
Getting Started

First Steps with MeVisLab



Getting Started

Copyright © 2003-2025 MeVis Medical Solutions
Published 2025-06-26

Table of Contents

1. Before We Start	10
1.1. Welcome to MeVisLab	10
1.2. Coverage of the Document	10
1.3. Intended Audience	11
1.4. Conventions Used in This Document	11
1.4.1. Activities	11
1.4.2. Formatting	11
1.5. How to Read This Document	12
1.6. Related MeVisLab Documents	12
1.7. Glossary (abbreviated)	13
2. The Nuts and Bolts of MeVisLab	15
2.1. MeVisLab Basics	15
2.2. Development in MeVisLab	16
2.3. MeVisLab Modules	17
2.4. Fields	18
2.5. Networks	19
2.6. Overview of Important Files	20
2.7. User Interfaces Controls	21
2.8. Scripting	22
2.9. How to Find More Information on Networks and Modules	22
3. Loading and Viewing Images	24
3.1. The MeVisLab GUI	24
3.2. Searching and Adding Modules	25
3.3. Using the ImageLoad Module	28
3.4. Adding Viewers to ImageLoad	33
3.4.1. Adding the View2D Module	33
3.4.2. Adding the View3D Module	37
3.5. Alternative Ways to Load Images	37
3.5.1. Dragging Images onto the Workspace	37
3.5.2. Using the LocalImage Module	38
3.6. A Note on Importing DICOM Images	39
4. Implementing a Contour Filter	41
4.1. Loading the Input Image	41
4.2. Implementing the Contour Filter	42
4.3. Parameter Connection for Synchronization	46
5. Defining a Region of Interest (ROI)	49
5.1. Creating a Viewer with a Selection Rectangle	50
5.2. Adding a Second Viewer for the Subimage	50
5.3. Adding the Interactivity for the Viewers	51
6. Excursion: Functionality Overview	56
6.1. Image Handling and Processing	56
6.1.1. Image Handling	56
6.1.2. Image Properties	56
6.1.3. Basic Image Processing	56
6.1.4. Filter	57
6.1.5. Segmentation	57
6.2. Visualization	57
6.2.1. 2D Viewing	57
6.2.2. 3D Viewing	58
6.2.3. Lookup Tables	58
6.3. Data Objects	58
6.3.1. Markers	58
6.3.2. Curves	59
6.3.3. Contours	59
6.3.4. Surface objects	59

6.4. Miscellaneous	59
6.4.1. Fields	59
6.4.2. Diagnostic	60
7. Creating an Open Inventor Scene	61
7.1. Introduction to Open Inventor	62
7.2. Creating the Applicator	64
7.3. Creating the Interaction	67
7.4. Creating the Anatomical Image	70
7.5. Finishing the Complete Open Inventor Scene	71
8. Starting Development with Package Creation	75
8.1. What are Packages	75
8.2. Creating a User Package for Your Project	77
9. Introduction to Macro Modules	78
10. Developing a Macro Module for an Applicator	80
10.1. Creating a Basic Global Macro	80
10.2. Adding the Macro Parameters and Panel	84
10.3. Programming the Python Script	89
10.4. Addition: Shifting the Whole Tip	93
11. GUI Design in MeVisLab	96
11.1. MeVisLab Definition Language (MDL)	97
11.1.1. MDL Validator	97
11.1.2. MDL Controls	98
11.1.3. MDL GUI definition	101
11.1.4. A Note on Fields in Scripting Interfaces	102
11.2. Developing the ExampleToggleButton	105
11.2.1. Creating the Macro Module	105
11.2.2. Defining the Interfaces	106
11.2.3. Programming the Button Action in Python	107
11.2.4. Referencing the Command in the MDL Script	108
11.2.5. Persistent Field Values	109
11.2.6. Implementing a Keyboard Shortcut	109
11.2.7. Arranging Multiple Buttons	110
11.2.8. Auto Layouting with the AlignGroups Control	111
11.2.9. Prototypes for Controls	111
11.2.10. Designing Larger GUIs	111
11.3. MDL Styles	112
11.3.1. How to Use MDL Styles	113
11.3.2. Defining Global Styles	114
11.3.3. Creating Custom MDL Controls	115
11.4. Customize GUI Appearance Using Qt Style Sheets (CSS)	116
12. Excursion: Image Processing in ML	119
12.1. Some Advanced Information on Image Processing	119
12.2. Structure of MeVisLab	119
12.3. Coordinate Systems	120
12.4. Affine Transformations	121
12.5. DICOM Data and Coordinates	122
12.6. Coordinate Systems in the MeVisLab GUI	124
12.7. Data Types for DICOM and TIFF	125
12.8. Image Processing Concepts: Pages, Slices, VirtualVolumes, and More	126
13. Introduction to C++ Modules	128
13.1. Module and Connection Specifics on the C++ Level	128
13.2. Some Tips for Module Design	129
13.2.1. Macro Modules or C++ Modules?	129
13.2.2. Combining Functionalities	129
13.2.3. Tips for Module Testing	130
13.3. Programming Examples	130
14. Developing ML Modules	132
14.1. Creating a New ML Module for Adding Values	132

14.1.1. Creating the Basic ML Module with the Project Wizard	132
14.1.2. Preparing the Project	137
14.1.3. Programming the Functions of the ML Module	137
14.1.4. GUI Creation/Optimizing	138
14.1.5. Creating an Example Network and Help File	139
14.2. Creating an ML Module For Simple Average	139
14.2.1. Creating the Basic ML Module with the Project Wizard	140
14.2.2. Editing the Header File of SimpleAverage	141
14.2.3. Editing the CPP File of SimpleAverage	141
14.2.4. Testing the Module	143
14.3. Combining Two Modules in One Project	144
14.3.1. Copying the Source Files	144
14.3.2. Editing and Recompiling the CMakeLists.txt File	144
14.3.3. Editing the Project in the Development Environment	145
14.3.4. Editing the Module Definition (.def)	146
14.3.5. Cleaning up Folders and Example Networks	147
15. Developing a Base Communication	148
15.1. A Note on Base Types Checks	149
15.1.1. Base Connectors and Field Types	149
15.1.2. Overriding Base Type Checks	150
15.1.3. Implementing Base Type Checks	150
15.2. Developing the MLBaseOwner Module and the BaseMessenger Class	151
15.2.1. Creating the BaseCommunication Project in the Wizard	151
15.2.2. Adding New Files	156
15.2.3. Adding References to the new Files in CMakeLists.txt	157
15.2.4. Adding Contents to BaseMessenger.h	157
15.2.5. Add Contents to BaseMessenger.cpp	158
15.2.6. Editing MLBaseCommunicationInit.cpp	159
15.2.7. Editing mlBaseOwner.h	159
15.2.8. Editing mlBaseOwner.cpp	160
15.2.9. Making MLBaseCommunication classes known	162
15.2.10. Adding an object wrapper for MLBaseCommunication objects	163
15.3. Developing the SoBaseReceiver Module	163
15.3.1. Creating the New Open Inventor Module with the Wizard	164
15.3.2. Editing CMakeLists.txt of SoBaseReceiver	167
15.3.3. Edit SoBaseReceiver.h	167
15.3.4. Editing SoBaseReceiver.cpp	168
16. Using the TestCenter	171
16.1. Introduction to Testing in MeVisLab	171
16.2. Developing a Test Case	172
16.2.1. Creating a New Test Case	172
16.2.2. Populating the Test Network	175
16.2.3. Editing the Module Settings	175
16.2.4. Creating a First Test Script with Manual Threshold Setting	176
16.2.5. Automating the Test Case with the FieldValueTestCaseEditor	179
16.2.6. Automating the Test Case with an Iterative Test	185
16.2.7. Grouping Test Functions	187
16.2.8. Enhancing Test Reports with ScreenShots	188
16.2.9. Disabling Test Functions	189

List of Figures

1.1. Welcome Screen and Documentation Links	13
2.1. Image Processing Pipeline	17
2.2. Network Layout	20
2.3. Module Context Menu: Show Help	22
3.1. MeVisLab User Interface	24
3.2. View Selection	25
3.3. Modules Menu and Module Browser	26
3.4. Quick Search Options	27
3.5. Quick Search Results	27
3.6. ImageLoad Module	27
3.7. ImageLoad Panel and Output Inspector	29
3.8. Adjusting the Window/Level	29
3.9. Output Inspector with Image Properties	30
3.10. Output Inspector with Additional Information Display	31
3.11. 3D Output Inspector	31
3.12. Connector Details in the Edit Menu	32
3.13. Connector Details in the Preferences	32
3.14. Connector Details Depending on Zoom	33
3.15. Setting up the Connection	34
3.16. Panel of View2D	34
3.17. Opening the Settings Panel of View2D	35
3.18. Settings Panel of View2D	35
3.19. Automatic and Settings Panel of View2D	36
3.20. Connecting the View3D Module	37
3.21. The View3D Panel	37
3.22. LocalImage Module	38
3.23. Show the Internal Network	38
3.24. Internal Network of the LocalImage Module	39
3.25. DicomImport	39
4.1. Example Network Contour Filter	41
4.2. Viewing the Input Image for the Contour Filter	42
4.3. Adjust Filter Parameters	43
4.4. Constructing the Filter Pipeline — Convolution Output	44
4.5. Constructing the Filter Pipeline — Morphology Output	44
4.6. Constructing the Filter Pipeline — Arithmetic2 Output	44
4.7. Creating a New Group	45
4.8. Resulting Contour Filter Network	45
4.9. Establishing the Parameter Connections	47
4.10. Resulting Network	47
5.1. Example Network ROISelection	49
5.2. Viewer with Selection Rectangle	50
5.3. Viewer for the Subimage	51
5.4. Searching for World to Voxel Conversion	52
5.5. WorldVoxelConvert Panel	52
5.6. WorldVoxelConvert Modules Added	53
5.7. Adding the Parameter Connections	54
5.8. Example Network ROI Selection	55
7.1. Example Network: Open Inventor Result	61
7.2. Applicator Only	62
7.3. Traversing in Open Inventor	63
7.4. Creating the Applicator Shaft	64
7.5. Coloring the Applicator Shaft	65
7.6. Adding an Applicator Tip	65
7.7. Adding Translation and Grouping	66
7.8. Finishing the Applicator	67

7.9. Using SoCenterballManip	68
7.10. SoCenterballManip — Turned	69
7.11. Connecting Parameters	69
7.12. Combining Interaction and Applicator	70
7.13. Loading a Local Image	70
7.14. Adding the GigaVoxel Renderer	71
7.15. Adding the Windowing to the Applicator	71
7.16. Combining the Groups	72
7.17. Combined Graphic Elements	73
7.18. Adding the Applicator Scaling	73
7.19. Improved Applicator/Interaction Arrangement	74
8.1. Example for a Package Tree	75
8.2. Preferences — Packages	76
8.3. Package Wizard	77
10.1. Starting a new Macro from the Existing Applicator	80
10.2. Existing Applicator with Clean Instance Names	81
10.3. Macro Module Wizard	81
10.4. Selecting a Genre	82
10.5. Macro Module Properties	83
10.6. File Browser with the New Macro Module Files	84
10.7. ApplicatorMacro as Macro Module	84
10.8. ApplicatorMacro.script in MATE	85
10.9. ApplicatorMacro Module with Output Connector	86
10.10. Internal Network of the ApplicatorMacro Module	86
10.11. Automatic Panel of the ApplicatorMacro Module	87
10.12. Panel of the ApplicatorMacro Module	88
10.13. Parameters for Diameter Setting	90
10.14. Changing the Diameter of the Applicator	91
10.15. Strange Behavior of the ApplicatorMacro	92
10.16. Adding the Correct Tip Translation	92
10.17. Complete ApplicatorMacro	93
10.18. Feeding the SoCalculator Module	94
10.19. Improved Applicator Macro Module	95
11.1. View3D Panels as Example for GUI Elements	96
11.2. Fields as Model	99
11.3. Controls as View/Controller	99
11.4. Controls as Views/Controller	100
11.5. View3D Panel with C++ Class Names of Included MDL Controls for Scripting	101
11.6. Command Execution Context	104
11.7. Contexts of the Scripting Console	105
11.8. ExampleToggleButton	106
11.9. ExampleToggleButton	109
11.10. Buttons in a Grid	111
11.11. View3D Panels with the <code>Panel</code> Control	112
11.12. Redesigned Panel	113
11.13. Entering Style Settings	114
11.14. ExampleToggleButton with Application Style Panel	114
11.15. Color Chooser Example Control	116
11.16. View3D Panel with Qt Widgets	117
12.1. MeVisLab Structure	119
12.2. Coordinate Systems	120
12.3. Matrix Multiplication	121
12.4. World Coordinates in Context of the Human Body	122
12.5. The DICOM Tag Browser	123
12.6. Image Properties for an Ideal Image	124
12.7. Image Properties for a Sagittal Image	124
12.8. Image Properties in the <code>Info</code> Module	125
14.1. Entering the ML Module Properties	133

14.2. Entering the Imaging Module Properties	134
14.3. Additional Module Properties	135
14.4. Entering the ML Module Properties — Fields	136
14.5. Example Network for SimpleAdd	139
15.1. Example Network for ML Module and an Open Inventor Module	148
15.2. Mouse-over Information for Base Connectors	149
15.3. Mouse-over Information for Different Base Connectors in One Module	149
15.4. Base Field Connection Checked for Type Compatibility	150
15.5. Project Wizard — Module Properties	152
15.6. Project Wizard — Imaging Module Properties	153
15.7. Project Wizard — Additional Module Properties	154
15.8. Project Wizard — Module Field Interface	155
15.9. Resulting <code>BaseOwner</code> Module	163
15.10. <code>SoBaseReceiver</code> Module Alternative	163
15.11. Project Wizard — General Module Properties	164
15.12. Project Wizard — Module Type	165
15.13. Project Wizard — Module Field Interface	165
16.1. Creating a New Test Case	173
16.2. New Test Case in Test Selection	174
16.3. New Test Case in the Package Path	175
16.4. Basic Test Case Setup	176
16.5. Test Functions in the <code>TestCaseManager</code>	178
16.6. Report for <code>ManualTest_75</code>	179
16.7. The <code>FieldValueTestCaseEditor</code> Panel	180
16.8. Dragging Fields into the Parameter List	181
16.9. Dragging Fields into the Expected Results List	182
16.10. The Resulting Panel	183
16.11. Our Automatic <code>FieldValue</code> Tests Added	184
16.12. Report for <code>AutomaticTest_1</code>	184
16.13. Our Iterative Test in the Test Center	186
16.14. Report for <code>AutomaticTest_2</code>	187
16.15. Grouped Test Functions	188
16.16. Report for <code>ScreenShot</code> Example	189

List of Tables

1.1. List of MeVisLab Documents	12
2.1. Module Types	17
2.2. Connectors	18
2.3. Connections	18
2.4. Important Files	21

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, for example 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 over assembling macro modules and programming modules in C++ to writing tests on a network level for modules. 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
- write tests for a manual and automatic testing of modules and networks



Note

Depending on your software license, not all features of MeVisLab may be available. For licensing information, please refer to the MeVisLab website (<https://www.mevalab.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 is required. The examples in this document will also make use of the MDL (MeVisLab Definition Language).
- 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. Conventions Used in This Document

1.4.1. 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).

Mouse-over: Move the mouse pointer to the object to display additional information in a tool tip, for example on panels and connectors.

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.

1.4.2. Formatting

Views: ***Parameter Connections Inspector***

MeVisLab modules: `ImageLoad`:

Parameters: *Diameter*

Programming code: `*outVoxel = *inVoxel0`

and also

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

1.5. 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 7, *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 10, *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 13, *Introduction to C++ Modules*](#).

In addition, the following sections might be of help:

- [Chapter 12, *Excursion: Image Processing in ML*](#) for some background on coordinate systems and how they are used in MeVisLab.
- [Chapter 8, *Starting Development with Package Creation*](#) for the package structure of the module database and how to create your own packages for development.

1.6. Related MeVisLab Documents

Besides the document at hand, the following documents are available:

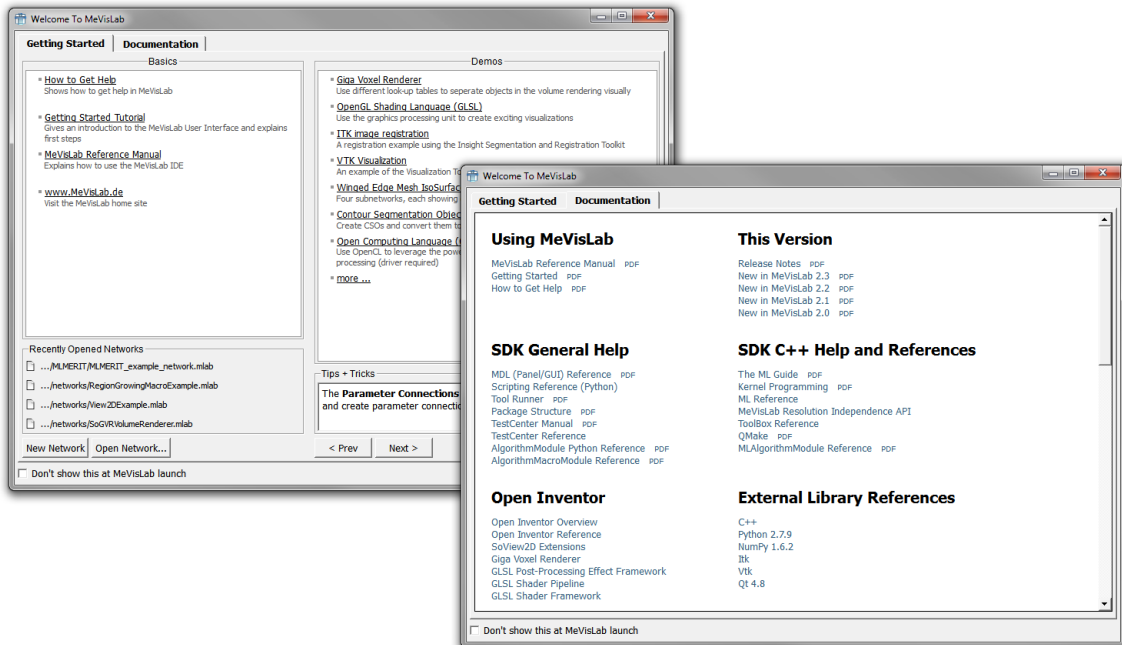
Table 1.1. List of MeVisLab Documents

Title	Contents
How to Get Help?	Overview over available help in MeVisLab
MeVisLab Reference Manual	Reference for the MeVisLab user interface
The ML Guide	MeVis Image Processing Library — Programming Guide
ML Reference	MeVis Image Processing Library — API description
MDL Reference	MeVisLab Definition Language (MDL) Panel/GUI Reference
Open Inventor Overview	Help for Open Inventor Modules
Open Inventor Reference	Reference for all implemented Open Inventor classes (converted from the original manpages)
Scripting Reference (Python)	Scripting Reference for Python in MeVisLab
Toolbox Reference	MeVisLab Toolbox Class Reference for various API libraries
TestCenter Reference	Class Reference for the TestCenter
Package Structure	Information about the package structure in MeVisLab
ToolRunner	Manual for ToolRunner, a stand-alone program for building projects and help files
CMake	CMake in the MeVisLab context, including explanations for how to use <code>CMakeLists.txt</code> files

To search in the online documentation, use **Help** → **Search in Documentation**, see the MeVisLab Reference Manual, “Search in Documentation”.

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 8, 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.

1.7. 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.
Module	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 Module.
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 <http://www.itk.org/> and <https://www.mevislab.de/> 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.
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 http://www.vtk.org/ and https://www.mevislab.de/ 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 2.1, “MeVisLab Basics”](#)
- [Section 2.2, “Development in MeVisLab”](#)
- [Section 2.3, “MeVisLab Modules”](#)
- [Section 2.5, “Networks”](#)
- [Section 2.6, “Overview of Important Files ”](#)
- [Section 2.7, “User Interfaces Controls”](#)
- [Section 2.9, “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
- Multi-platform (Windows and Linux supported)
- Scripting support (Python)
- 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 around 960 Standard modules in the MeVisLab SDK core, around 3300 modules delivered in total (with around 350 ITK modules, around 1400 VTK modules, and around 440 modules in the Fraunhofer MEVIS release)
- Long time maintenance

2.2. Development in MeVisLab

In MeVisLab, development can be done on three levels:

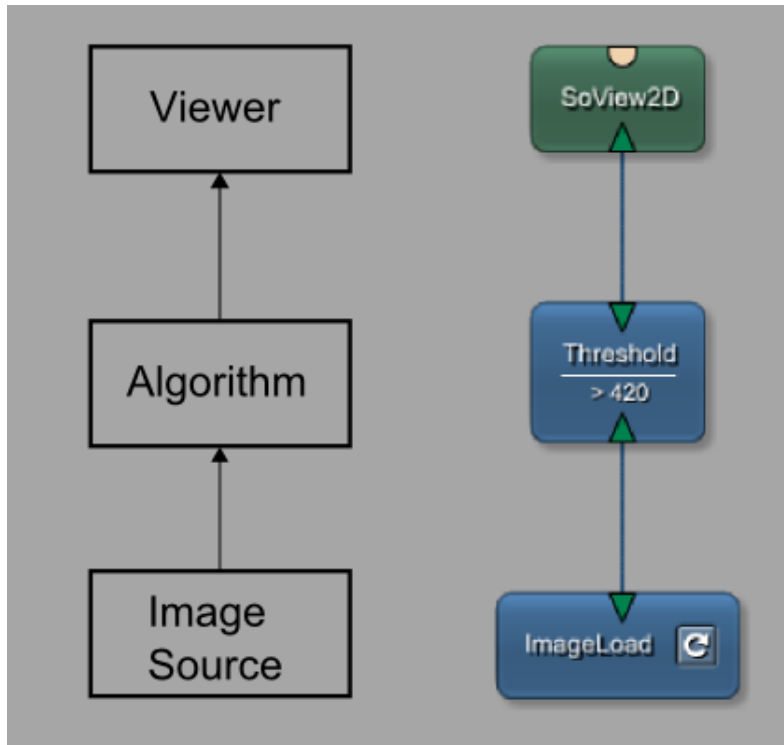
- **Visual level:** Programming with “plug and play”: Individual image processing, visualization and interaction modules can be combined to complex image processing networks using a graphical programming approach.
- **Scripting level:** Creating macro modules and applications based on macro modules: Python 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 which is only available for very close partners of MeVis).

In MeVisLab, the algorithms are visualized as 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" (for s cene o bject)
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	Shape	Definition
	triangle	ML images
	half-circle	Inventor scene
	square	Base objects: pointers to data structures

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



Note

A special case are Inventor engine fields; they have no value representation themselves unless connected to a fitting destination field. See Section 28.3, “Connecting Inventor Engines to ML Modules” for more details.

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. Fields

Fields define the interface of a module, that means they are also the basis of the connector types given above.

They come in two types:

1. In/out fields — connected by data connections

- Images
- Nodes
- Objects

2. Parameter fields — connected by parameter connections

- Numbers, Strings, Booleans
- Vectors
- Triggers

Field changes trigger events handled by field listeners. Field connections are a special forms of field listeners.



Tip

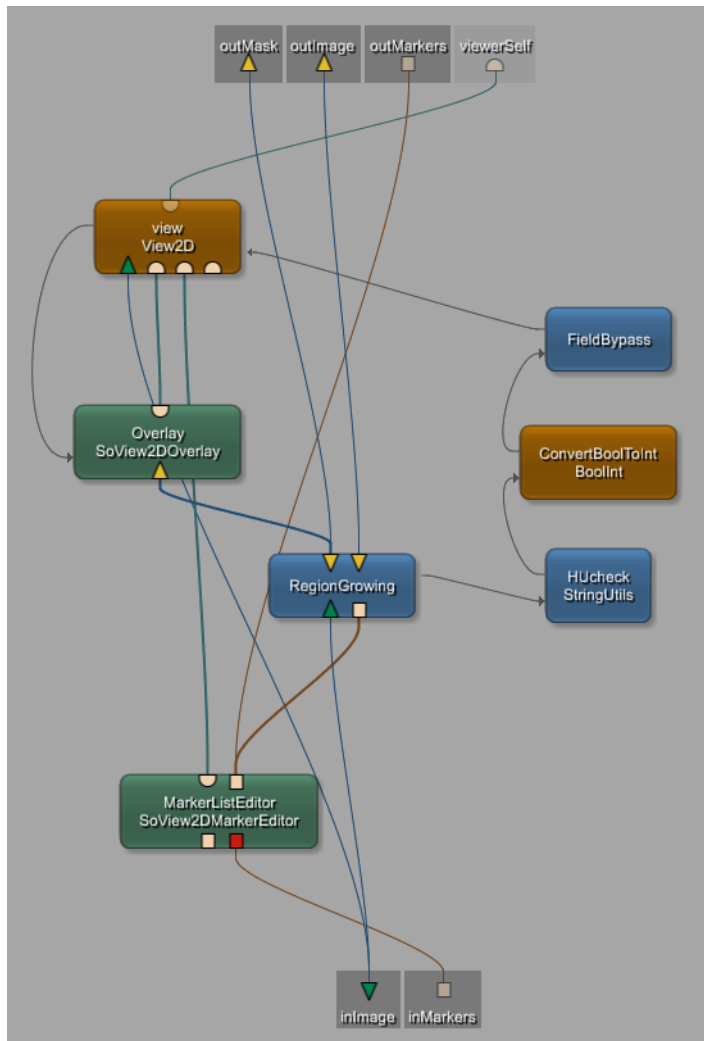
Read Section 2.9, “FieldListener” for information about the use of explicit field listeners for Python scripting.

