InstallDir)Packages/MeVisLab/Standard/Sources/Examples/GettingStarted/MLSimpleMerged`.

## 12.3.4. Editing the Module Definition (.def)

1. Open the file `MLSimpleAverage.def` in Mate.

Copy the definition of the module `SimpleAverage` into the clipboard (this is at least from the line

```
MLModule SimpleAverage {
```

to the last closing curly bracket `}}`)

2. Open the file `MLSimpleAdd.def`.

Paste the definition of the `SimpleAverage` module below the definition of the `SimpleAdd` module.

Exchange the line in the definition of the `SimpleAverage` module

```
DLL = "MLSimpleAverage"
```

by the line

```
DLL = "MLSimpleAdd"
```

## 12.3.5. Cleaning up Folders and Example Networks

1. Copy the example network and HTML documentation of the `SimpleAverage` module to the according folders of the `SimpleAdd` module. The paths to those files should be relative, so they are still correct.
2. (Re)move the old files and folders of the `SimpleAverage` module from the folders `/Sources` and `/Modules` so that no conflicts arise.
3. (Re)start MeVisLab.

Both modules can now be added e.g. via a quick search. However, you will find that in the About information, the same DLL will appear for both modules.



---

# Chapter 13. Developing Inventor, WEM and CSO Modules

The following chapter gives a short brief overview and some references to the possibilities of developing Inventor, WEM and CSO modules.



## Tip

Additional documents on various MeVisLab features and aspects can be found in the Toolbox Class Reference.

## 13.1. Inventor Modules

New Inventor modules may be added by creating some basic Open Inventor module types with the Project Wizard and extending them. For the available options, see the MeVisLab Reference Manual, chapter “Project Wizard”.

For documentation on Open Inventor, see the Inventor Module Help (for an introduction on Open Inventor and module-related help) and the Inventor Reference (converted from the original man pages).

## 13.2. Winged Edge Mesh Library (WEM)

The approach of the WEM (Winged Edge Mesh) library is to unitize the generation, the processing and the rendering of surface representations. The library in MeVisLab offers a basis for dealing with common tasks: an iso surface can be generated at a certain threshold out of medical images, the resulting surface can be reduced in its amount of primitives or can be smoothed by using different algorithms. For rendering, the surface can be colored in order to reflect certain additional information or according to a flexible coloring scheme out of the image data itself. Finally, all the generated and modified surfaces can be saved and loaded with a variety of different file formats that are compliant with standard 3D applications.

New WEM modules may be created with the Project Wizard, see the MeVisLab Reference Manual, chapter “Project Wizard”.