2.5. 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 internal network of the `RegionGrowingMacro` 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 9, Introduction to Macro Modules](#) for more information).



Tip

For information on the involved classes for the programming of modules, connectors, and connections, see [Chapter 13, Introduction to C++ Modules](#).

2.6. 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 the network's modules, their settings, their connections, and module groups.
.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 Section 10.2, “Adding the Macro Parameters and Panel” for an example on GUI programming.
.mhelp	File with descriptions of all fields and the use of a module. See MATE as a Module Help Editor in the MeVisLab Manual.
.py	Python file, used for scripting in macro modules. See Chapter 10, Developing a Macro Module for an Applicator for an example on macro programming.
.dcm	DCM part of the imported DICOM file, see Section 12.7, “Data Types for DICOM and TIFF” .
.tiff	TIFF part of the imported DICOM file, see Section 12.7, “Data Types for DICOM and TIFF” .
.mlimage	6D image saved with all DICOM tags, lossless compression, and in all data types.
.gvr	Precomputed octree file for direct volume rendering.

For files related to module programming in C++, see [Chapter 13, Introduction to C++ Modules](#).

2.7. 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, etc.

To these controls, scripting can be added.

An example for the programming of a small module panel is given in [Section 10.2, “Adding the Macro Parameters and Panel”](#).



Tip

See the `ExampleGUIScripting` module or other example modules. 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.

2.8. Scripting

MeVisLab offers scripting interfaces. The scripts can be implemented in Python.

Scripts can be triggered by field listeners or user interface controls.

The trigger source defines the “context” of the script execution, i.e., the set of objects accessible by the script code.

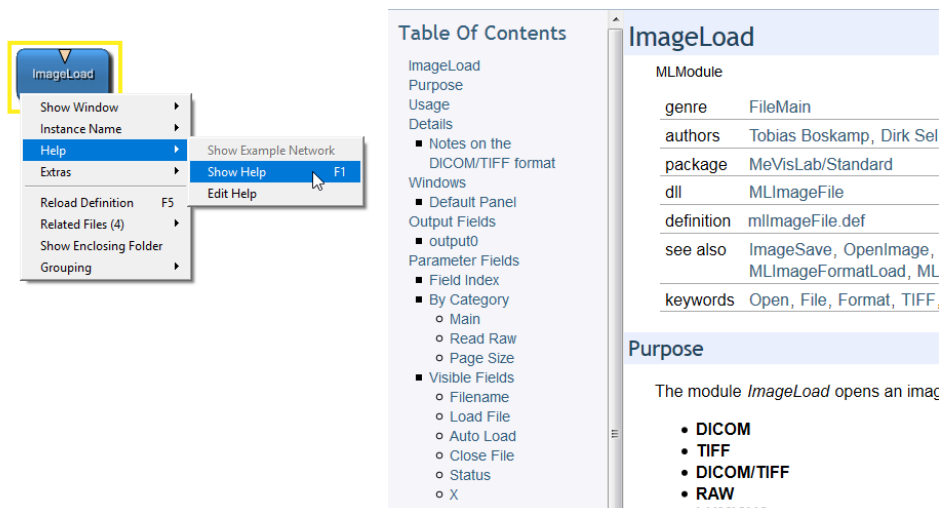
- Modules, fields, connections
- User interface controls, windows
- Wrapped C++ objects like ML images, CSOs, or markers

Global objects provide access to MeVisLab core and system functionality.

2.9. 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 **Help** → **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 **Help** → **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)

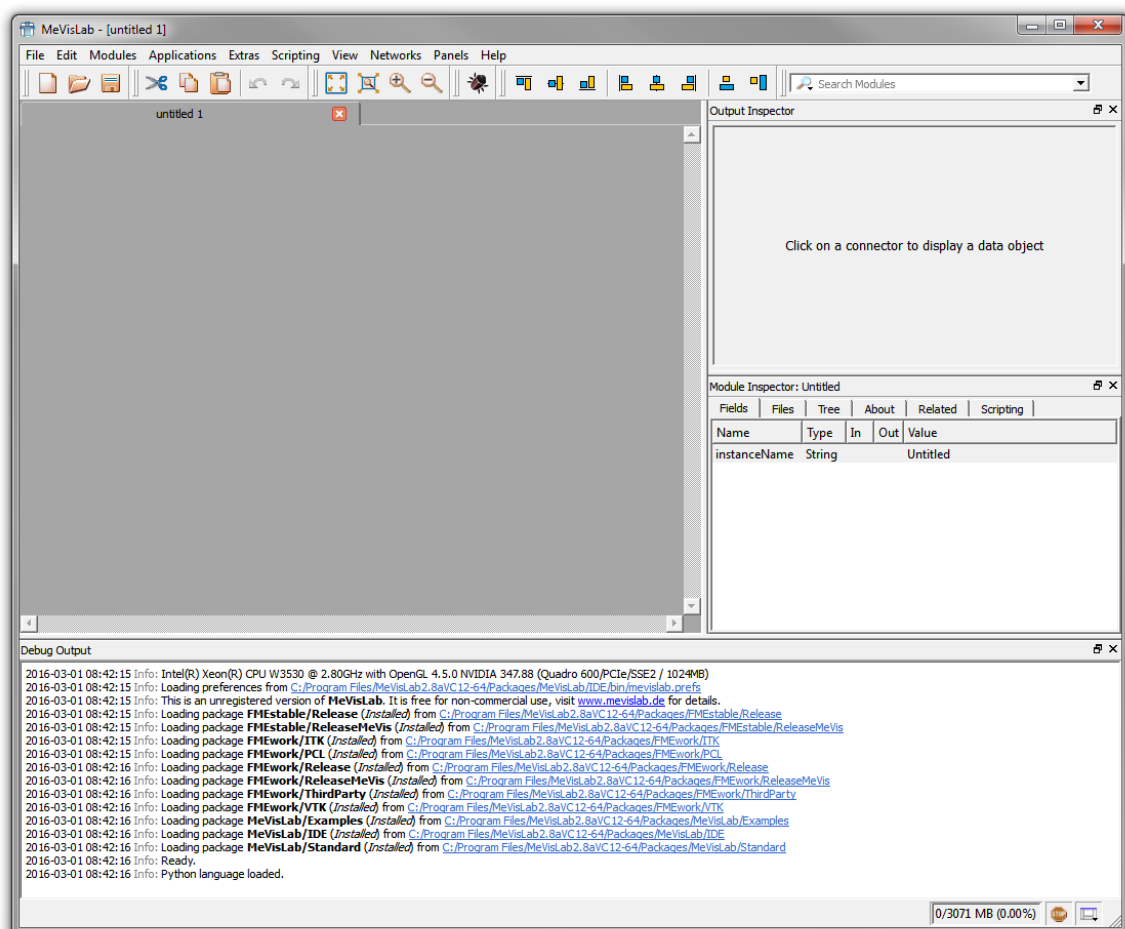
In addition, two special topics are discussed:

- [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.

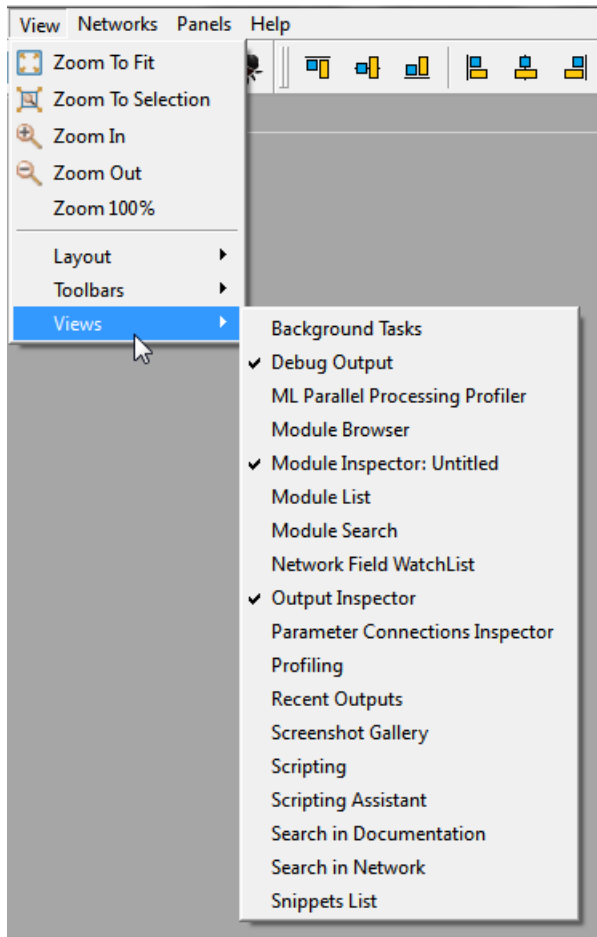
Figure 3.1. MeVisLab User Interface



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 whether 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. View Selection



Some View arrangements 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 “Layout”.

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**.

3.2. Searching and Adding Modules

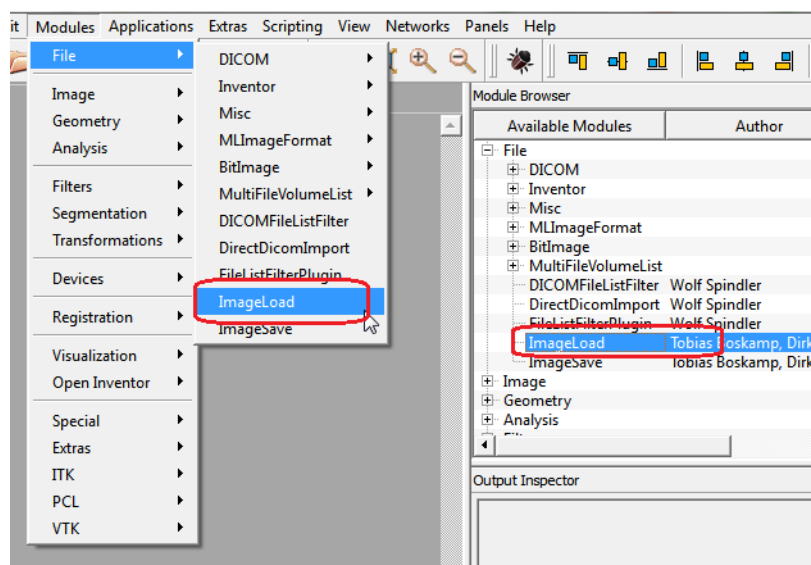
There are several ways to add a module to the current network, for example:

- 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 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

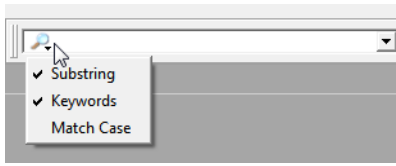
For a module to get listed, it has to be available in the SDK distribution or in your user-defined packages. If in doubt or missing something, check out the loaded packages in the Preferences (on Windows and Linux: **Edit** → **Preferences** → **Packages**; on Mac OS X: **MeVisLab** → **Preferences** → **Packages**). For details on packages, see [Chapter 8, 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.

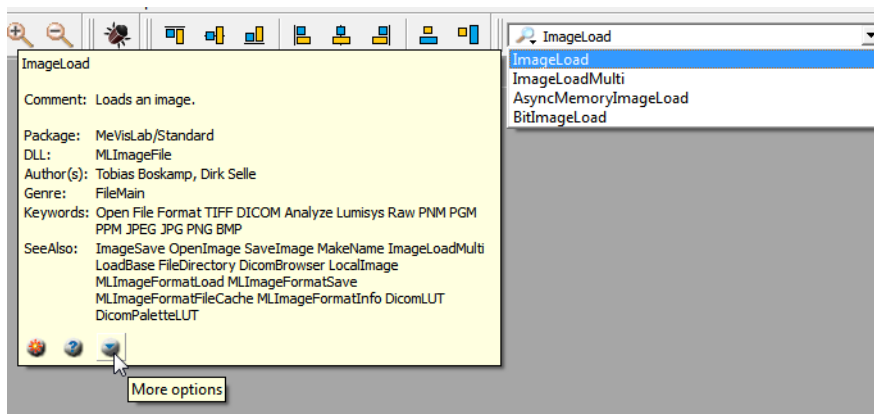


Tip

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.

On the left-hand side of the bottom of the tooltip, you will find three buttons that you can click.

- Adds the module to the network. If no network exists, a new network is added before adding the module.
- Shows the help file for the module in a browser.
- Opens a context menu with further options.

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



Note

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 MRI scan of a head (`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 3.5.2, “Using the LocalImage Module”](#)).



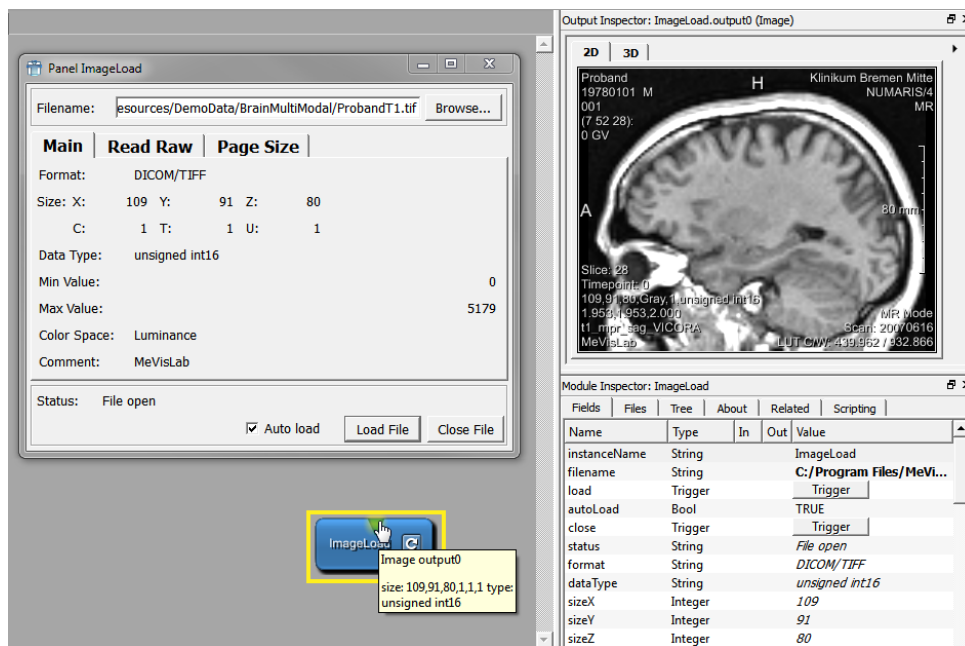
Tip

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.
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**.

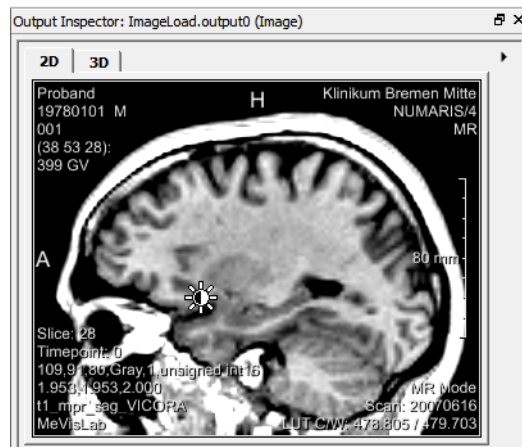
The great thing about the **Output Inspector** is that it will display the output of any connector (or data connection) in the process chain (as long it is a format the inspector can interpret). So if you are ever unsure about what is actually the input or output of a module, simply click the connector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

Figure 3.7. ImageLoad Panel and Output Inspector

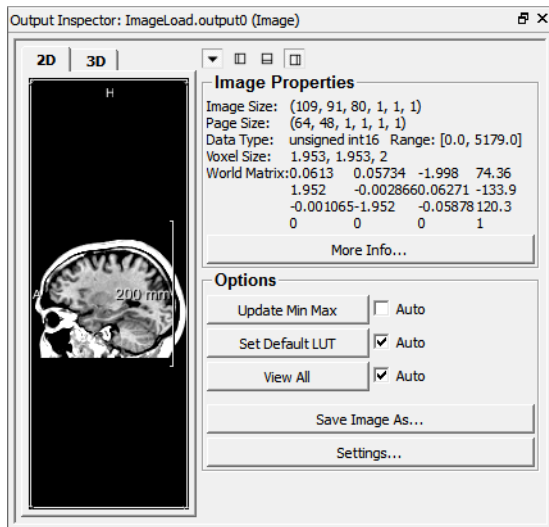
Your image does not look like this? One reason might be that the slice of the image you are looking at has no information. Click on the **Output Inspector** and scroll through the slices by

- using the mouse wheel
- keeping the middle mouse button (mouse wheel) pressed and moving the mouse up and down
- pressing the arrow keys Up or Down (Left or Right slice through time points)

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 Window/Level

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 12, Excursion: Image Processing in ML](#) for more information):

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

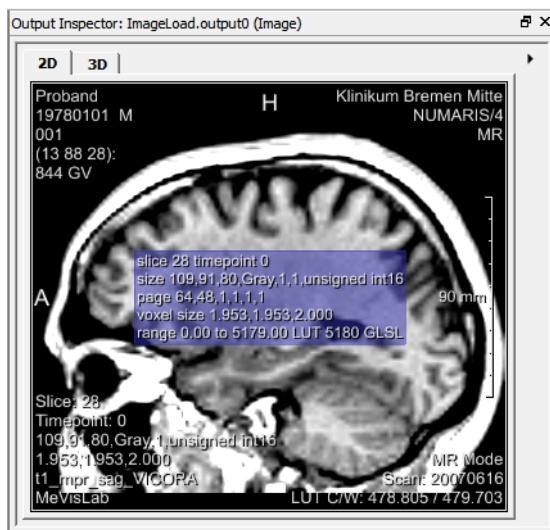
A number of options are available:

- **More Info...:** opens the panel of an Info module showing additional information about the image.
- **Update Min Max:** scans in the input image for the real min/max values. Also resets the LUT on base of the new min/max values.
- **Set Default LUT:** sets the LUT on base of the image's min/max values or on stored DICOM tags if available.
- **View All:** centers the rendered image in the 3D view, has no effect in the 2D view.
- **Save Image As...:** Saves the image to disk.
- **Settings...:** Shows the panel of the used 2D viewer. Has no effect on the 3D rendering.

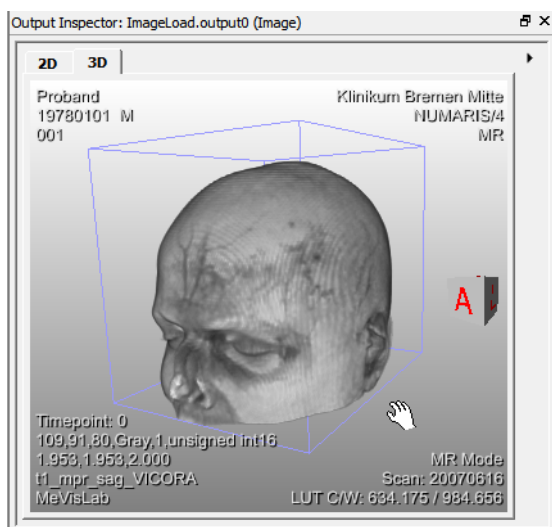
The layout of the Output Inspector's viewer and control panel can be adjusted.

In addition, two key shortcuts are available:

- **A:** Toggles the display of the annotations.
- **I:** Toggles 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

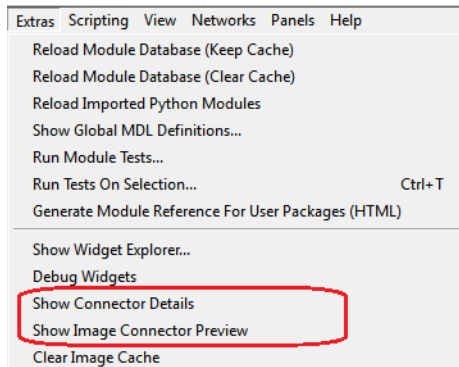
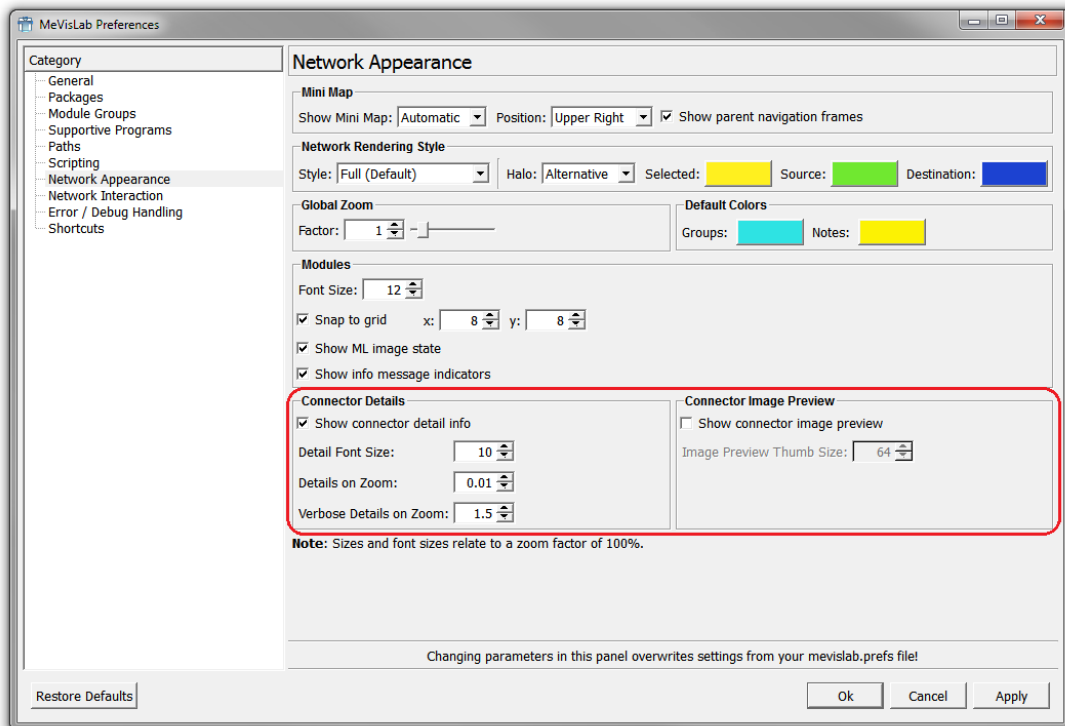
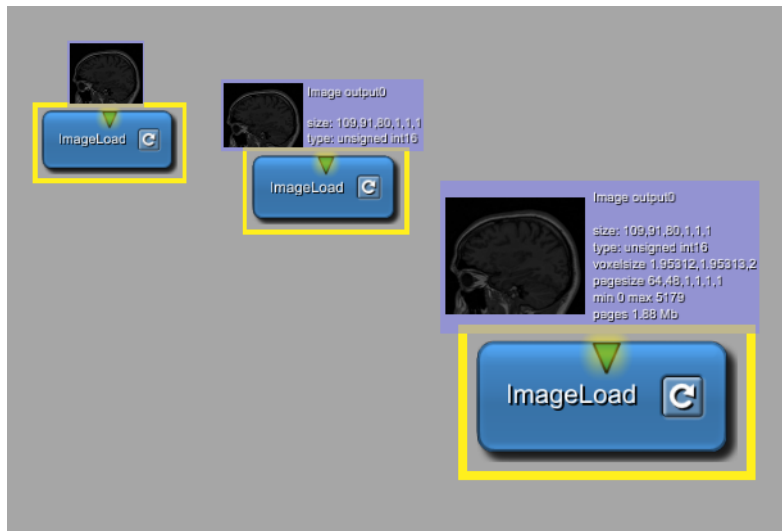


Figure 3.13. Connector Details in the Preferences



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

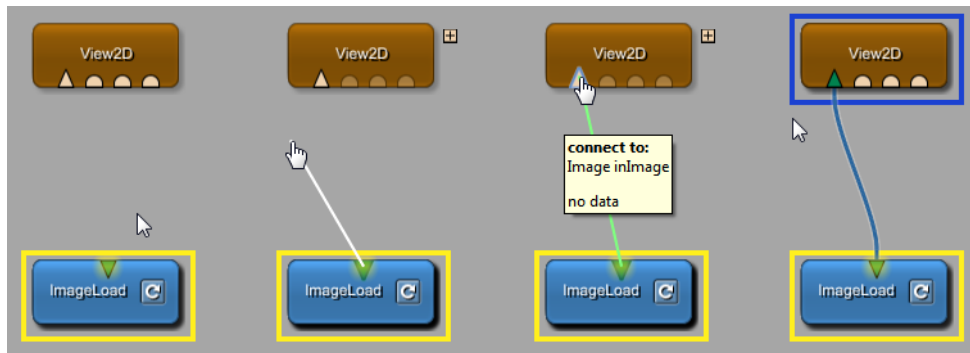
Instead of using the *Output Inspector* (whose display might change with every clicked connector), it is sensible to add a viewer to the network. There are two standard macro modules available in MeVisLab which provide standard viewer configurations for 2D and 3D rendering, namely *View2D* and *View3D*. 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**.

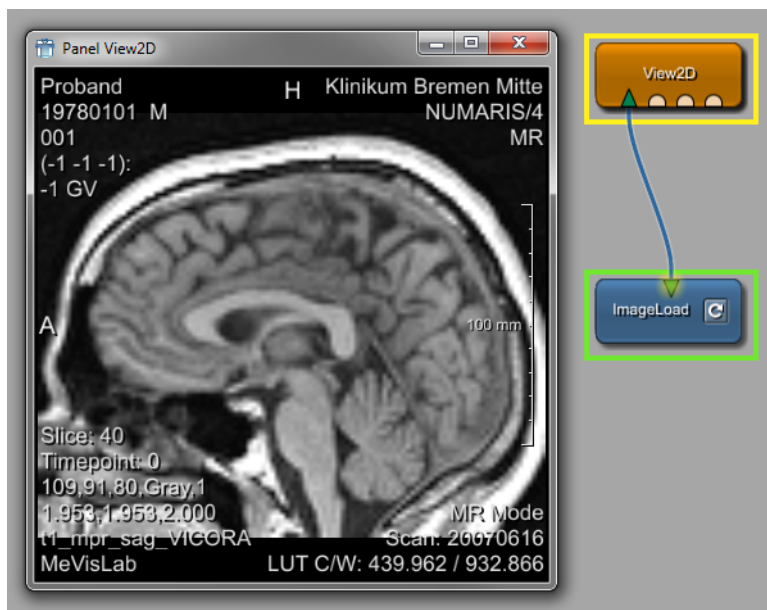
The *View2D* module has one input connector for the image to be rendered, as well as three Inventor inputs.

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**Tip**

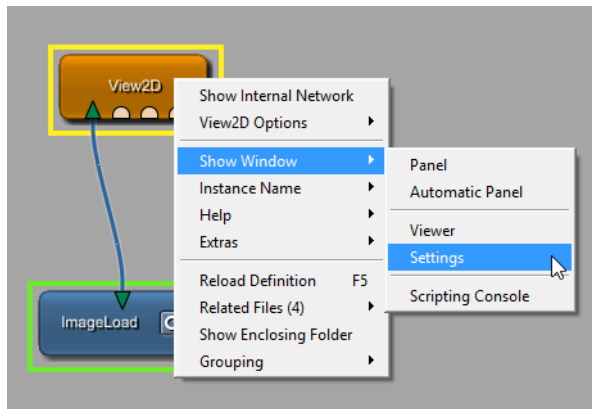
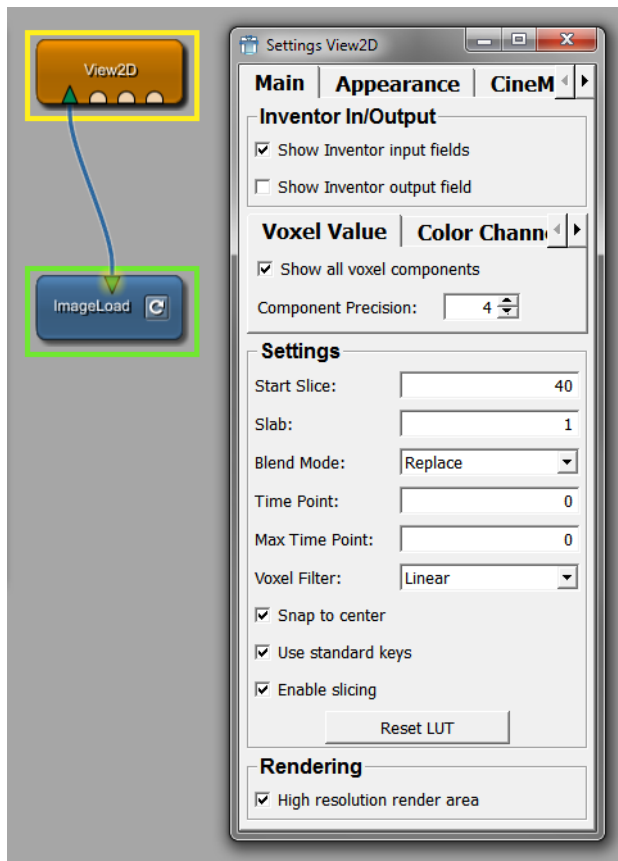
There are many more ways to connect and to disconnect modules, see Section 3.4, "Connecting, Disconnecting, Moving, Copying, and Replacing Connections".

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.

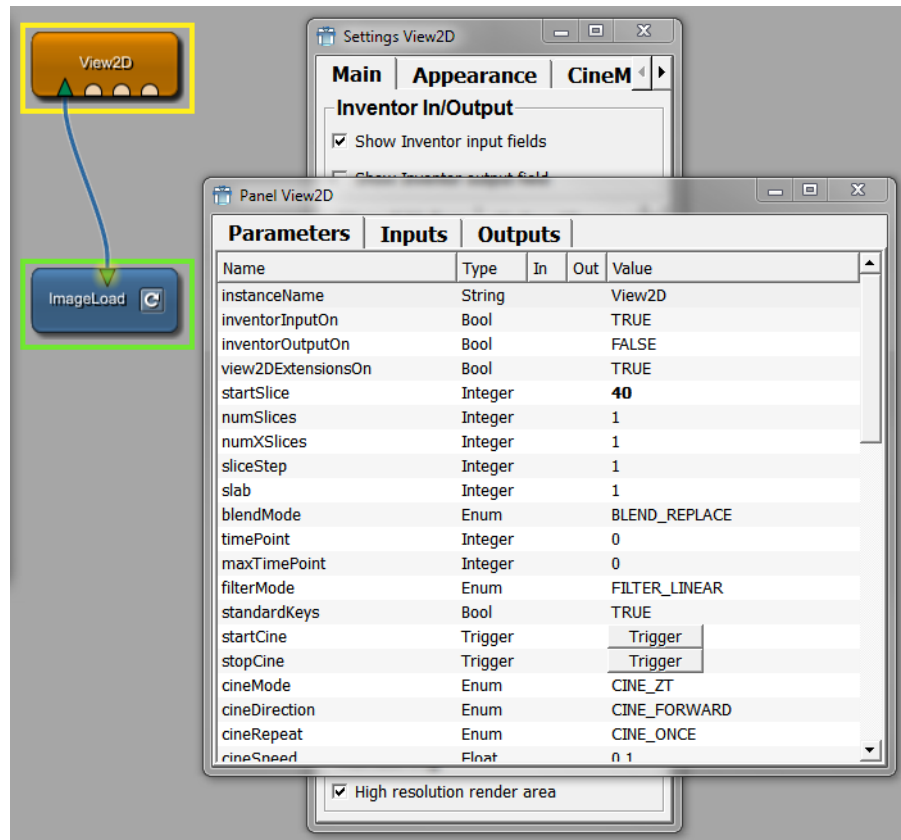
Figure 3.16. Panel of View2D

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****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 2.7, “User Interfaces Controls”](#)) or more controls than fields.

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 how to get a list of all shortcuts, 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**

The most recent network file can be opened via **File** → **Open Most Recent File** which has an own keyboard shortcut.

**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 the networks 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

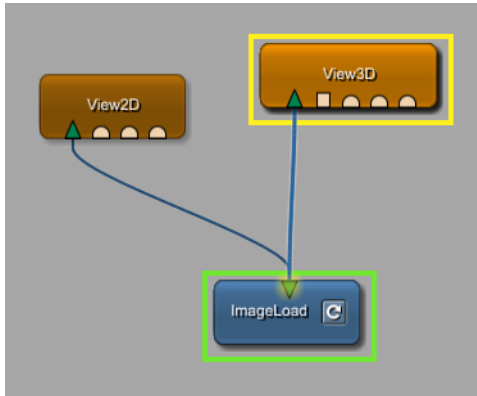
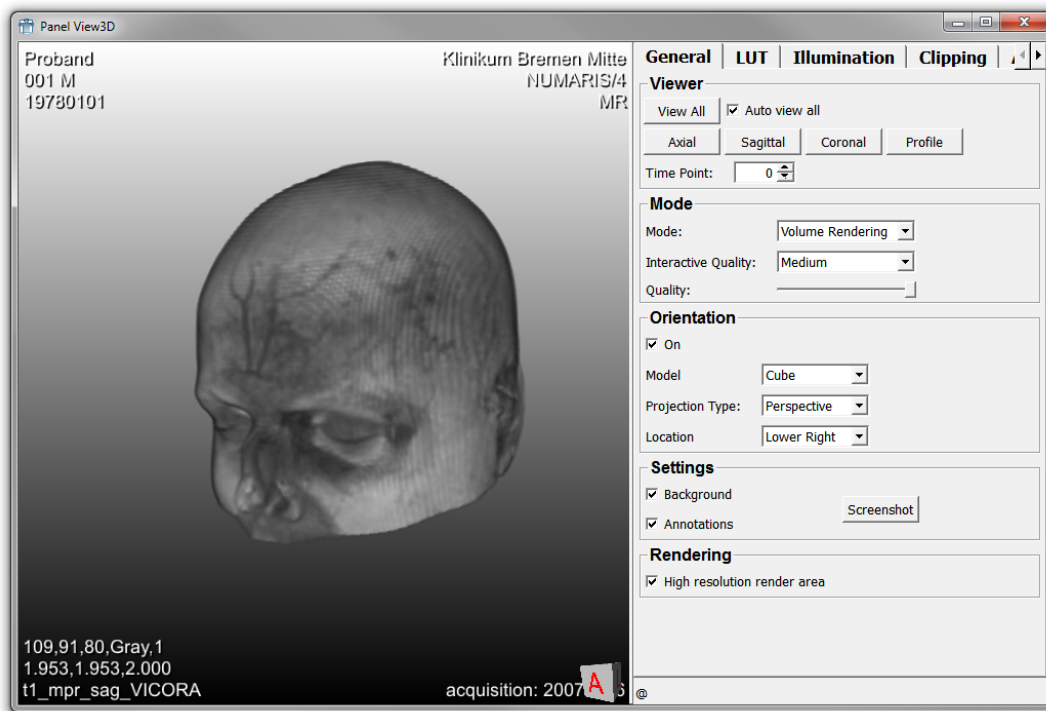


Figure 3.21. The View3D Panel



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).

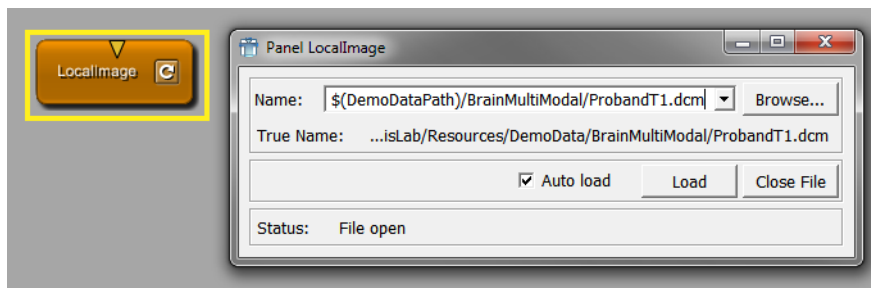
- onto an existing `ImageLoad` module
- onto the filename field of an existing `ImageLoad` module

3.5.2. Using the LocalImage Module

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

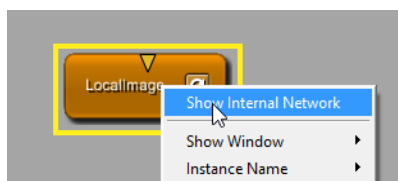
`LocalImage` is a macro module that allows for image selection based on relative paths. This method is recommended for image referencing because it enables an easier exchange of networks between cooperating parties. The list of supported variables can be seen when using the drop-down box of the input widget.

Figure 3.22. LocalImage Module

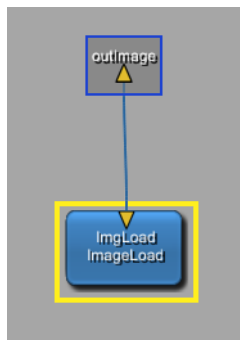


Macro modules are a combination of an internal network and a script. You can open the internal network via the module's context menu or by pressing **SHIFT** and double-clicking the module. Alternatively, the internal network can be opened in the preview state of a network (see the MeVisLab Manual).

Figure 3.23. 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 — a feature that has become somewhat obsolete by the introduction of the `isFilePath` attribute on string fields, which accomplishes roughly the same without the need for extra code.

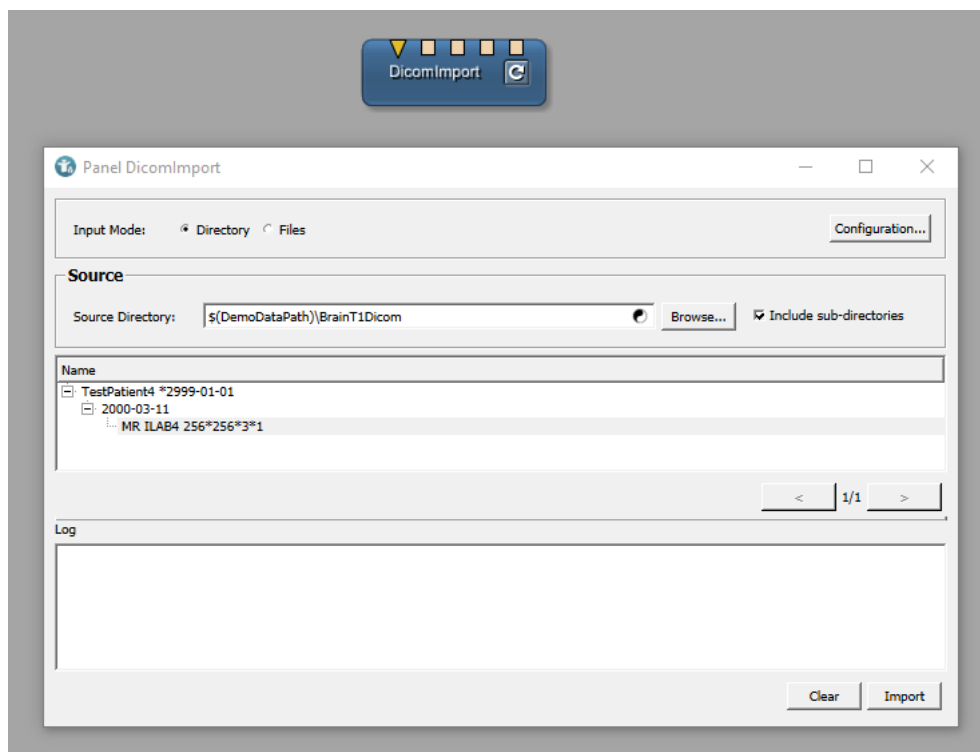
Figure 3.24. Internal Network of the LocalImage Module

3.6. A Note on Importing DICOM Images

Without importing your DICOM slices, the standard MeVisLab image loading modules like `ImageLoad` will only be able to load single DICOM slices separately. For further information, see the chapter [Chapter 12, Excursion: Image Processing in ML](#).

The DICOM import is mainly provided by the modules `DicomImport` and `DirectDicomImport`. The use of `DicomImport` will be described here:

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 3.25. DicomImport

2. Enter the necessary data.
 - a. Select the **Source Directory** where your DICOM slices are located. In the MeVisLab installation path you can find some example DICOM slices in the directory `$(InstallDir)/Packages/MeVisLab/Resources/DemoData/BrainT1Dicom`. All subdirectories will be scanned recursively by default.

- b. Click the **Import** button. The lower part of the module panel will show error messages (if there are any).

Most of the import process happens asynchronously. When the progress bar at the bottom of the module panel disappears the import has finished. The area above the error message area will contain the list of imported patients, which can be expanded to show studies, series, and finally image volumes. You can click on the volume entries, which will be provided at the first and second output connector of the module. (The second output only contains the combined DICOM tree, while the first one also provides the image volume.) By default no entry will be selected.

You can now connect any module that processes images (or DicomTree objects) to the module.

If your DICOM import fails, or doesn't provide the expected results, check the settings of the module by clicking the **Configuration...** button, especially check the sections **Sort/Part**. You should probably also consult the help page for this module which is available through the context menu of this module via **Help** → **Show Help**.



Tip

DICOM multi-frame files can be opened directly in MeVisLab through the `ImageLoad` module; therefore, the use of `DicomImport` is not absolutely necessary for displaying the data. `ImageLoad` will not split or re-arrange the frames in a multi-frame file, though.

DICOM files without image data can also be opened with `LoadDicomTree`.



Note

MeVisLab has its own 3D file format which stores the image values and the image DICOM tags in a file with the file extension `.mlimage`, which can be stored with `MLImageFormatSave` and loaded with `MLImageFormatLoad`.

There is also an older format that stores image and tags separately in two files with the same base file name but different file extensions: `<filename>.tiff` and `<filename>.dcm`. These pairs can be loaded with `ImageLoad`

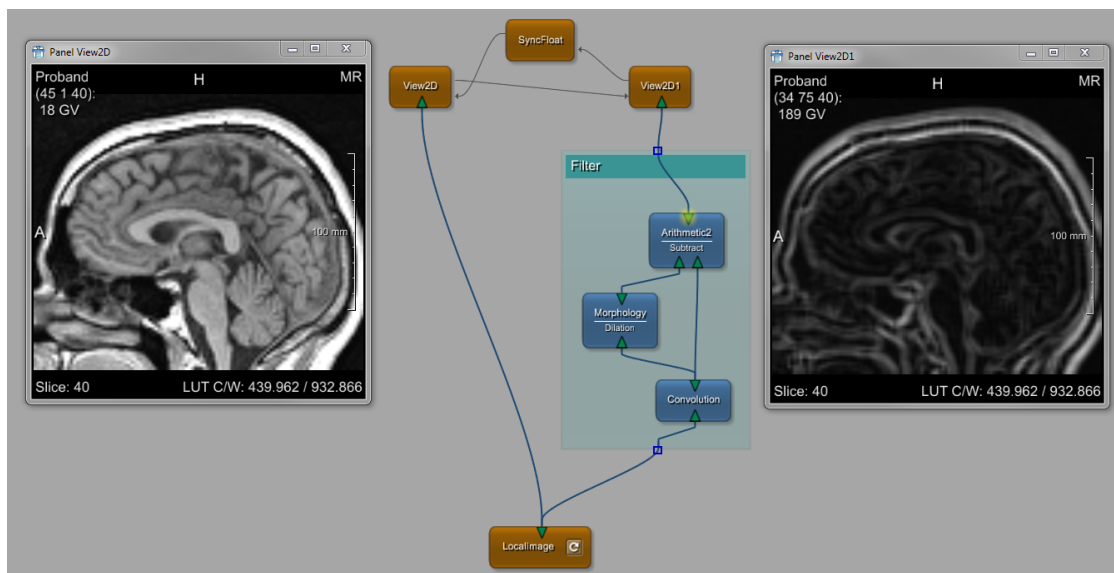
Chapter 4. Implementing a Contour Filter

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.

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

This will be our resulting network:

Figure 4.1. Example Network Contour Filter



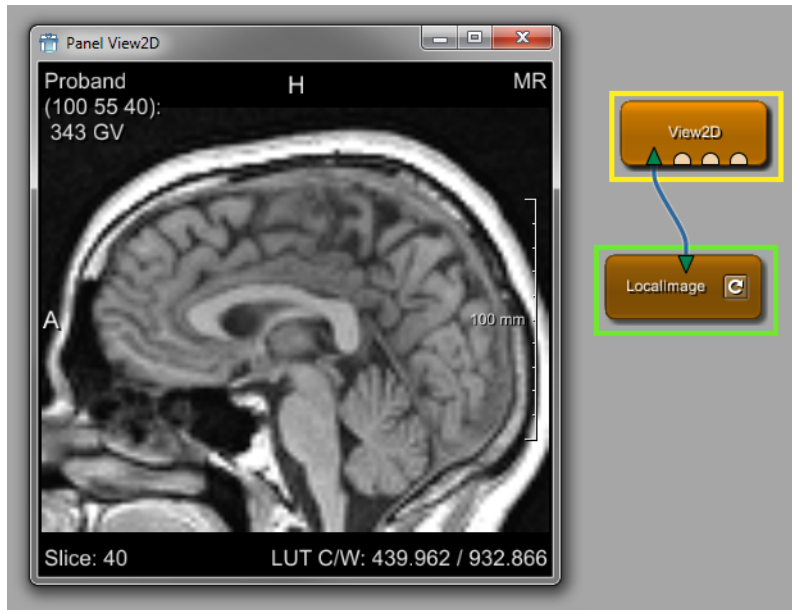
Note

In this example, the Inventor inputs of the `View2D` modules are hidden by unchecking the context menu option **View2D Options** → **Show Inventor Inputs**. For more information, see the MeVisLab Reference Manual, chapter “Additional Inputs”.

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. The default image loaded by `LocalImage`, ProbantT1 is fine.
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. Show 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:

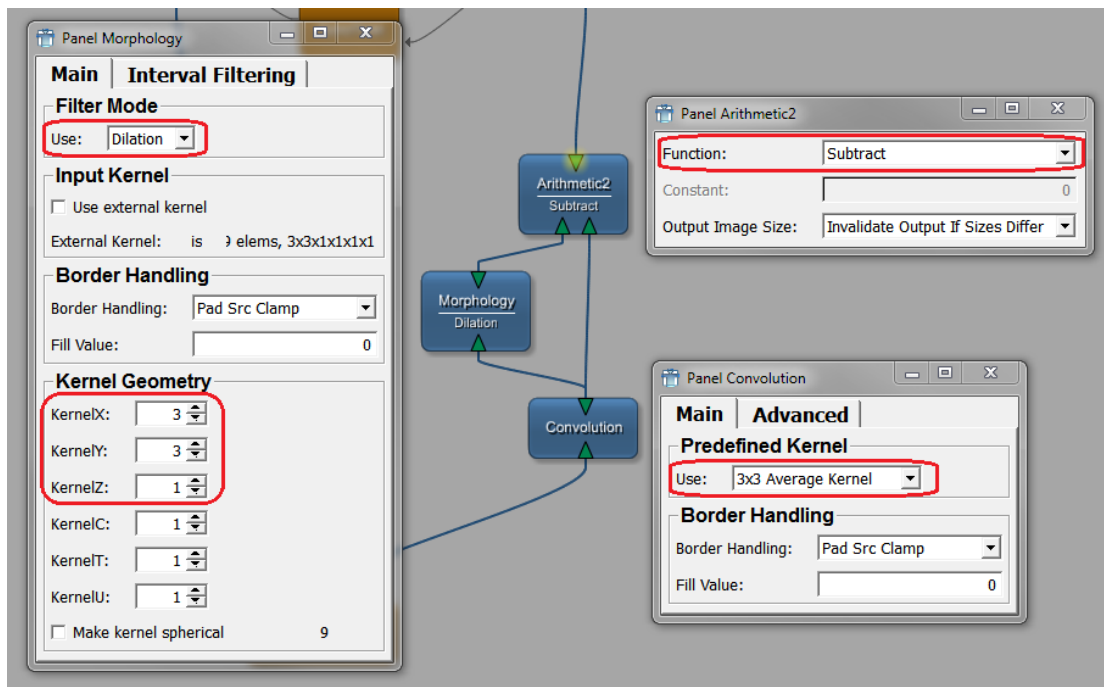
- a. via **Modules** → **Filters** → **Kernel** → **Convolution**,
- b. via **Modules** → **Filters** → **Morphology** → **Morphology** and
- c. via **Modules** → **Analysis** → **Arithmetic** → **Binary** → **Arithmetic2**.