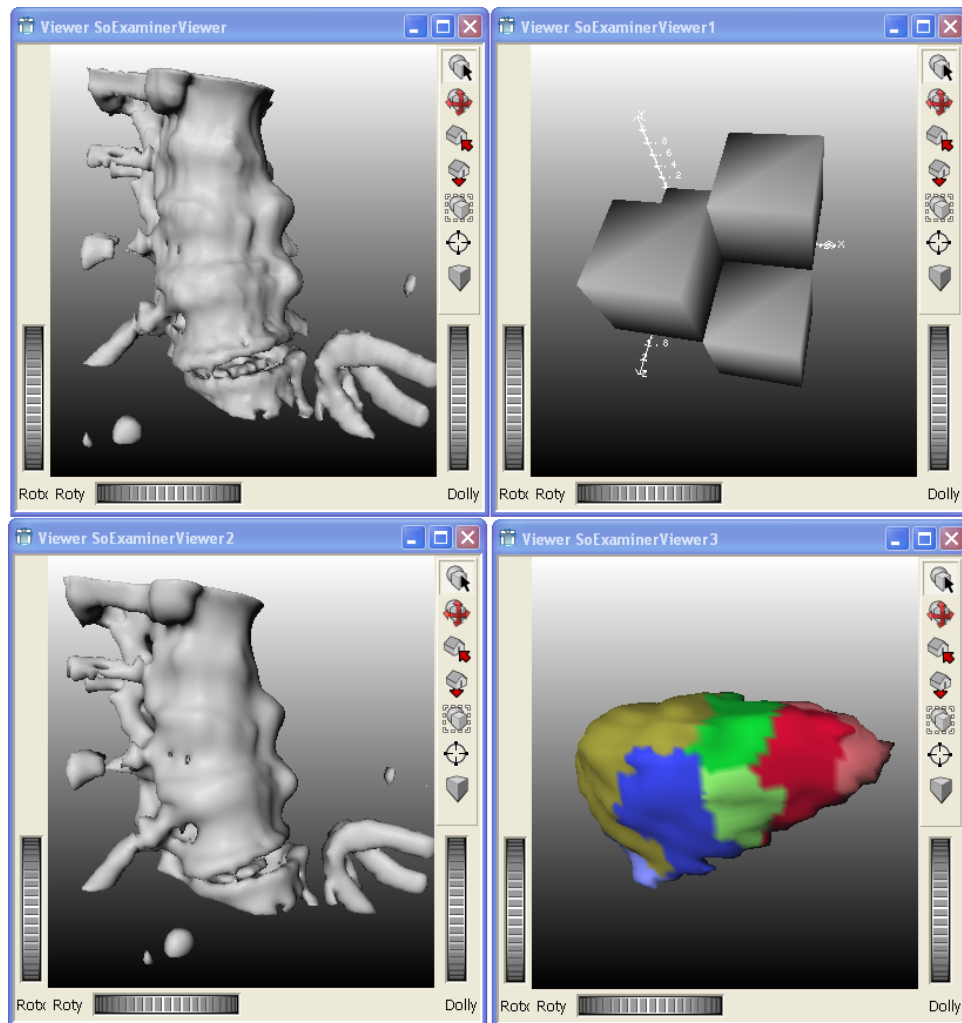
## Note

The WEM wizard is intended for implementing ML-based modules. Although there are WEM modules based on Open Inventor in the library, the creation of those is not documented.

For documentation on WEM in MeVisLab, see the Toolbox Class Reference, section “WEM”. The chapter “WEM Data Structure” gives an overview of the concepts behind WEM.

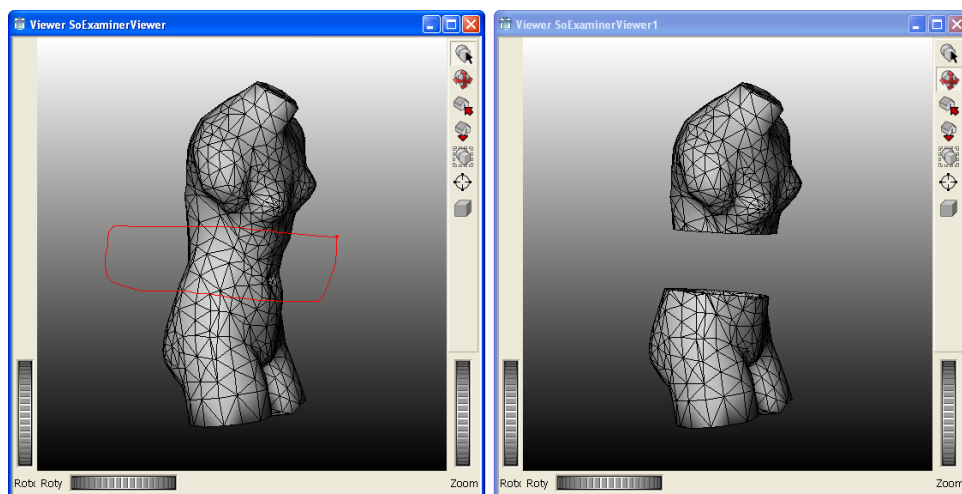
For available WEM modules, enter “WEM” in the quick search. Their example networks offer insights into the features and functionality of WEM.

**Figure 13.1. WEM IsoSurface Example Network**



Although the focus of WEM is more on calculation and display than on interaction, interactivity can be implemented like in the following example:

**Figure 13.2. WEM Extrude Example Network**



## 13.3. Contour Segmentation Objects (CSO)

The CSO library provides data structures and modules for freehand drawing, semi-automatic or automatic generation of contours in voxel images. Furthermore, these contours can be analyzed, maintained, grouped, and converted into a voxel image again.

In the CSO library, all coordinates of the object are stored in world space. The contours themselves are called CSO and are 3D objects. The CSOs are not attached to any special image and can freely be interchanged between different images or the same image in different resolutions. Due to their 3D nature, the CSOs are not restricted to the axial plane or to ortho planes in general, but can be generated on oblique MPRs. In one CSOList, arbitrarily oriented CSOs can coexist.

For documentation on CSO, see the page `$(InstallDir)Packages/MeVisLab/Standard/Modules/ML/MLCSOModules/Overview/CSOOverview.html` and also the Toolbox Reference, section “CSO”.

CSO modules cannot be created with the wizard. For extending CSO features, see the two base classes `CSOGenerator` (`$(InstallDir)Packages/MeVisLab/Standard/Sources/ML/MLCSO/CSOBase/CSOModuleBase/`) and `CSOProcessor` (`$(InstallDir)Packages/MeVisLab/Standard/Sources/Inventor/SoCSO/CSOProcessor/`).

For available CSO modules, enter “CSO” in the quick search. Their example networks offer insights into the features and functionality of CSO.

**Figure 13.3. Freehand Contours with the SoView2CSOEditor Example Network**