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 3x3x1.
 - 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.

3. 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

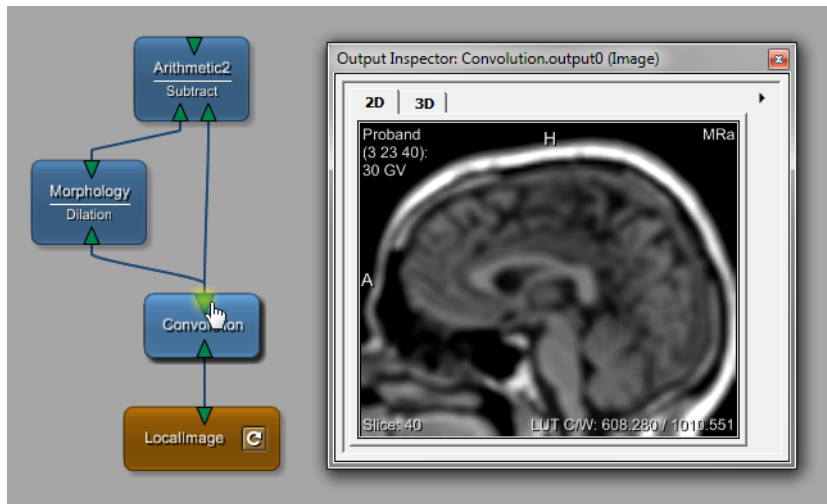


Figure 4.5. Constructing the Filter Pipeline — Morphology Output

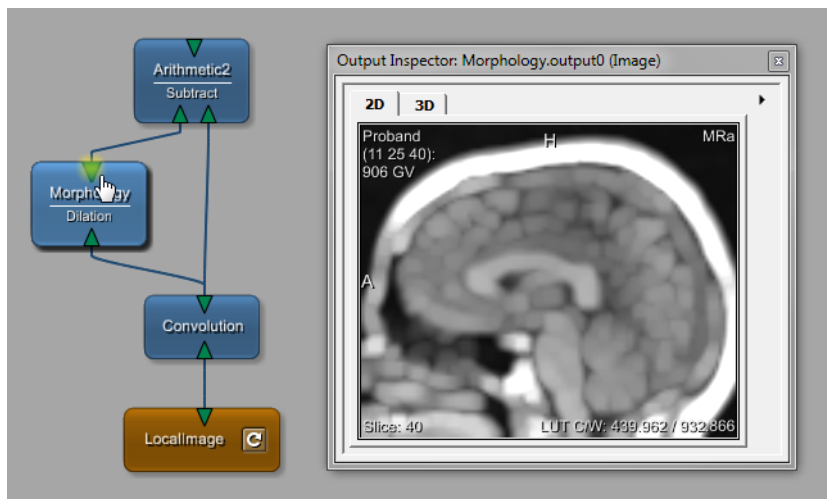
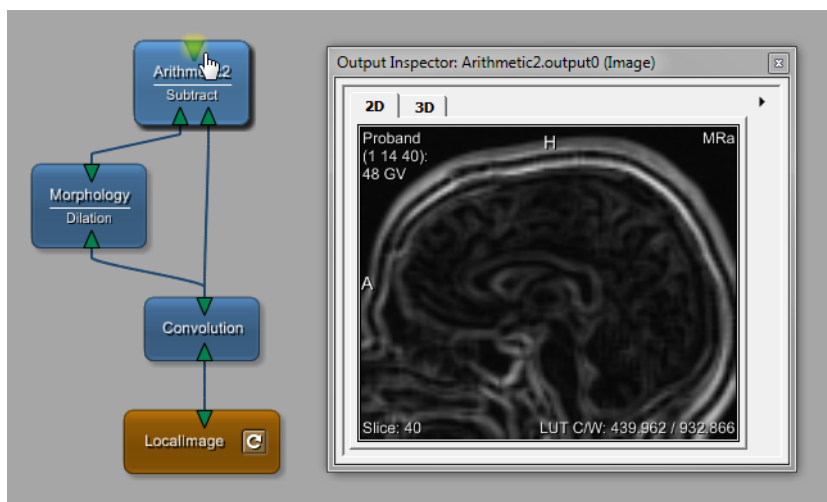


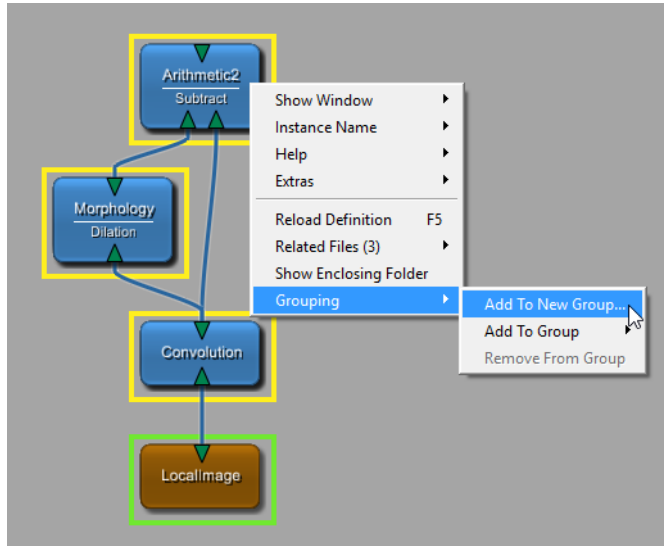
Figure 4.6. Constructing the Filter Pipeline — Arithmetic2 Output



4. To distinguish the image processing pipeline, you can create a group for it. For that:

- Select the three modules, for example by dragging a selection rectangle around them, or by single-selecting the modules while pressing **SHIFT**.
- Right-click the selection to open the context menu and select **Add to New Group**.
- Enter a name for the new group, for example “Filter”.

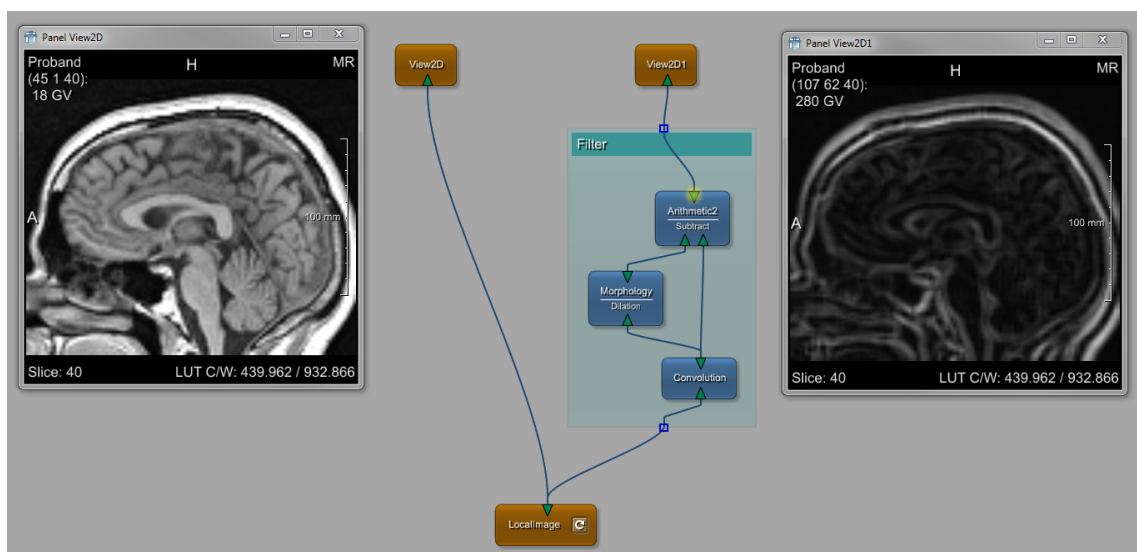
Figure 4.7. Creating a New Group



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, chapter “Using Groups”.

- 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) it is also possible 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.)



Note

MeVisLab prevents the creation of infinite loops between fields in most cases. A notable exception is a loop between Inventor fields when ML or macro interface fields constitute intermediate fields. In this case the loop cannot be detected and - once triggered - will lead to a background computational load. This can be avoided by using the `SyncFloat` or `SyncVector` modules (see “Using SyncFloat to Reduce System Load”) or by using scripting to only propagate real value changes.

- 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. Add a `SyncFloat` module to the network and open its panel with a double-click.

2. 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.
3. 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 gray arrow with the arrowhead pointing to the module that receives the parameter as input.
4. In the other direction drag the `startSlice` field from the `SoView2D1` panel to the `float1` field of the `SyncFloat` panel, and from the same panel the `float2` field to the `startSlice` field of the `SoView2D` panel. The intermediate `SyncFloat` module breaks the inevitable notification loop by only triggering the second connection at real value changes.



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

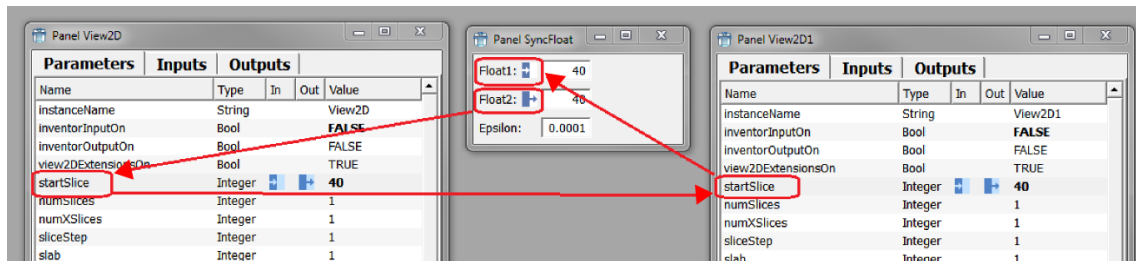
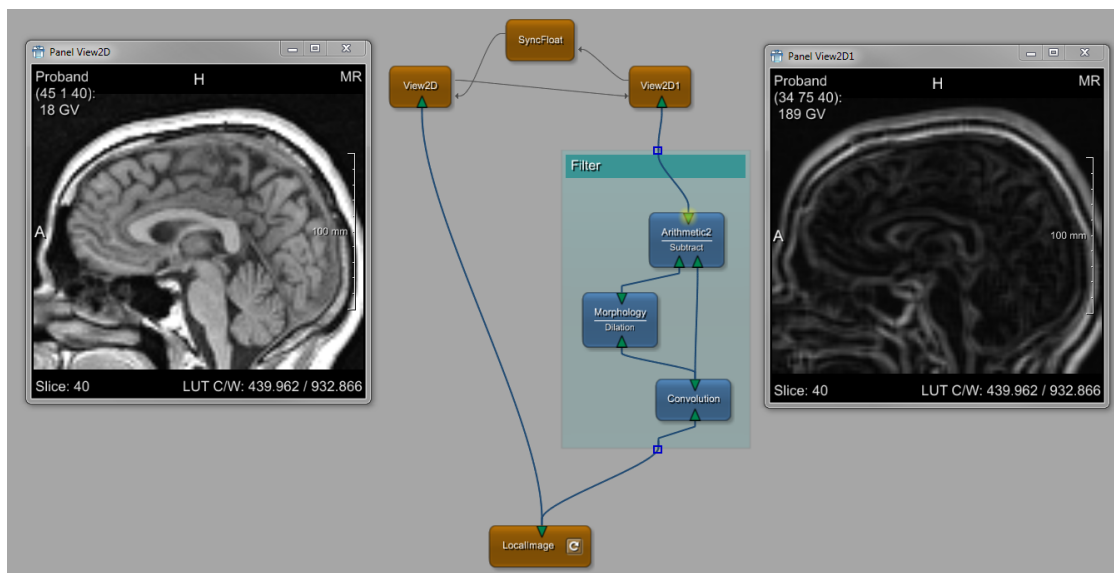


Figure 4.10. Resulting Network



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** → **more...** → **ContourFilter.mlab**).

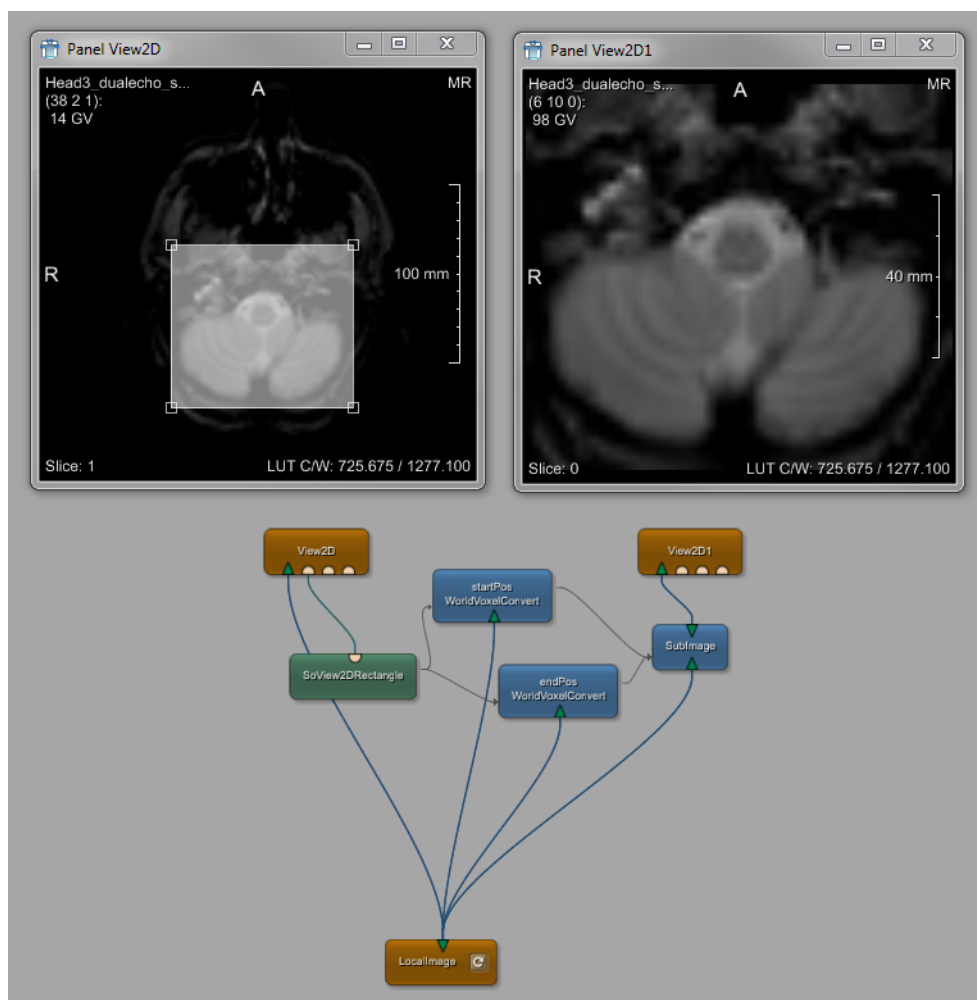
Chapter 5. Defining a Region of Interest (ROI)

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”](#)
- [Section 5.2, “Adding a Second Viewer for the Subimage”](#)
- [Section 5.3, “Adding the Interactivity for 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 12, Excursion: Image Processing in ML](#).

5.1. Creating a Viewer with a Selection Rectangle

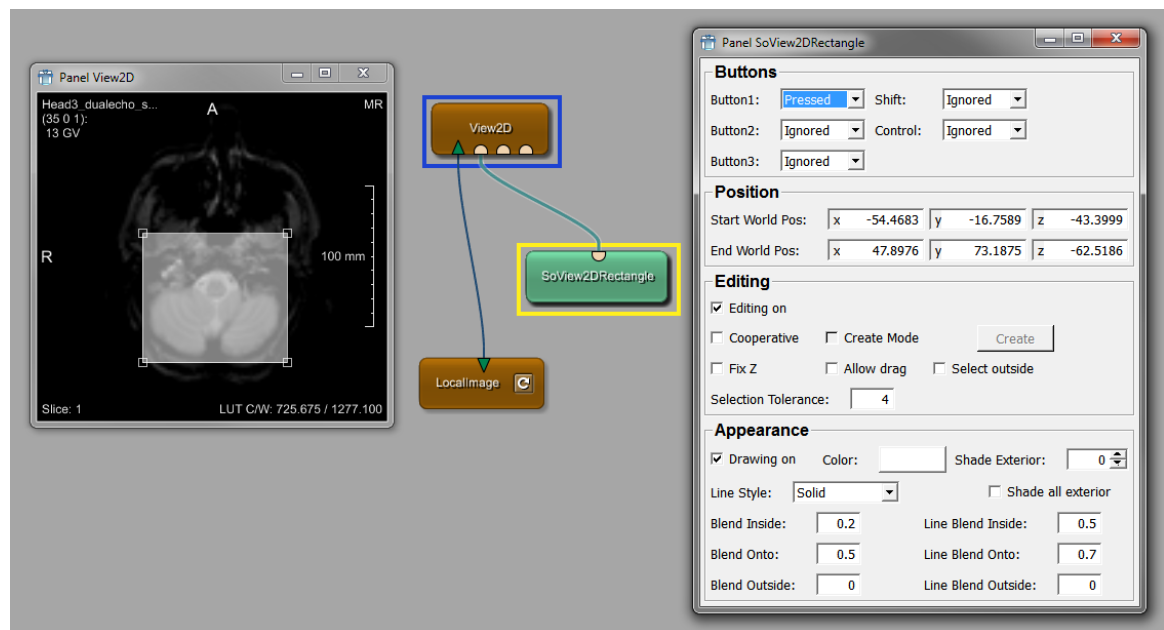
The first part is building a simple network with an image load module, a viewer, and a module that allows for drawing a selection rectangle.

1. Add `LocalImage` and the `View2D` module to the new network and connect their image connectors.
2. Double-click on `LocalImage` to open the panel, and select the image `Head3_dualecho_small.dcm` for this example. Load the image.
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.)

Double-click on `SoView2DRectangle` to open 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

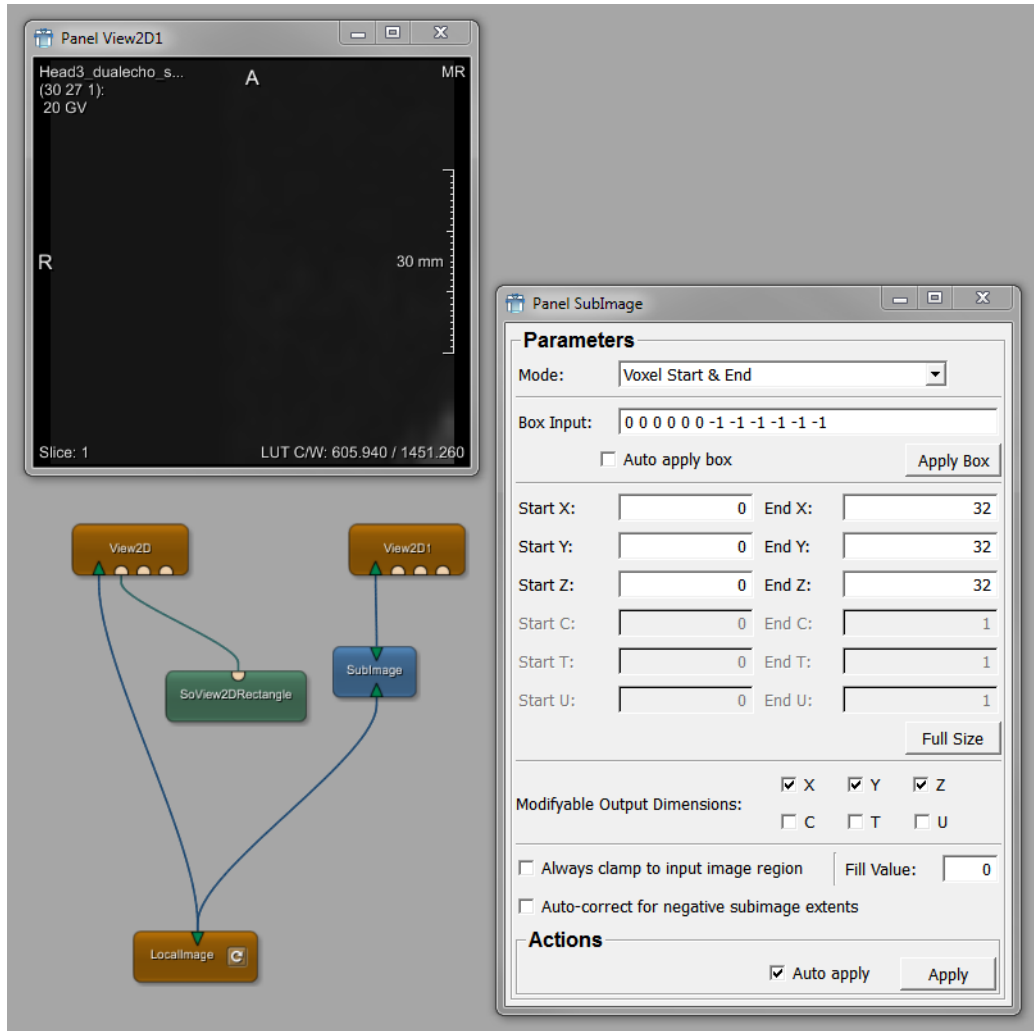
Add the second viewer part, which consists of two modules:

- 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.”

On the `SubImage` module, check the option **Auto apply** so all changes to the module's parameter take an immediate effect. Also, set the module's **Mode** to "Voxel Start & End", because we will use the start and end voxel position of the interactively drawn rectangle to define the subimage.

Figure 5.3. Viewer for the Subimage



Just leave **X**, **Y**, and **Z** as **Modifiable Output Dimensions**; uncheck **T** here.

We have not yet defined how the world positions of `SoView2DRectangle` are connected to the subimage, so the current subimage is rather random, depending on the initial parameter state of the `SubImage` module.

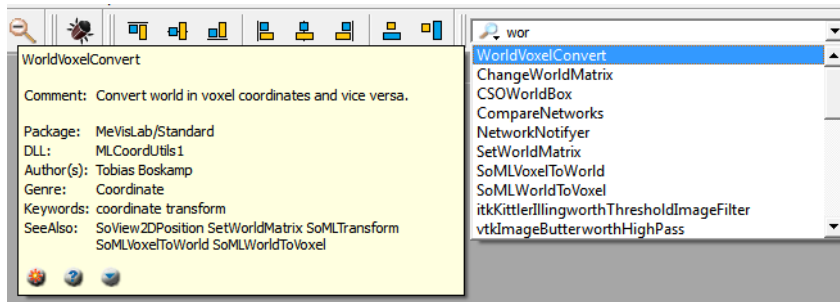
5.3. Adding the Interactivity for the Viewers

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.

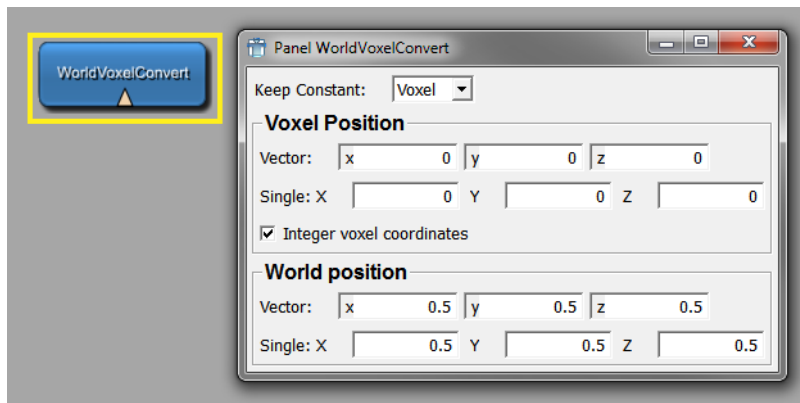
1. As we need world and voxel, enter those words in the quick search to find the module:

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.

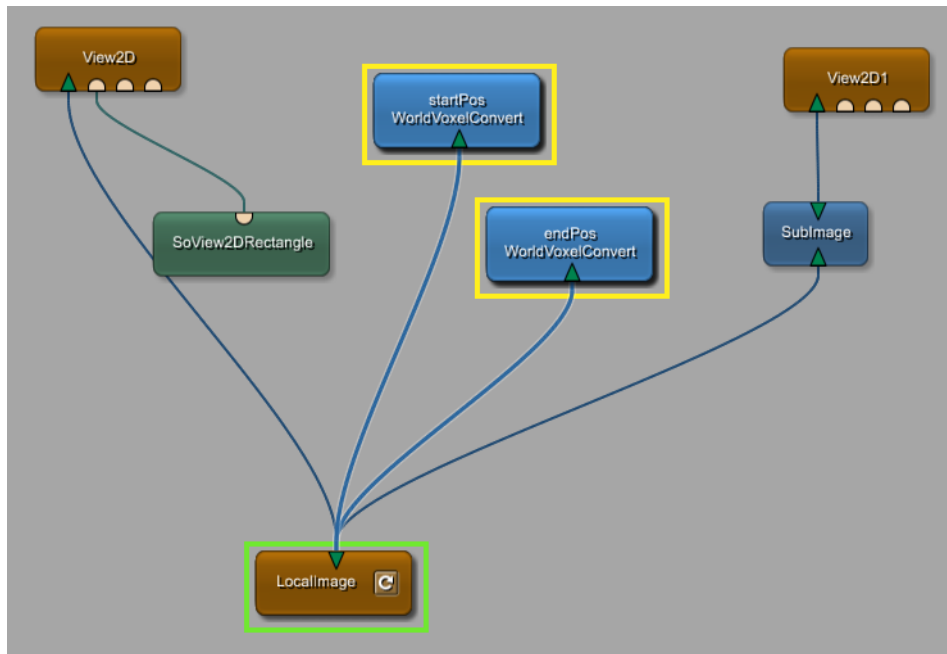


Tip

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

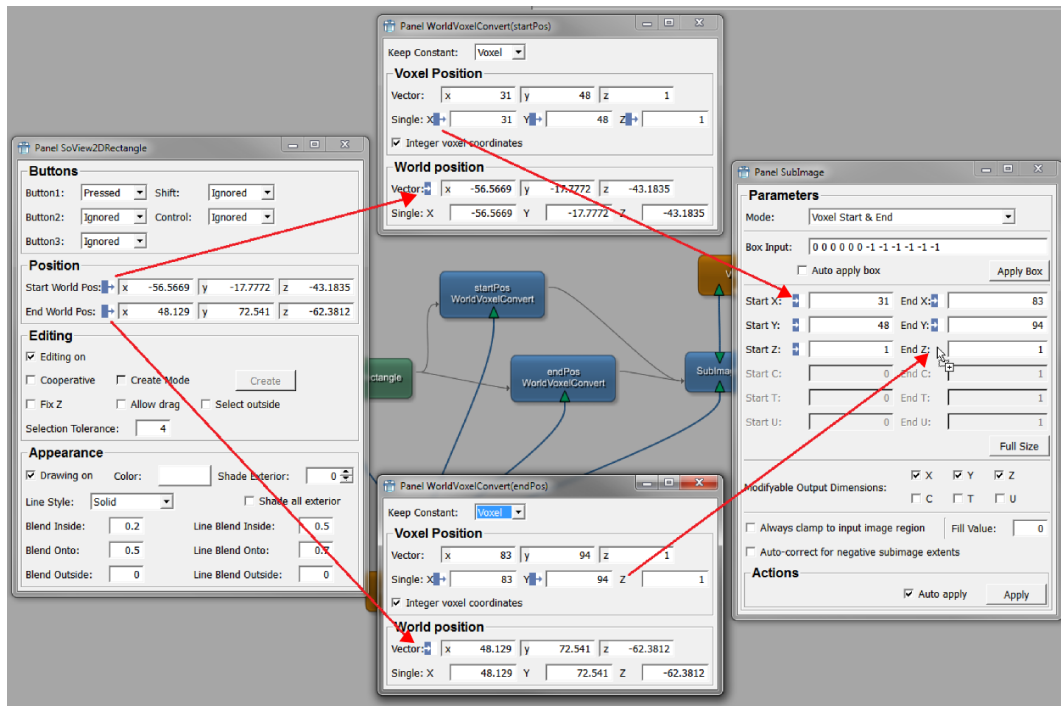
4. In both `WorldVoxelConvert` modules, check the option **Integer voxel coordinates**.
5. Both `WorldVoxelConvert` modules need the original image for obtaining the world-to-voxel matrix, so connect them to `LocalImage` (the image output can be connected to an unlimited number of modules).

Figure 5.6. WorldVoxelConvert Modules Added



6. For the parameter connections, proceed as follows:
- Connect the `SoView2DRectangle` **Start World Position** to the `WorldVoxelConvert(startPos)` **World Position Vector**.
 - Similarly, connect the `SoView2DRectangle` **End World Position** to the `WorldVoxelConvert(endPos)` **World Position Vector**.
 - 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.
 - 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.

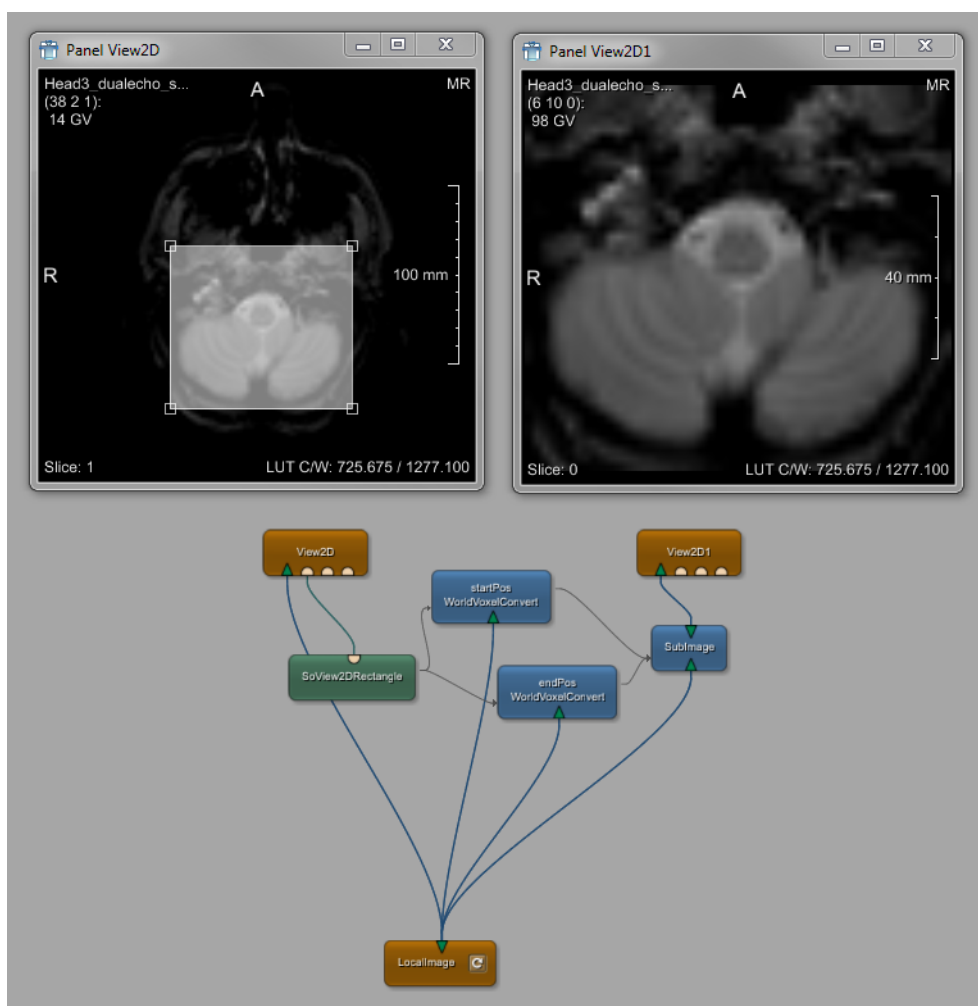
Figure 5.7. Adding the Parameter Connections



7. If you have not done that before, 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** → **more...** → **RoiExample.mlab**).

Chapter 6. Excursion: Functionality Overview

In the following chapter, we will list a number of typical modules for typical questions, with brief information on their main purposes.



Tip

To learn about module details, read the module help and have a look at the example network.

- [Section 6.1, “Image Handling and Processing”](#)
- [Section 6.2, “Visualization”](#)
- [Section 6.3, “Data Objects”](#)
- [Section 6.4, “Miscellaneous”](#)

6.1. Image Handling and Processing

6.1.1. Image Handling

- ImageLoad: opens an image file stored in one of the following formats: DICOM, TIFF, DICOM/TIFF, RAW, LUMISYS, PNM, Analyze, PNG, JPEG.
- LocalImage: works like ImageLoad, but loads images relative to the network or the local MeVisLab installation.
- ImageSave: saves an image to file using one of the following image file formats: DICOM, TIFF, DICOM/TIFF, RAW, LUMISYS, PNM, Analyze, PNG, JPEG.

6.1.2. Image Properties

- Info: shows information about the currently connected input image, like image size, page size, voxel size, total volume, world matrix, etc.
- MinMaxScan: scans the input and updates the minimum and maximum values of the output image. The data type can be adapted, left unchanged or set to an arbitrary one.
- ImagePropertyConvert: allows to freely change page size, minimum or maximum value, data type, or world matrix of an image.
- ImageStatistics: computes some statistics of the input image voxels (subset of voxels).

6.1.3. Basic Image Processing

- SubImage: extracts subimages from an input image based on either voxel start/size, voxel start/end or world start/end. Can also be used to create a region larger than the input image.
- Resample3D: resamples an image in 3D on an arbitrary planar parallel grid. 17 filters are available.
- Reformat: can be used to reformat an image to a reference image, or to create reformatted overlays via SoView2D/SoOrthoView2D.
- Scale: scales the input image to another interval. The source and target scale interval can be defined.
- Arithmetic1: performs arithmetic operations on one image. For example, in case of *Add*, the constant value is added to each voxel of the input image.
- Arithmetic2: performs arithmetic operations on two images. For example, in case of *Add*, the values of input image 2 are added to each voxel of input image 1.
- Switch: selects one of up to 25 input images depending on an input parameter. The selected image is passed unchanged to the output.

- **Mask:** masks the image of input 1 with the mask at input 2. For example, in case of *Masked Original* (default), all pixels from the first input are passed unchanged to the output if non-zero values are found at their positions in the second input image. Otherwise background values are passed to the output.
- **TestPattern:** generates a test image of a defined size, page size, data type and pattern (stripes, checkers, ramps, etc.).
- **AddNoise:** produces noise based on a scalar input image and a chosen distribution, for example uniform noise, Gaussian noise, Salt&Pepper, etc.

6.1.4. Filter

- **Convolution:** offers standard kernel-based filters like Average, Gauss, Laplace or Sobel.
- **ExtendedConvolution:** offers standard convolution filters similar to `Convolution` but with more flexible kernel sizes and kernel geometry.
- **Rank:** offers rank-based kernel filters like Min, Max, Median, Rank or Index.
- **Morphology:** implements dilation and erosion filters. By using threshold intervals, filtering can be applied selectively to regions in the image.
- **CalculateGradient:** computes the slope of image value changes in regions around each voxel in an image.

6.1.5. Segmentation

- **Threshold:** transforms the input image into a binary image, in which voxels below the threshold are set to the image minimum value, and voxels at or above the threshold are set to the image maximum value. Can be used with a relative threshold.
- **IntervalThreshold:** processes an image by filtering just those image values that lie in a certain gray value interval. Voxels outside this range can be set to zero or to a user-defined fill value. This can be useful for the segmentation of objects that are expected to have gray values in a defined interval.
- **RegionGrowing:** provides a simple threshold/interval-based 1D/2D/3D/4D region growing algorithm. A threshold/interval and at least one seed are necessary as start parameters.
- **RegionGrowingMacro:** extends the options of `RegionGrowing` by adding a viewer (`View2D`) and a marker editor to simplify its usage.
- **ComputeConnectedComponents:** performs a connected component analysis on 2D / 3D grayscale images. You need other modules mentioned in the `seeAlso` of the module to process its output.

6.2. Visualization

6.2.1. 2D Viewing

- **View2D:** provides a viewer for viewing a 3D image as 2D slices, with the possibility to scroll through. Annotations are displayed and the LUT of the displayed image can be changed by dragging the mouse with the right mouse button pressed.
- **View2DExtensions:** encapsulates a set of viewer extensions that are commonly used in conjunction with a 2D viewer, including extensions for navigation (browsing through slices, zoom and pan), level/window adjustment, and drawing annotations.
- **SoView2D:** displays a slice (or a slab) of a volume image in a 2D viewer.
- **SoRenderArea:** provides Open Inventor rendering and event handling inside a MeVisLab window. To be useful, the connected scene graph must contain a camera and at least one light source (see the example network).
- **SoView2DOverlay:** blends a 2D image over another one.
- **SoView2DPosition:** shows the last clicked position in a 2D viewer. The style of the displayed position marker can be set to crosshairs, circle and voxel rectangle.
- **SoView2DRectangle:** allows for a drawing and interactive adjustment of a 2D rectangle in a 2D viewer. Although this module is called `SoView2DRectangle`, it actually draws a 3D box.
- **SoMouseGrabber:** grabs mouse events in an Inventor scene and converts them to x, y float fields. The mouse coordinates can be filtered and scaled before the x and y fields are set.

- **SoKeyGrabber**: watches keyboard events in an Inventor scene and triggers Trigger fields depending on the keys pressed, for example on Last Key, SHIFT, CTRL, ALT, etc.
- **OrthoView2D**: provides a 2D view displaying the input image in three orthogonal viewing directions.
- **SoOrthoView2D**: renders orthogonal slices of a volume image in one 2D viewer.
- **SynchroView2D**: provides two 2D viewers that are synchronized via their world coordinates.

6.2.2. 3D Viewing

- **SoGVRVolumeRenderer** (also called Giga Voxel Renderer, GVR): an octree-based render that allows high-quality volume rendering of 3D/4D images. This module is complemented with a set of extension modules that allow to customize the rendering, all starting with the SoGVR* prefix.
- **SoExaminerViewer**: provides Open Inventor rendering and event handling inside a MeVisLab window. Open Inventor rendering attributes such as the background color, transparency type, draw style, etc. can be set.
- **View3D**: allows volume rendering of a 3D dataset. It encapsulates the complex features of the `SoGVRVolumeRenderer` module and provides access to basic rendering features.
- **SoBackground**: renders a color ramp in the background of an Open Inventor scene. The ramp can be flipped and rotated 90 degrees. The module should always be used to give optical depth to an Open Inventor scene.

6.2.3. Lookup Tables

- **ApplyLUT**: applies a lookup table (LUT) to an input image. The voxel values of the input image are used as LUT index values, the LUT entry values are rescaled relative to the Max Entry parameter and stored in the output image.
- **SoLUTEditor**: allows to edit a RGBA Lookup Table and output it as a MLLut object. Also offers an optional histogram display for orientation.
- **SoMLLUT**: provides an ML lookup table (LUT) object to the Open Inventor scene graph.
- **LUTPrimitive**: generates a single-channel, parametrized lookup table (LUT) object that can be used with the `ApplyLUT` module or within 2D/3D viewers (in conjunction with `SoMLLUT`).
- **LinearLUT**: generates a lookup table (LUT) object by interpolating two specified entries. The interpolation is performed in gray (luminance) or RGB values, with or without alpha channel. The generated LUT can be used with the `ApplyLUT` module or within 2D/3D viewers (in conjunction with `SoMLLUT`).
- **RampLUT**: generates an RGB and alpha ramp lookup table (LUT) object. The two ramps for RGB and alpha channels can be parametrized independently. The generated LUT can be used with the `ApplyLUT` module or within 2D/3D viewers (in conjunction with `SoMLLUT`).
- **TableLUT**: generates a lookup table (LUT) object from a table of sampling points (as a string), each consisting of an index value and up to four channel values. The generated LUT can be used with the `ApplyLUT` module or within 2D/3D viewers (in conjunction with `SoMLLUT`).
- **LUTCombiner**: generates an output lookup table (LUT) by combining up to six input LUTs. For each of the input LUTs, the parameters *Mode* (Add, Blend, Subtract, etc.) and *Mask* (R, G, B, RGB, etc.) can be set.
- **LUTCompose**: generates an output lookup table (LUT) by composing up to four input LUTs. The composition of LUTs can be interpreted as the chained evaluation of the lookup functions.

6.3. Data Objects

6.3.1. Markers

- **XMarkerListContainer**: stores a list of XMarker objects as an XMarkerList object. The contents can be displayed, edited and saved. An XMarker object consists of a 6D Position, a 3D Vector, a Type and a Name property.
- **SoView2DMarkerEditor**: allows for an interactive placement, editing and showing of markers on a 2D viewer.

- So3DMarkerEditor: displays markers in 3D and provides some possibilities to interactively edit the markers.

6.3.2. Curves

- ProfileCurve: extracts a profile curve from an image along any data dimension, by reading voxel values from its input image at positions along a specified line.
- SoDiagram2D: displays 2D curves, such as time series, gray scale profiles, histograms, etc.

6.3.3. Contours

- CSOManager: allows for editing the setting parameters and default parameters for CSOs and CSOGroups, as well as for the maintaining of the togetherness of CSOs and CSOGroups.
- SoCSO3DRenderer: enables a visualization of the CSOs of a CSOList in 3D as an Open Inventor scene. Needs a valid CSOList for input (for example via CSOManager).
- CSOIsoGenerator: allows for a generation of iso contours for a whole image at a fixed iso value. Needs a CSOList that is to be filled (for example via CSOManager).
- SoView2DCSOExtensibleEditor: allows for editing and drawing CSOs. To be used in combination with CSOManager, a CSO sub-editor and a 2D viewer for output.
- SoCSOSplineEditor: allows for a freehand or point-by-point generation of CSOs. Those CSOs are smoothed by a spline interpolation or approximation.
- SoCSOEllipseEditor: allows for generating an ellipse or circle CSO.

6.3.4. Surface objects

- SoWEMRenderer: renders a WEM as an Open Inventor scene.
- WEMIsoSurface: generates the iso surface of a scalar volume image at a certain threshold.
- WEMSmooth: smoothes a WEM by applying either a surface smooth (Laplacian), or a smoothing of the surface's normals.
- WEMBulgeEditor: interactively bulge a WEM surface in a 2D viewer with SoView2DWEMBulgeEditor or directly in 3D with SoWEMBulgeEditor.

6.4. Miscellaneous

6.4.1. Fields

- SettingsManager: loads/saves field contents from/to a file or a settings string.
- SoCalculator: calculates by evaluating expressions (with access to input/output fields) and writing the result to the output fields.
- StringUtils: offers a collection of general purpose operations on strings, for example comparison, case-conversion, find+replace, etc.
- BoolInt: translates between a Boolean and an integer value.
- BoolString: translates between a Boolean and a string value.
- ComposeVector3: composes a vector from float values x, y, and z.
- DecomposeVector3: decomposes a vector into single float values of x, y, and z.
- ComposeMatrix: composes a matrix from float values or from vectors.
- DecomposeMatrix: decomposes a matrix to float values of components or vectors.
- WorldVoxelConvert: converts between voxel and world coordinates with respect to the image that is connected to the operator's image input field. All coordinate fields are interconnected, changes in one field are immediately reflected in the other fields.
- FieldIterator: iterates through a list of field values and successively assigns these values to a collection of specified fields in a network. Can be used to batch-process a number of images, or to perform an operation for a list of parameter values and store the results in different output image files.

- FieldShift: saves the last ten field changes of an input field. Can be used to collect the most recently selected coordinates, text string changes, enum changes etc.
- FieldListener: displays information about a field and logs field changes.

6.4.2. Diagnostic

- SystemInfo: lists information about the computer, the operating system, and the OpenGL driver and version details.
- Stopwatch: measures the time needed for an operation. Three methods are available: Start-Stop, external duration and image computation.
- SoActionLog: log actions occurring in an OpenInventor scene.

Chapter 7. Creating an Open Inventor Scene

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

- [Section 7.2, “Creating the Applicator”](#)
- [Section 7.3, “Creating the Interaction”](#)
- [Section 7.4, “Creating the Anatomical Image”](#)
- [Section 7.5, “Finishing the Complete Open Inventor Scene”](#)

Here a look at what we want to accomplish: a dynamically definable applicator (needle for minimally invasive surgeries) shall be placed at a position and an angle relative to the rendering of an anatomical image.

Figure 7.1. Example Network: Open Inventor Result

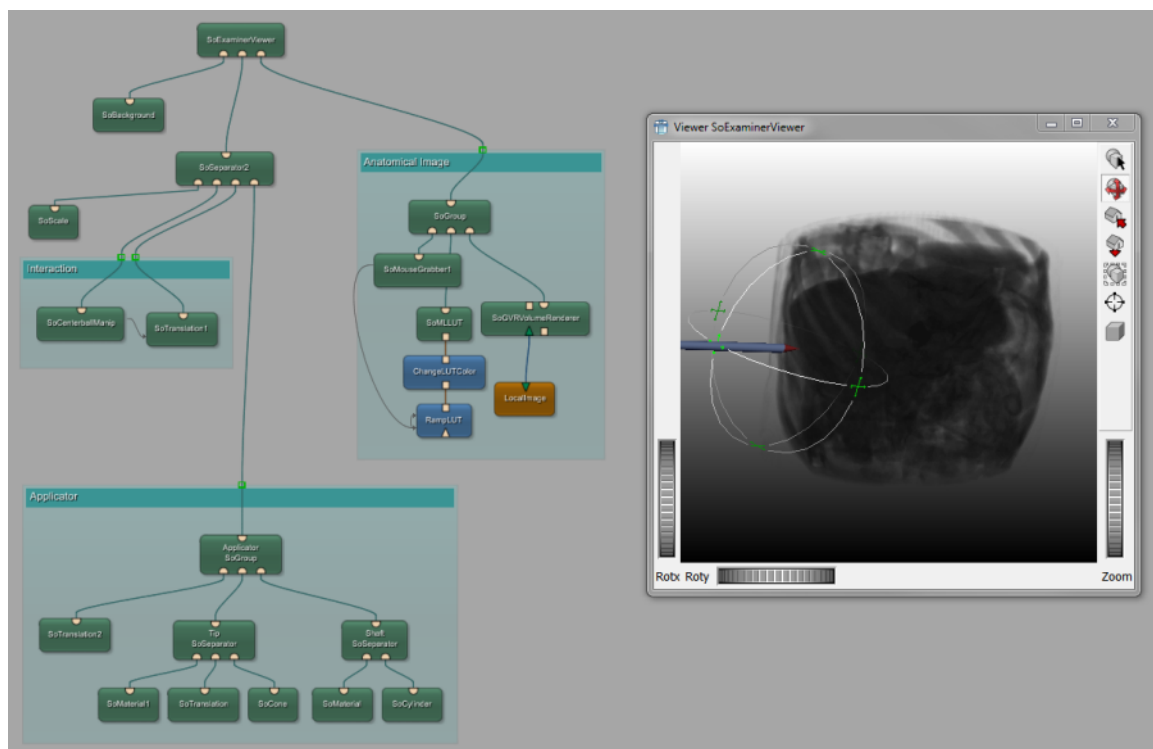
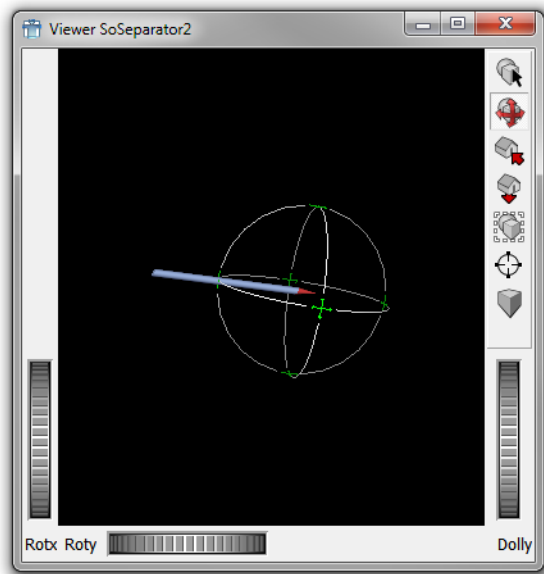


Figure 7.2. Applicator Only

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.

7.1. 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).

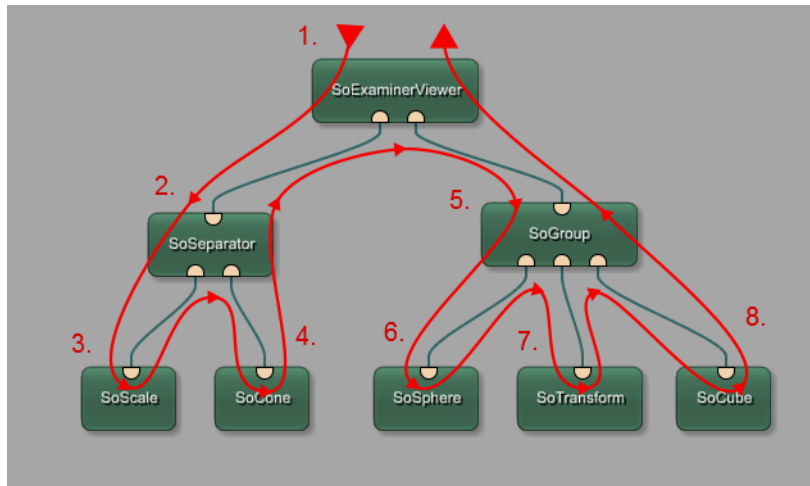
Characteristics of an Open Inventor scene graph:

- 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). For more information about connecting to an Inventor group node, see Section 3.4, “Connecting, Disconnecting, Moving, Copying, and Replacing Connections”.

In [Figure 7.3, “Traversing in Open Inventor”](#), the red arrow shows the order of traversal, from top to bottom and left to right. The numbers designate the order in which each module is passed first, from 1 to 8.

Figure 7.3. Traversing in Open Inventor



Typical functions of Open Inventor modules are:

- Draggers and manipulators
- Group nodes
- Light sources
- Transformations
- Cameras
- 3D viewers
- Geometric objects (Spheres, Cones, 3D Text, Nurbs, Triangle 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.

Another important point is that field changes in Open Inventor modules are handled differently to ML modules:

- Field changes in ML modules are executed synchronously: The field change leads to an immediate execution by calling its `handleNotification(Field*)` method.
- Field changes in Open Inventor modules are executed asynchronously: The field changes is stored in a delayed queue. In general, it is not known when this queue will be processed. Processing can be enforced by calling `MLAB.processInventorQueue()`.

For further information on Open Inventor modules in MeVisLab, please refer to the Open Inventor Reference and the Inventor Module Help. For general 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 (<https://www.mevislab.de/>).

7.2. 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

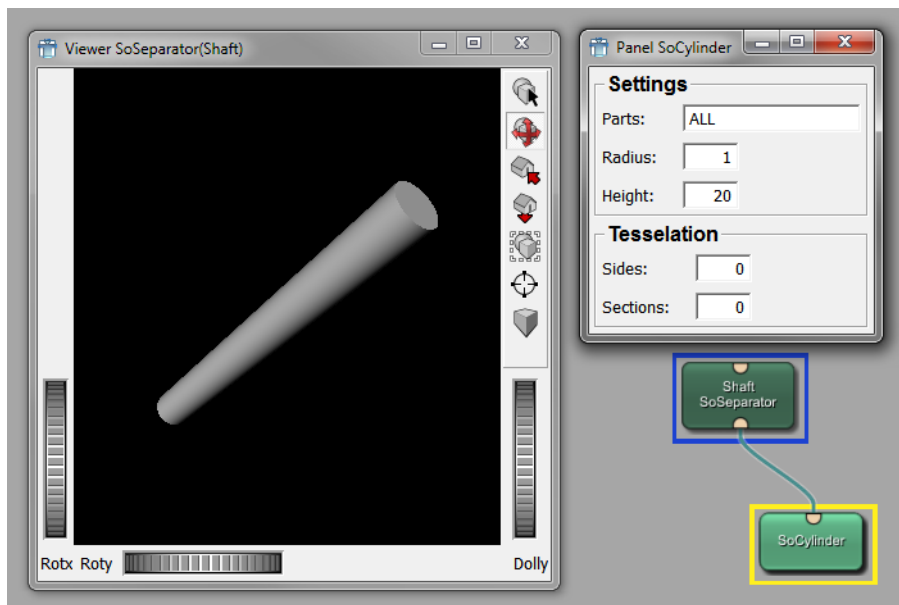
Several Open Inventor modules come with an in-built viewer, like `SoSeparator`, `SoGroup`, `SoRenderArea` and more. For a complete viewer experience, use `SoExaminerViewer` and its associated macro module `SceneInspector`.



Note

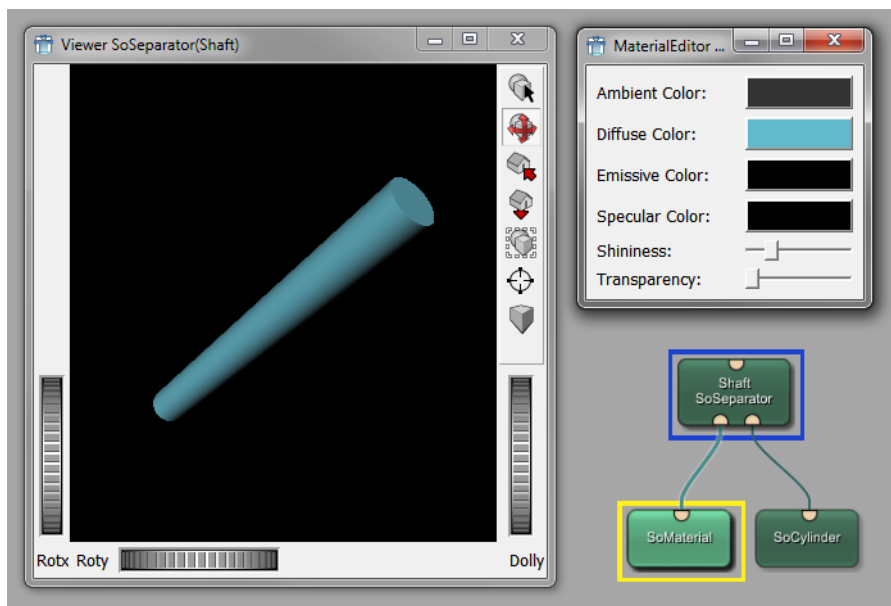
Each of the viewers have their own persistent settings. So if you copy and paste such modules into another network, the zoom settings etc. will be those of the previously used state! If confused, always add fresh modules via the search or the **Modules** menu.

Figure 7.4. Creating the Applicator Shaft



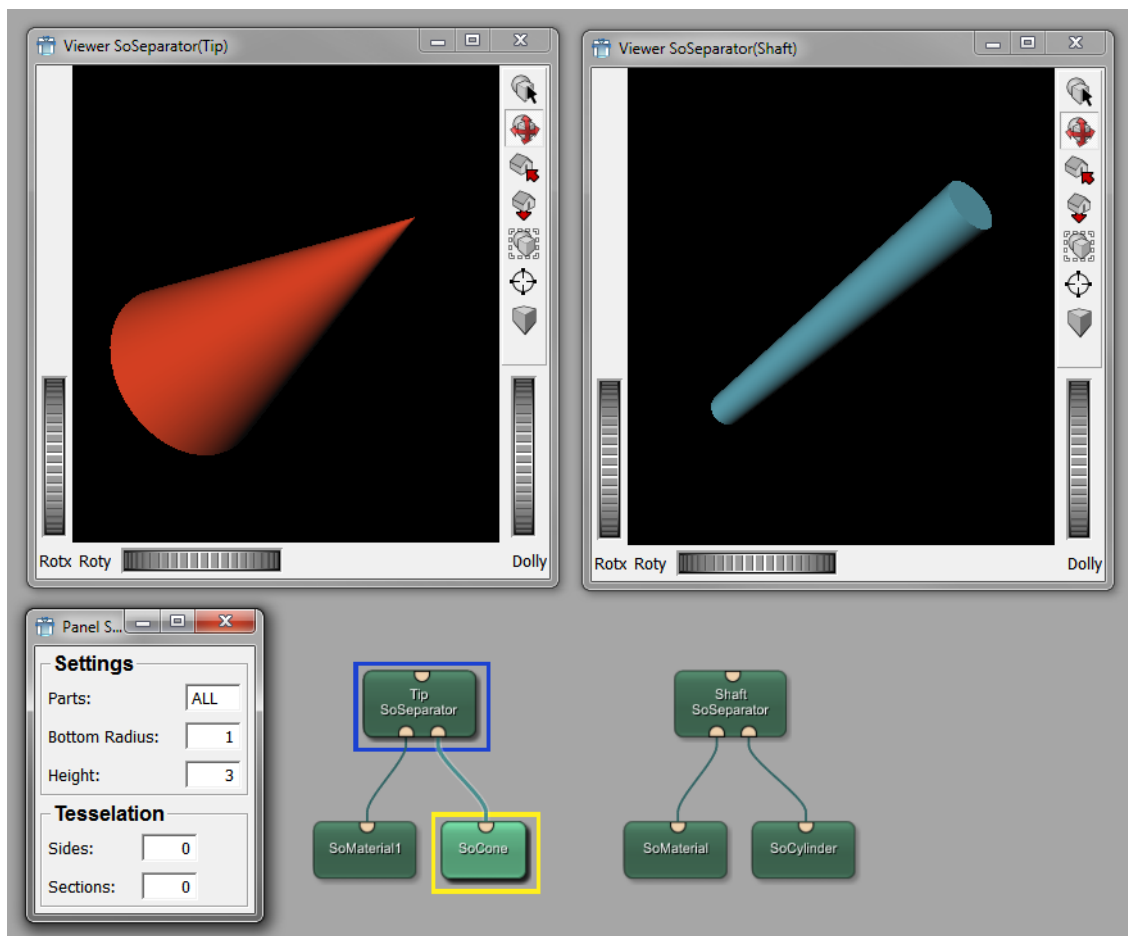
3. Usually, such Open Inventor objects will be colored. Add the `SoMaterial` module before the `SoCylinder` module and edit the material settings. Feel free to play around with the color settings.

Figure 7.5. 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.

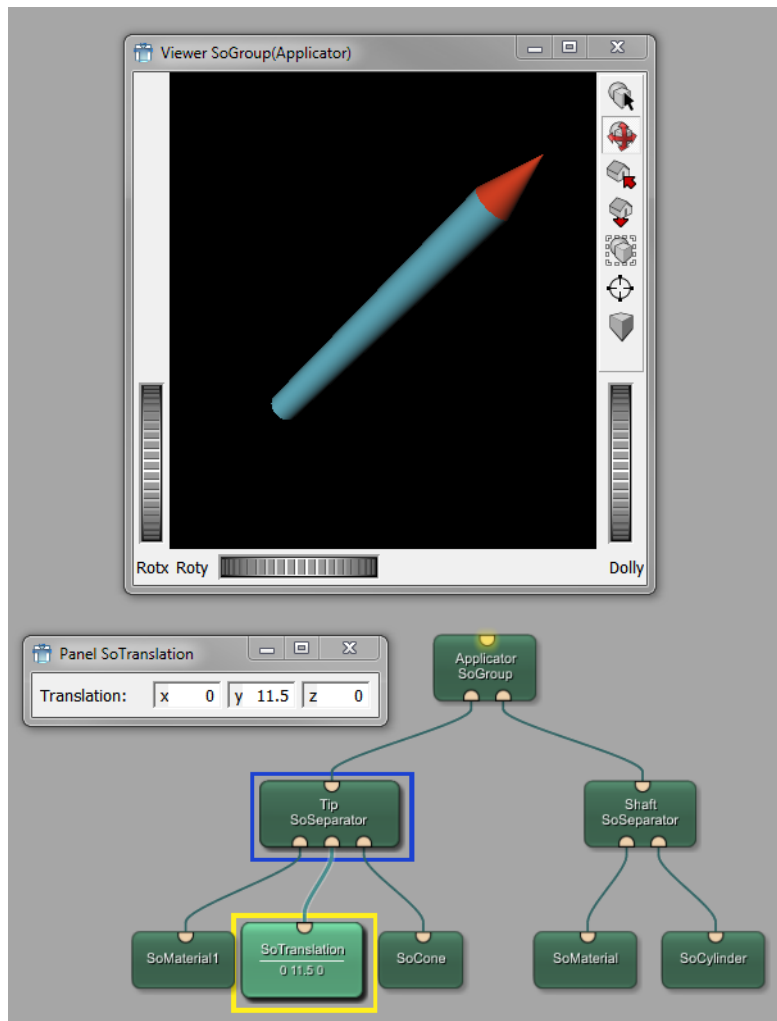
Figure 7.6. Adding an Applicator Tip



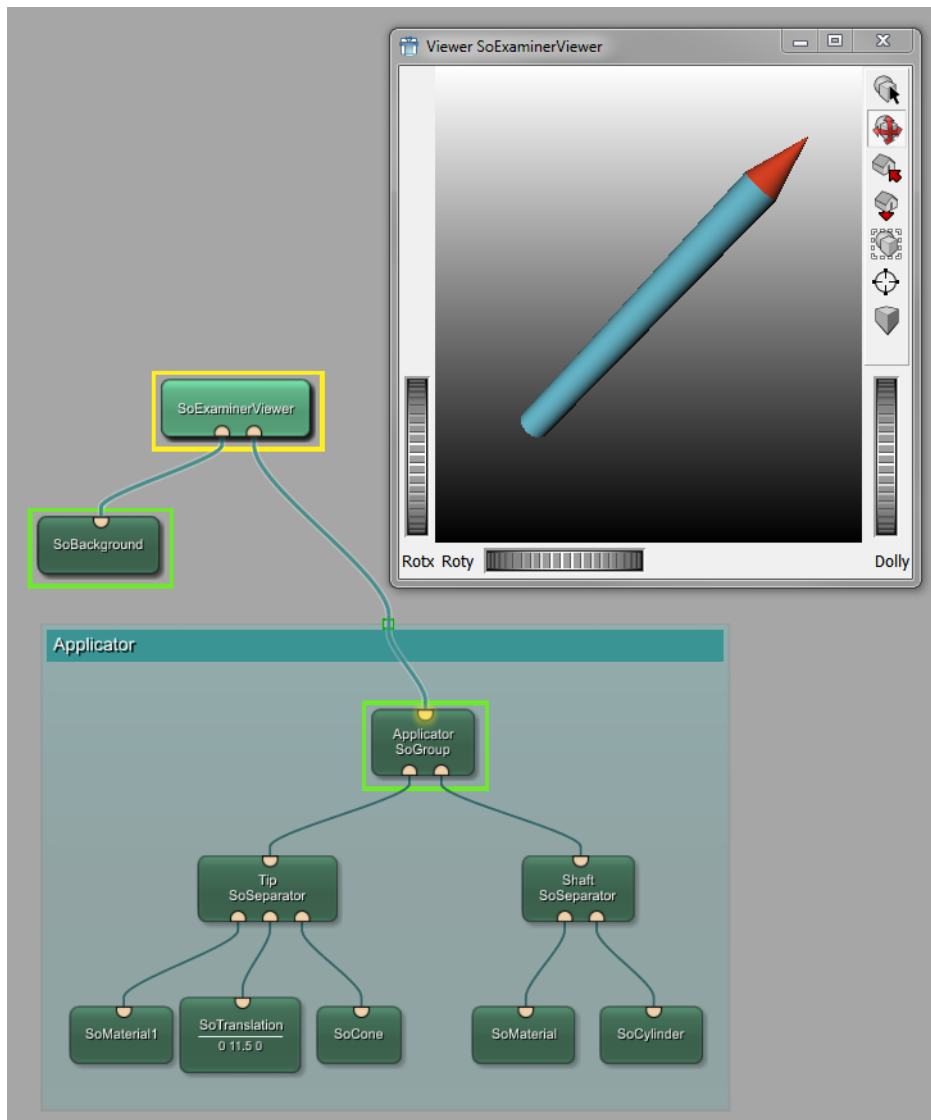
To combine the two independent elements (shaft and tip), we have to a) combine them and b) translate the tip (or shaft) in relation to the other, otherwise the two Open Inventor elements would be placed at the same position, namely the origin of the Inventor's world coordinate system [0,0,0]. (For more information on coordinate systems, see [Chapter 12, Excursion: Image Processing in ML](#).)

5. For the translation, add a `SoTranslation` module in front of to the cone, and set the y-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 7.7. Adding Translation and Grouping



6. For a finishing touch, add a `SoExaminerViewer` for display and a `SoBackground`. The latter adds a gray 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 7.8. Finishing the Applicator

7.3. 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.”



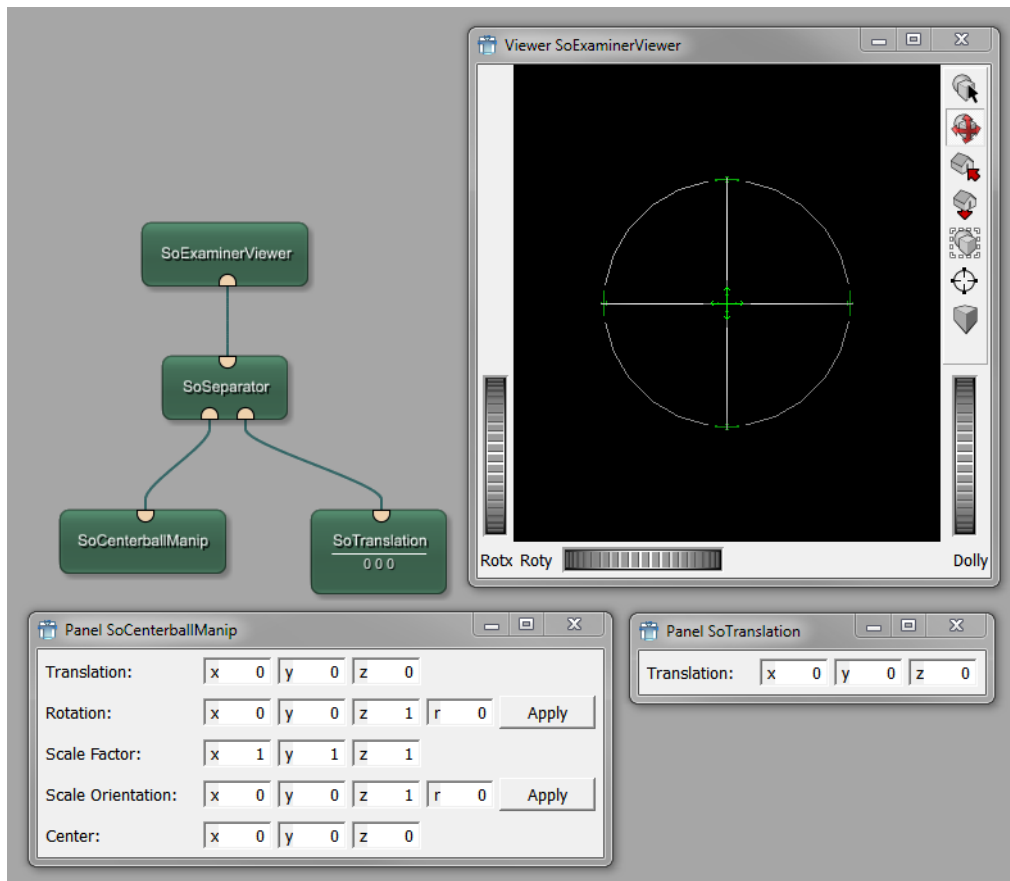
Note

When attaching the `SoCenterballManip` the first time, it might appear very small in the viewer. Just click on it to trigger a rescaling. Once rescaled, the manipulator will keep its size.

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.

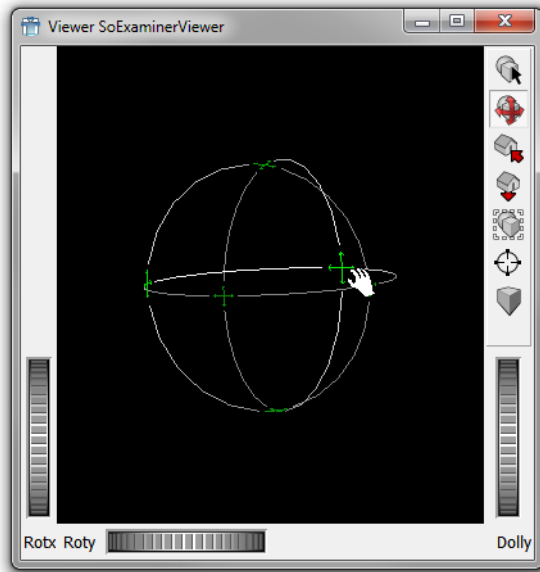
1. To keep the interaction separate from the applicator, add another separator.
2. Then add the modules `SoCenterballManip` and `SoTranslation`. The translation module is necessary to position the centerball (as the latter is foremost intended for rotation and not perfect for translation).

Figure 7.9. Using `SoCenterballManip`



To see the actual ball, use the mouse to rotate the view.

Figure 7.10. SoCenterballManip — Turned



Tip

Press the **ALT** button to toggle between the view mode (for navigation) and the pick mode (for interaction, changes the data on the panel of *SoCenterballManip*).

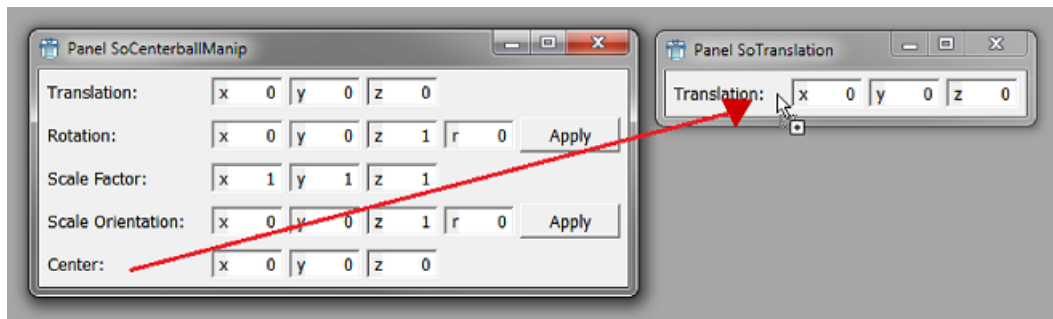
3. To connect the translation of the modules, a parameter connection has to be established between the *Center* field of *SoCenterballManip* and the *Translation* field of *SoTranslation*. This is done by opening the panels, clicking near the *Center* field and dragging it onto the other panel until 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

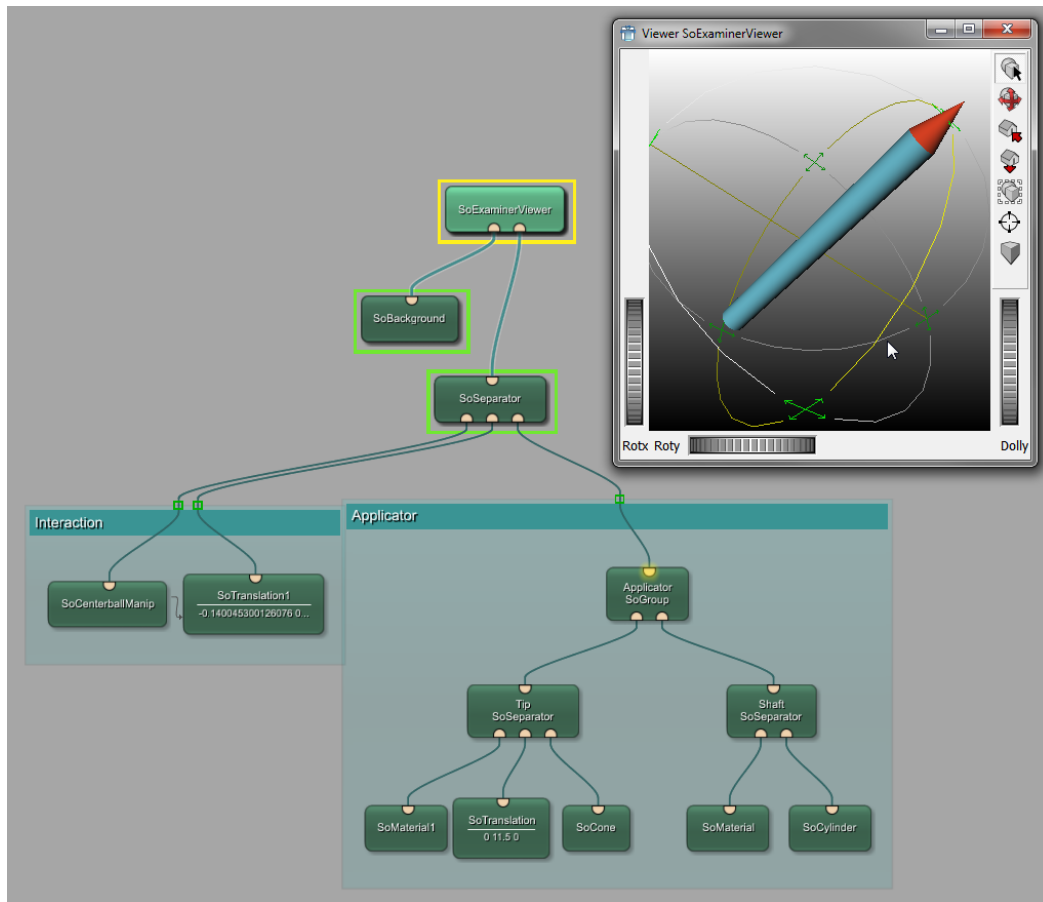
For an overview of all parameter connections, open the **Parameter Connections Inspector** via the menu bar, **View** → **Views** → **Parameter Connection Inspector**.

Figure 7.11. Connecting Parameters



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

Figure 7.12. Combining Interaction and Applicator



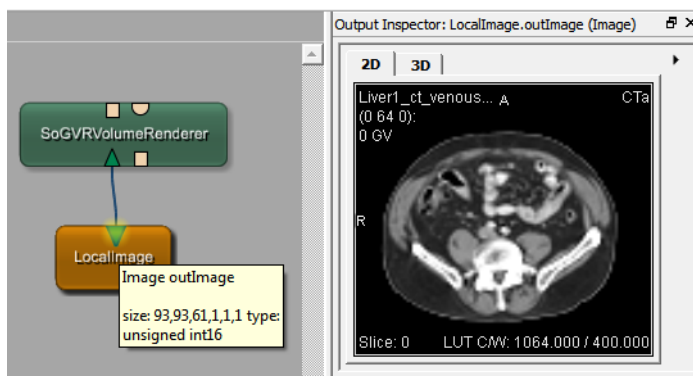
The applicator can now be rotated or dragged into any direction by using the handles on the manipulation sphere.

7.4. Creating the Anatomical Image

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

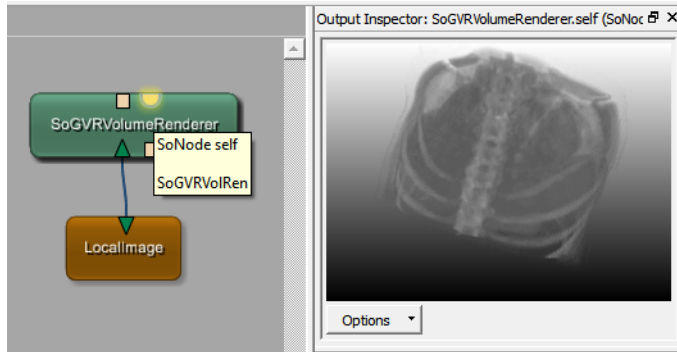
1. As first step, add a `LocalImage` module. Select an image from the demo data folder, for example the liver set at `$(DemoDataPath)/Liver1_CT_venous.small.dcm`. You can view the result in the normal **Output Inspector**.

Figure 7.13. 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.

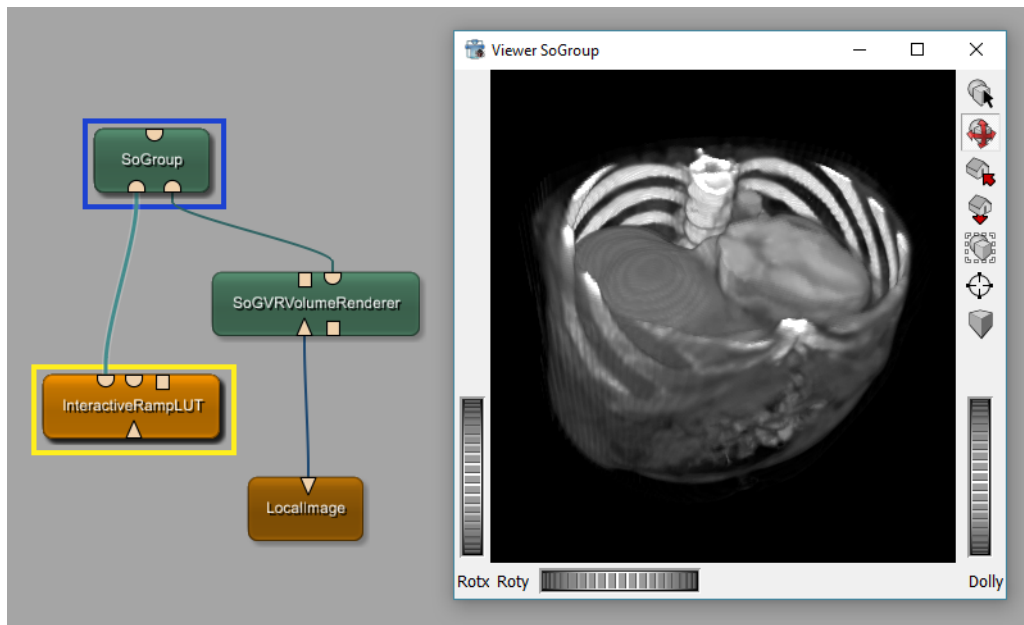
Figure 7.14. Adding the GigaVoxel Renderer



For the windowing we use the `InteractiveRampLUT` module. This module changes the windowing values by tracking the mouse while the right mouse button is pressed.

3. Add the module to your applicator network and connect it to the `SoGroup` module, in front of the rendering module.

Figure 7.15. Adding the Windowing to the Applicator



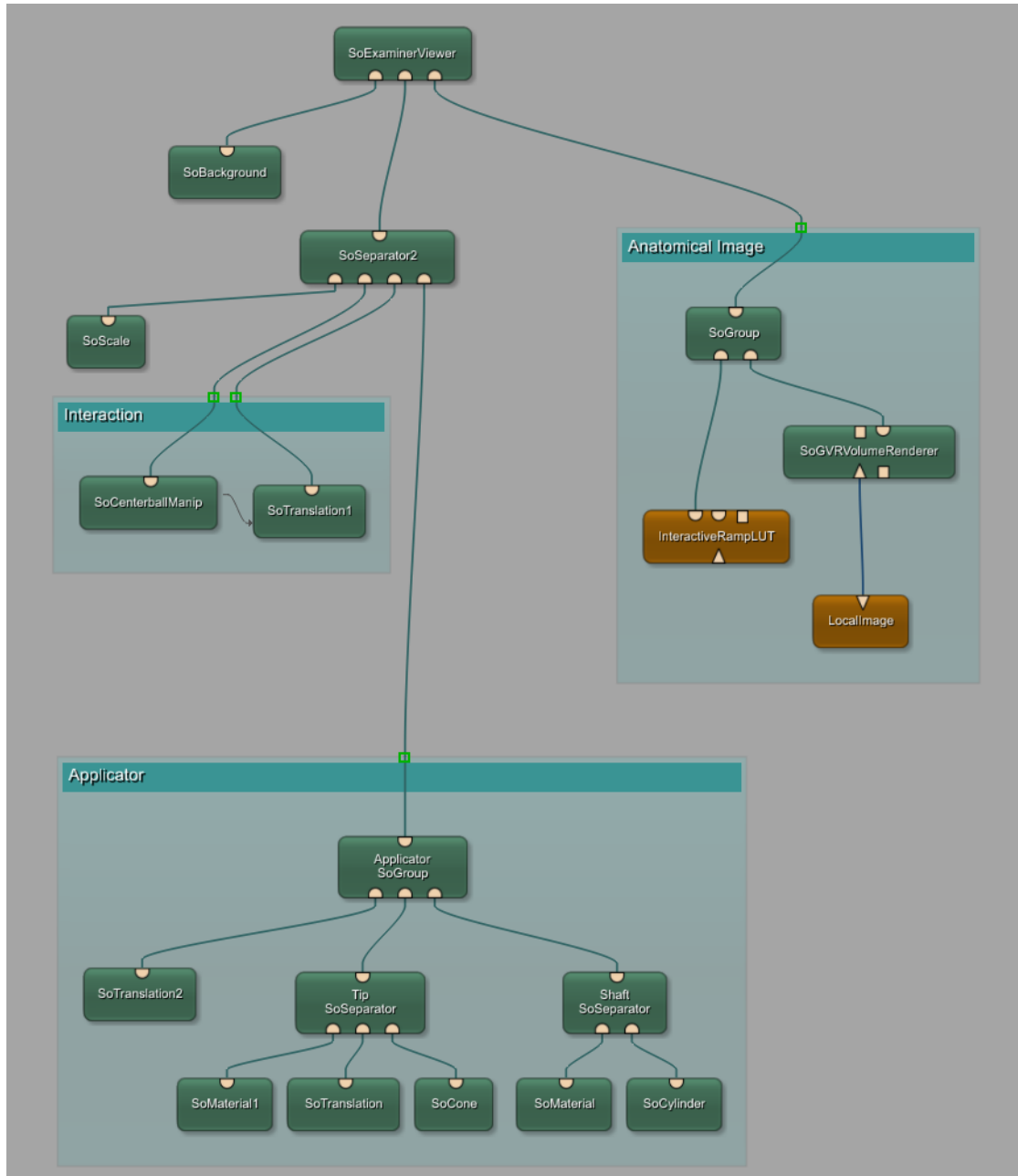
The default settings of the `InteractiveRampLUT` are suitable for our purposes, so we don't need to change anything.

7.5. 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.

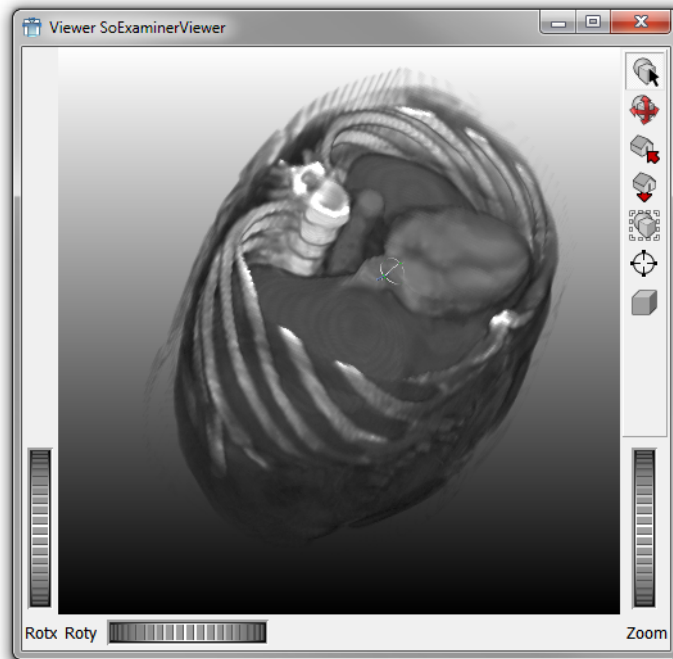
Figure 7.16. Combining the Groups



Note

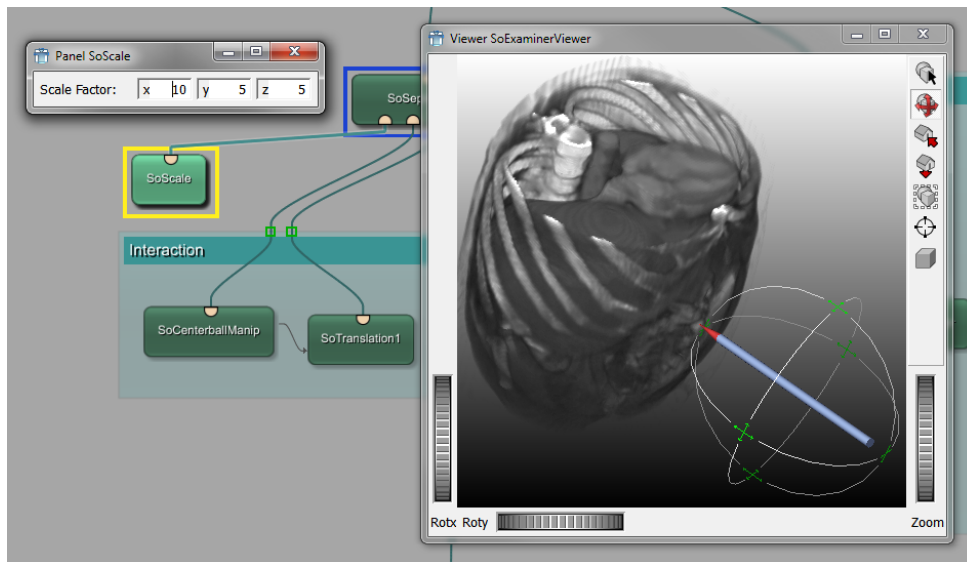
Because the scene with the anatomical image can be rendered with transparencies, add it right-most to the viewer so it is rendered last.

Figure 7.17. 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 7.18. Adding the Applicator Scaling

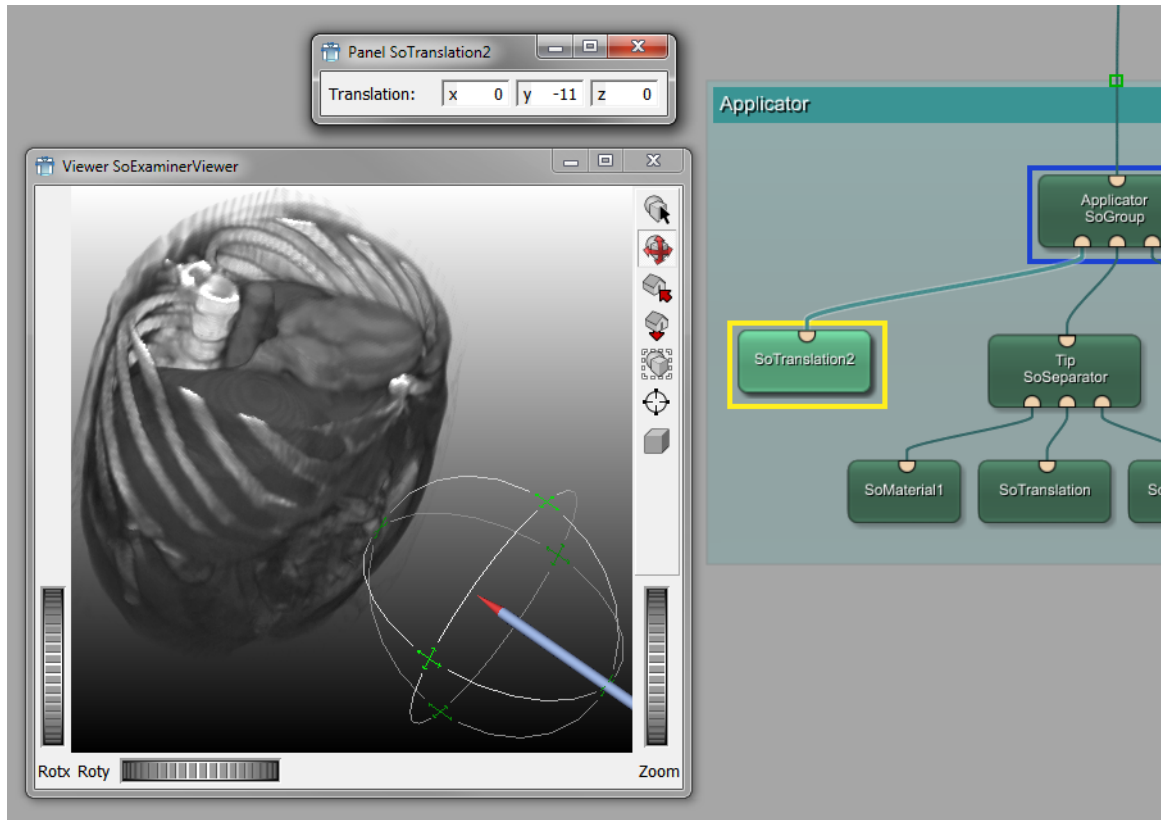


3. Then take the applicator and move it to the body to point at whatever spot you want to point at.

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 7.19. 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** → **more...** → **ApplicatorExample.mlab**).



Tip

In the chapter [Chapter 10, Developing a Macro Module for an Applicator](#), the applicator modules will be used as the starting point for programming a Python macro.

Chapter 8. Starting Development with Package Creation

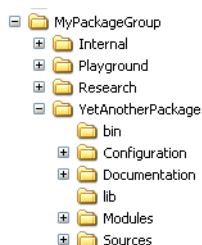
8.1. What are Packages

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 8.1. 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

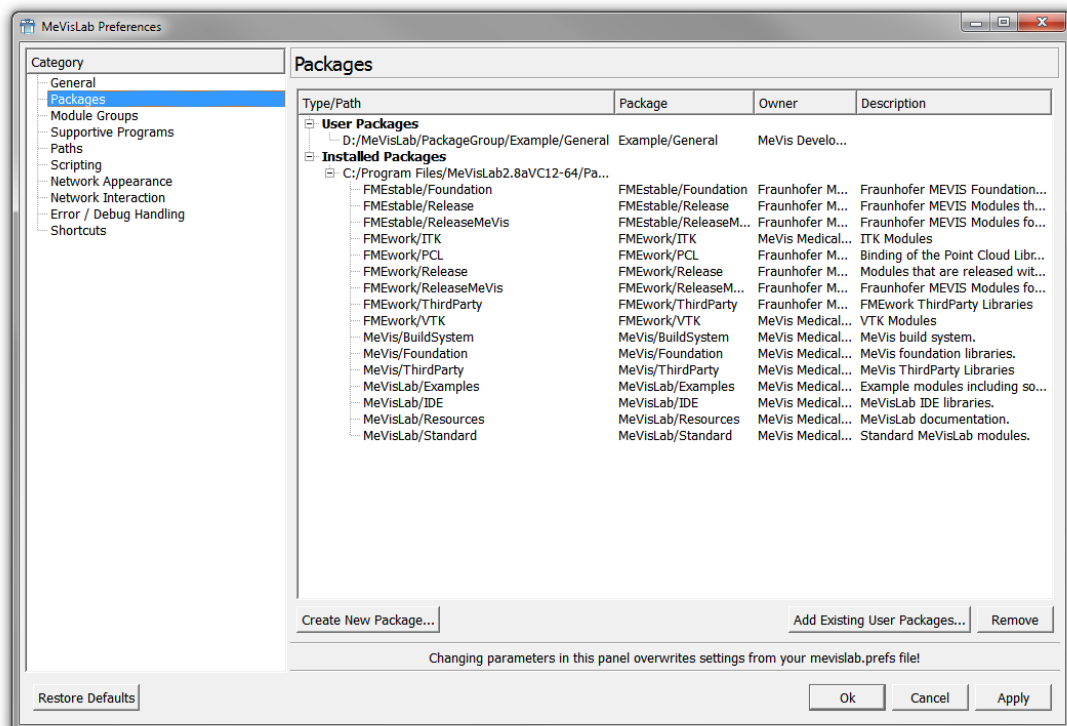
- the UserPackagePath (as set in the MeVisLab Preferences dialog)

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 meta-tool ToolRunner. See the ToolRunner documentation for details.
- by checking the MeVisLab Preferences, section “Packages”.

Figure 8.2. 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 grayed out and labeled “(Overwritten)”.

You can:

Create New Package: Opens the Package Wizard (see [Section 8.2, “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.

8.2. 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 8.3. Package Wizard

Package Wizard

General settings for your package

Package Information

Package Group: *

Package Name: *

Package Owner:

Package Description:

Target Directory

Target Directory: *

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, enter “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 “Example/General” 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 9. Introduction to Macro Modules

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

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 Package 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` 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 10, 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.

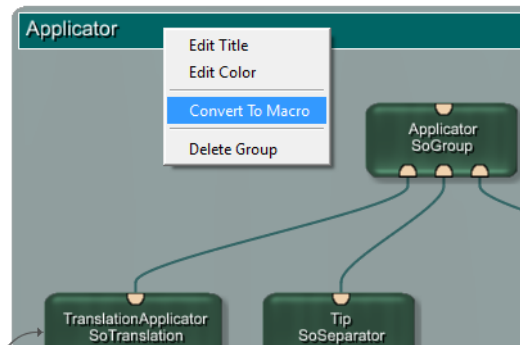
Why this encapsulation?

- 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**

You can also convert a group to a (local) macro via the group's context menu.



Chapter 10. 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 10.1, “Creating a Basic Global Macro”](#)
- [Section 10.2, “Adding the Macro Parameters and Panel”](#)
- [Section 10.3, “Programming the Python Script”](#)
- [Section 10.4, “Addition: Shifting the Whole Tip”](#)



Note

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

10.1. Creating a Basic Global Macro

Our first global macro needs an internal network. We will use the *Applicator* module group as this network.

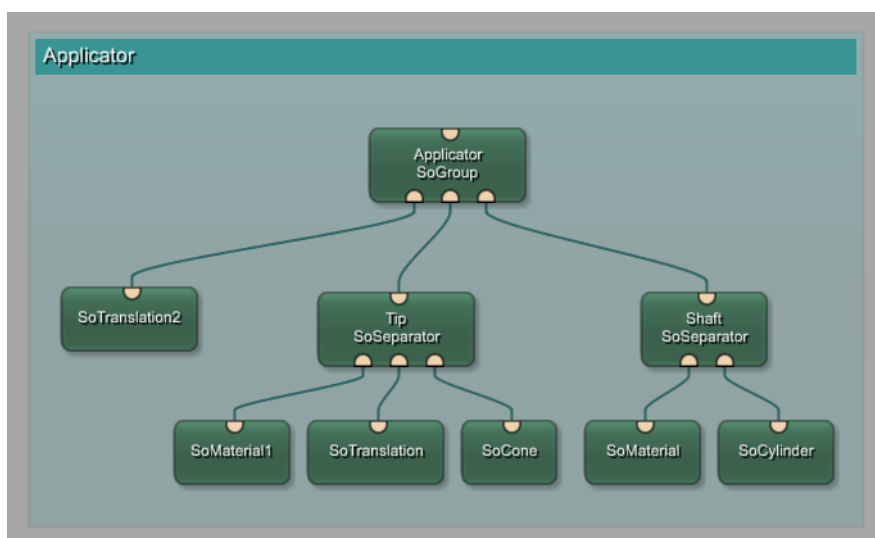
1. For a start, open a new network tab (**File** → **New** or a keyboard shortcut) and copy and paste the applicator modules (**Edit** → **Copy**, **Edit** → **Paste** or the respective keyboard shortcuts) to the new network.



Tip

You can select the Applicator group with a double-click on its title bar and just copy the group.

Figure 10.1. Starting a new Macro from the Existing Applicator

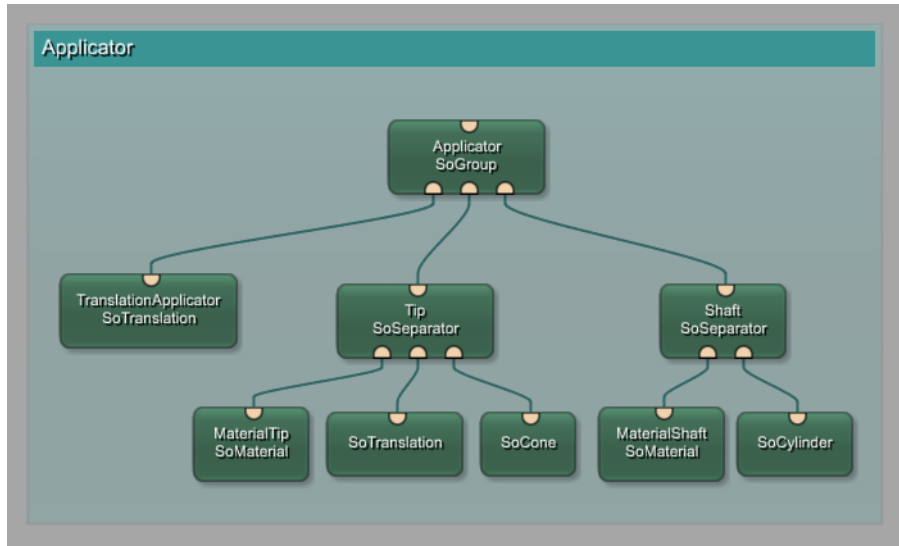


2. Clean the automatic instance names of the modules — as they will be used for a new macro, there is no need to have names like “SoTranslation2”. Remove all numbers and write all module instance

names starting with capital letters (if you want to) by right-clicking the module and selecting **Edit Instance Name** from the context menu.

In our example, this is the resulting network:

Figure 10.2. Existing Applicator with Clean Instance Names



When the module names are cleaned up, save the network at some convenient location. On creating the global macro, this network will have to be referenced and is copied to its final destination on finishing the creation of the global macro.

3. Open the **File** → **Project Wizard** and choose **Macro Module**.
4. Enter the properties for your new module.

Figure 10.3. Macro Module Wizard

Modules (Scripting)/Macro Module

Module Properties

Enter the general properties of the module.

General Module Properties

Name: * Author: *

Comment:

Keywords:

See Also:

Genre: ☒ Add reference to example network

Select Target Package

Package: *

Project Properties

Directory Structure:

Project: * Prefix:

☒ Include project files

* : Required fields

- **Name:**

The name as entered above is displayed, for example `ApplicatorMacro`. You can edit the name here. The module name has to be unique within the MeVisLab module database (including the SDK module database). Therefore, you may need to change the module name slightly in case of a collision.

- **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**

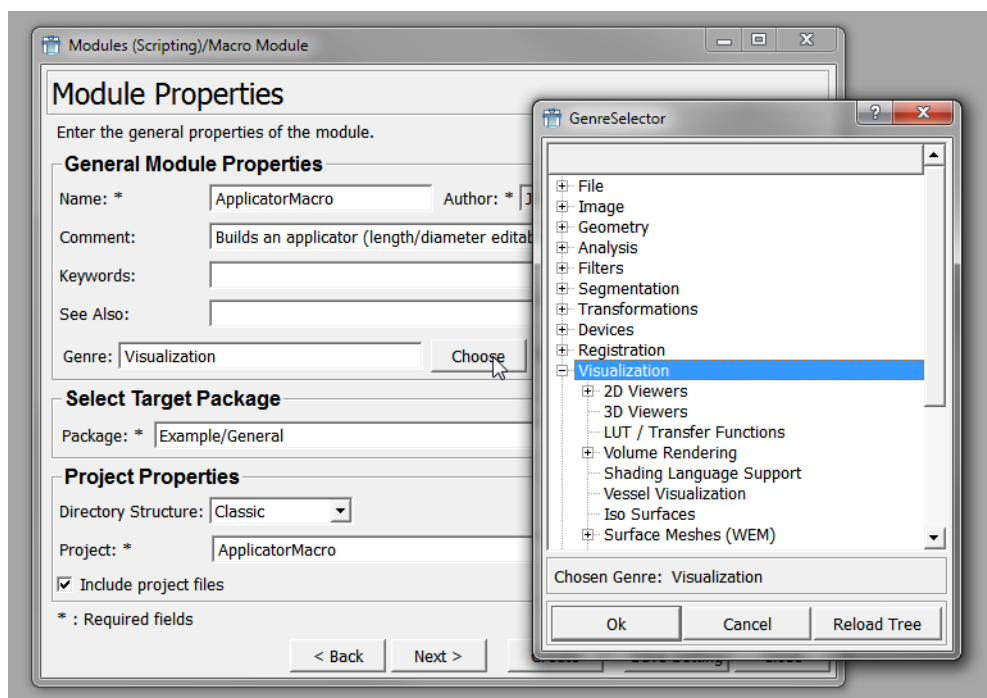
Enter the genre. Genre entries are mandatory; they define the place of the module in the **Modules** menu and the **Module Browser**. For suggestions, check out similar modules in the database.



Tip

The macro module wizard offers to choose from a tree of available genres:

Figure 10.4. Selecting a Genre



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 “Example/General” as created in [Section 8.2, “Creating a User Package for Your Project”](#).



Note

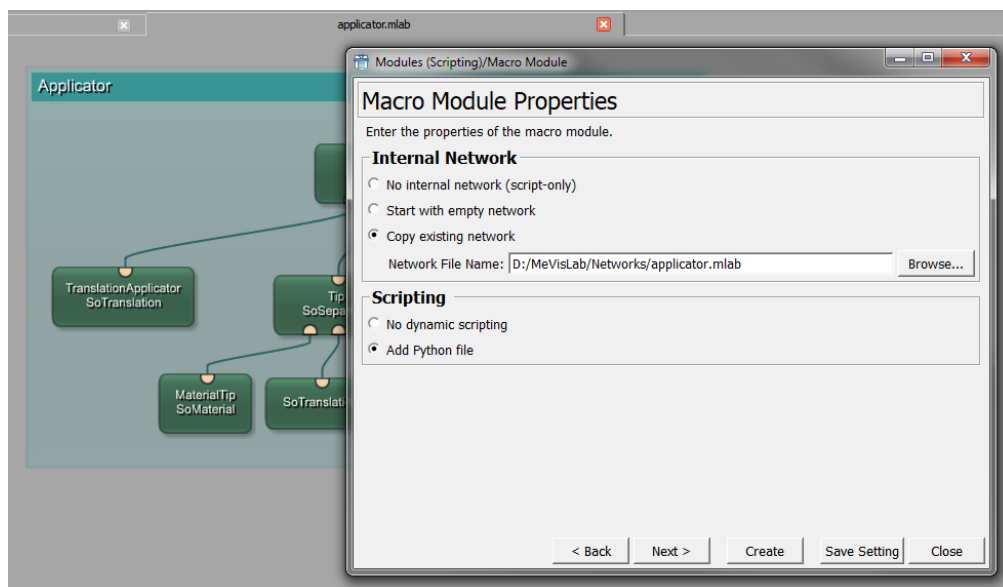
Only existing Target Packages can be selected; if you want to use a new one, you have to create it before creating the module.

Click **Next**.

5. On this tab, browse to the previously saved network and set it as the **Network File Name**.

You might leave the option to add Python scripting unchecked as we will add the scripting file later on manually in this tutorial.

Figure 10.5. Macro Module Properties



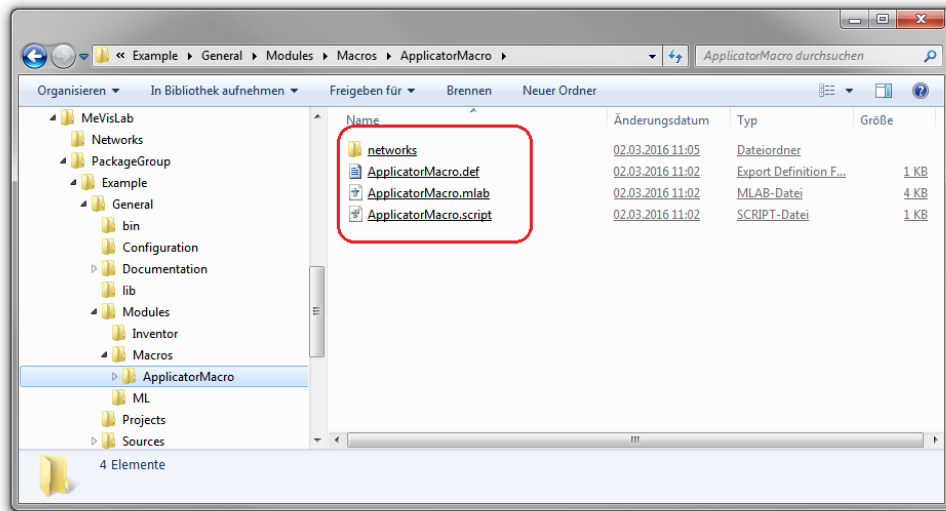
Click **Create**.

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

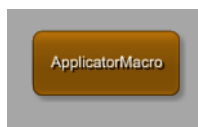
- `.def`: module definition file, for registering the module(s) to the MeVisLab module database.
- `.mlab`: network file which includes the modules and their settings.
- `.script`: MDL script file for the panel and from which Python code may get called.

Figure 10.6. File Browser with the New Macro Module Files



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

Figure 10.7. ApplicatorMacro as Macro Module



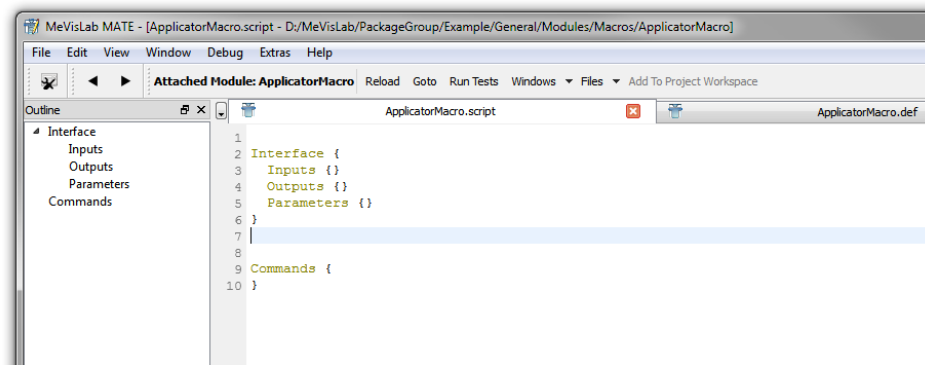
6. To display the internal network on a second tab, right-click the module and select **Show Internal Network** from the context menu. Alternatively, you can hold **Shift** and double-click the macro module.

10.2. 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 10.8. ApplicatorMacro.script in MATE



Tip

MATE comes with some special features like autocompletion, syntax highlighting, indentation, etc. for MDL, Python and help files. For an extensive list, see the MeVisLab Reference Manual, chapter "MATE".

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

- 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.
- Commands:** defines the scripting file to be executed upon the activity of defined fields.
- 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.

- 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 `Applicator`. 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.

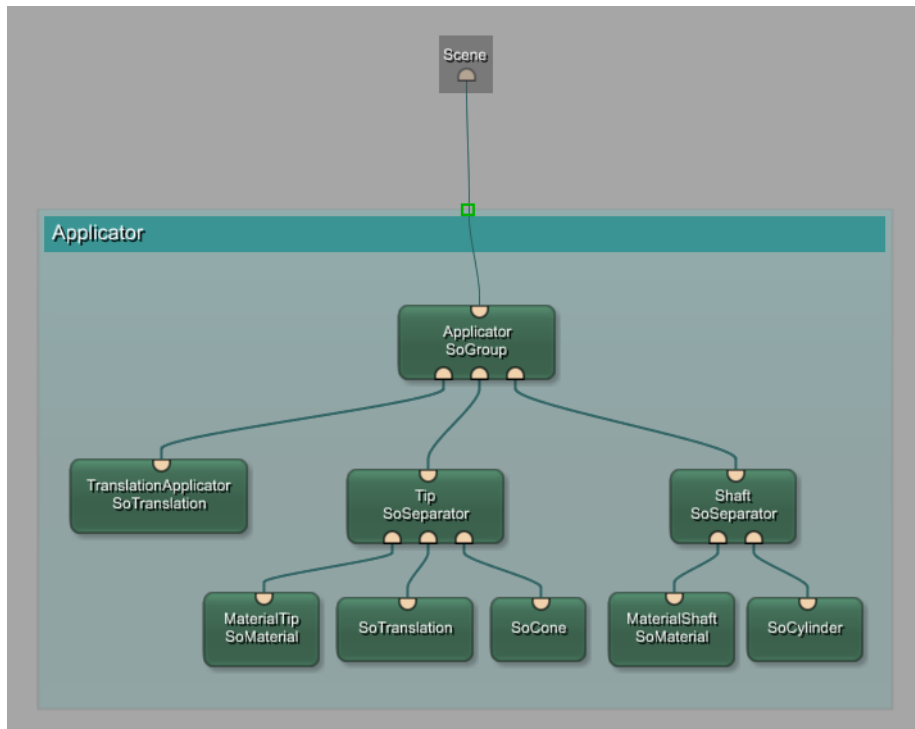
- Then reload the module by right-clicking the macro module and selecting **Reload Definition** to apply the changes. The `ApplicatorMacro` module now shows an Open Inventor output connector.

Figure 10.9. 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 10.10. 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
    }
  }
}
```

```

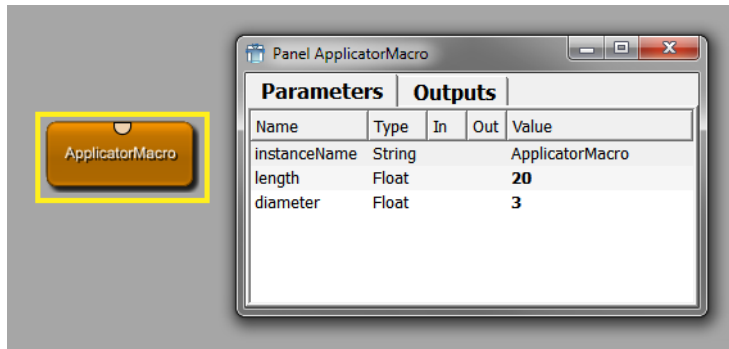
    }
    Field diameter {
      type = float
      value = 3
      min = 0.1
      max = 10
    }
  }
}

```

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 10.11. 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 {

}

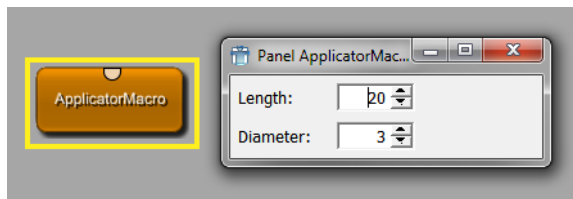
Window {
  Category {
    Field length { step = 1 }
    Field diameter { step = 0.1 }
  }
}

```

Save the script and reload the macro module.

- 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 10.12. 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 (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 in MeVisLab (search for modules starting with "Test...").

10.3. 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. **Note**



In your code, you may need to import some of the global classes like `MLAB`, `MLABFileDialog` or `MLABFileManager` from the "mevis" module (e.g. from `mevis import MLAB`) to get access to some convenience functions. See the scripting reference for a list of all available helper functions.

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

```
def AdjustLength():  
    pass  
  
def AdjustDiameter():  
    pass
```



Note

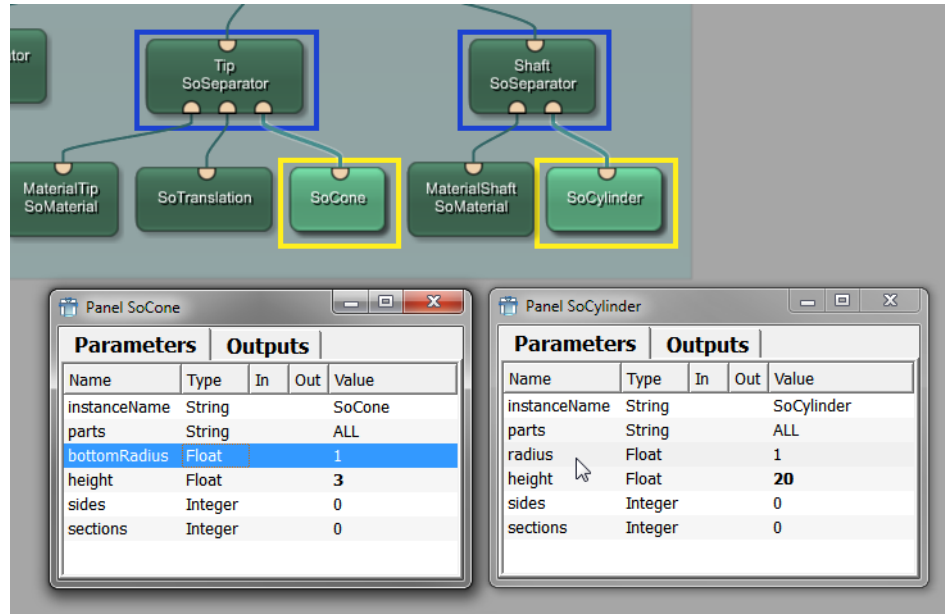
In Python, block structure is defined by indentation. Therefore, it is important to indent the lines as shown in the code examples. In the MATE editor, this will happen automatically.

3. 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
```

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 10.13. 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
```

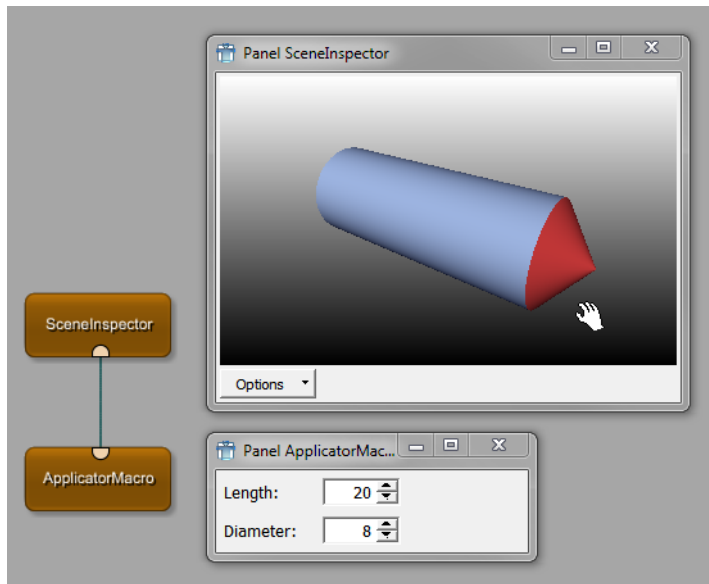
As the radius is half the diameter, a correcting factor of 0.5 has to be added to the diameter equation.

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

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

4. 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 10.14. Changing the Diameter of the Applicator



5. 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
```

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:

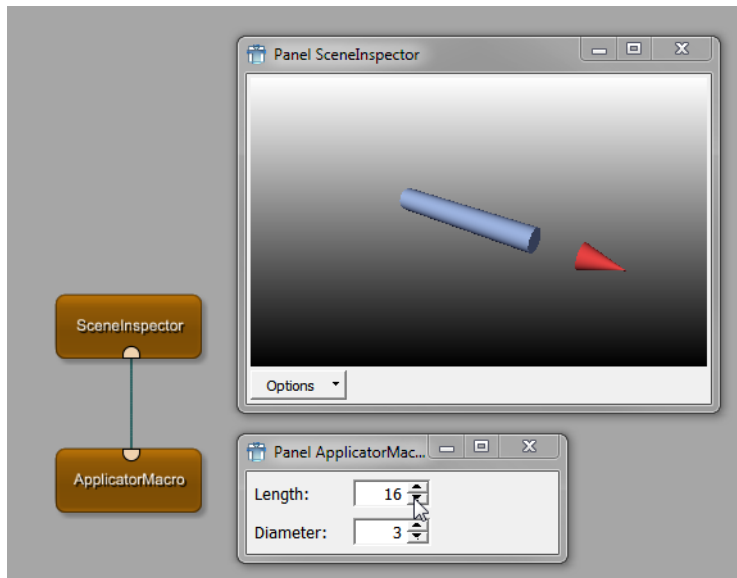
```
def AdjustLength():
    overallLength = ctx.field("length").value
    tipLength = ctx.field("SoCone.height").value

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

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

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 10.15. Strange Behavior of the ApplicatorMacro



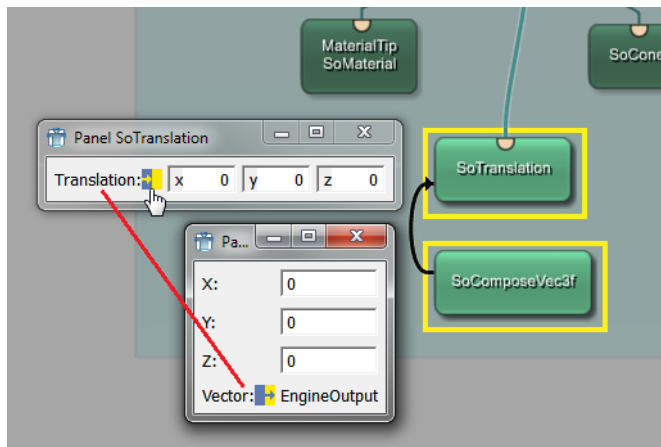
This is due to not having connected the calculated `tipTranslation` with the `TranslationTip` module yet.

- 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`. Since `SoComposeVec3f` supports an arbitrary number of elements on x,y,z, we have to use `setListValue`.

```
ctx.field("SoComposeVec3f.y").setListValue([tipTranslation])
```

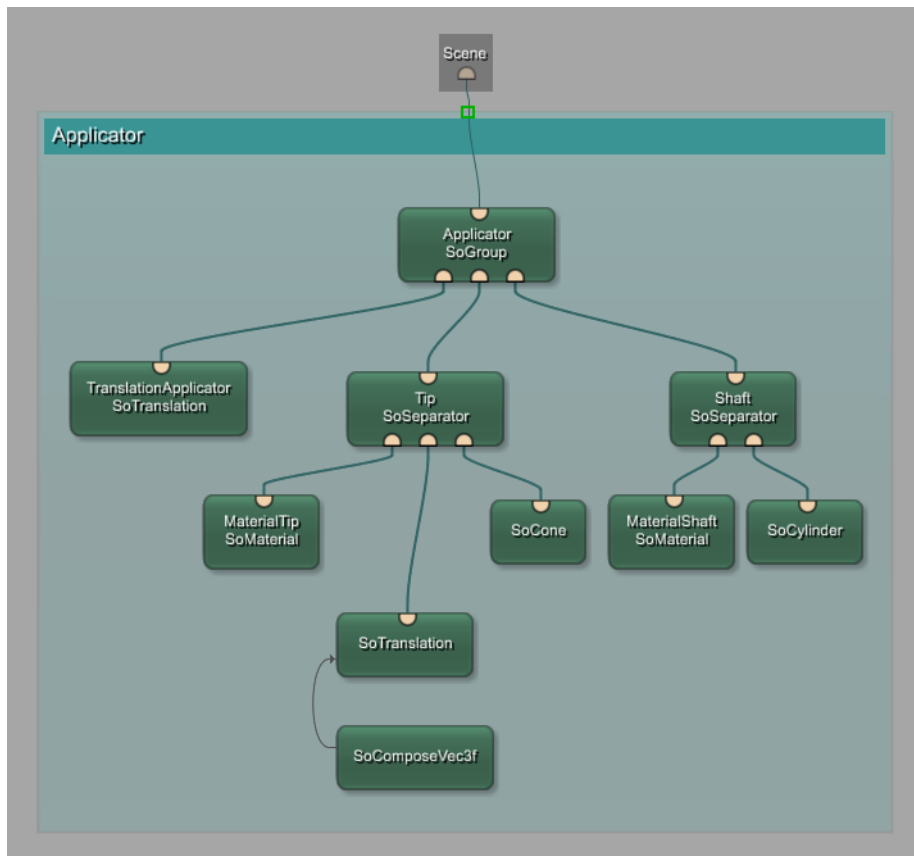
- In a last step, this translation needs to be connected to the tip's `SoTranslation` module via a parameter connection in the network.

Figure 10.16. Adding the Correct Tip Translation



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

Figure 10.17. Complete ApplicatorMacro



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

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

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").setListValue([tipTranslation])
```

10.4. 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 in 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.

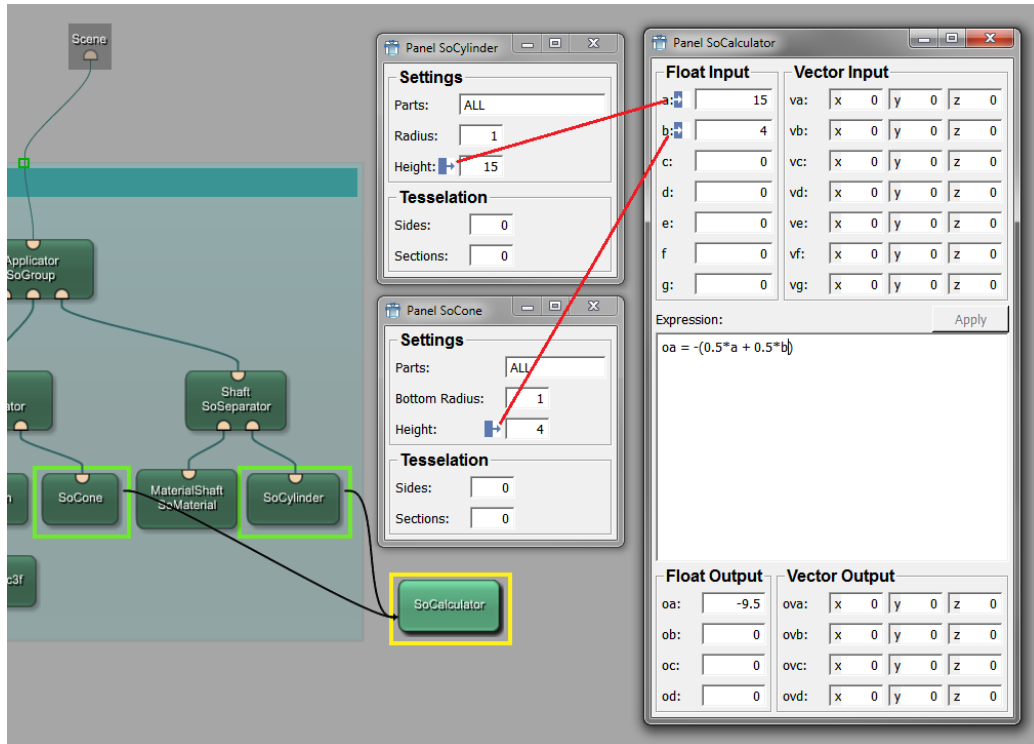
For this, the following modules need to be added:

- `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 10.18. 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`

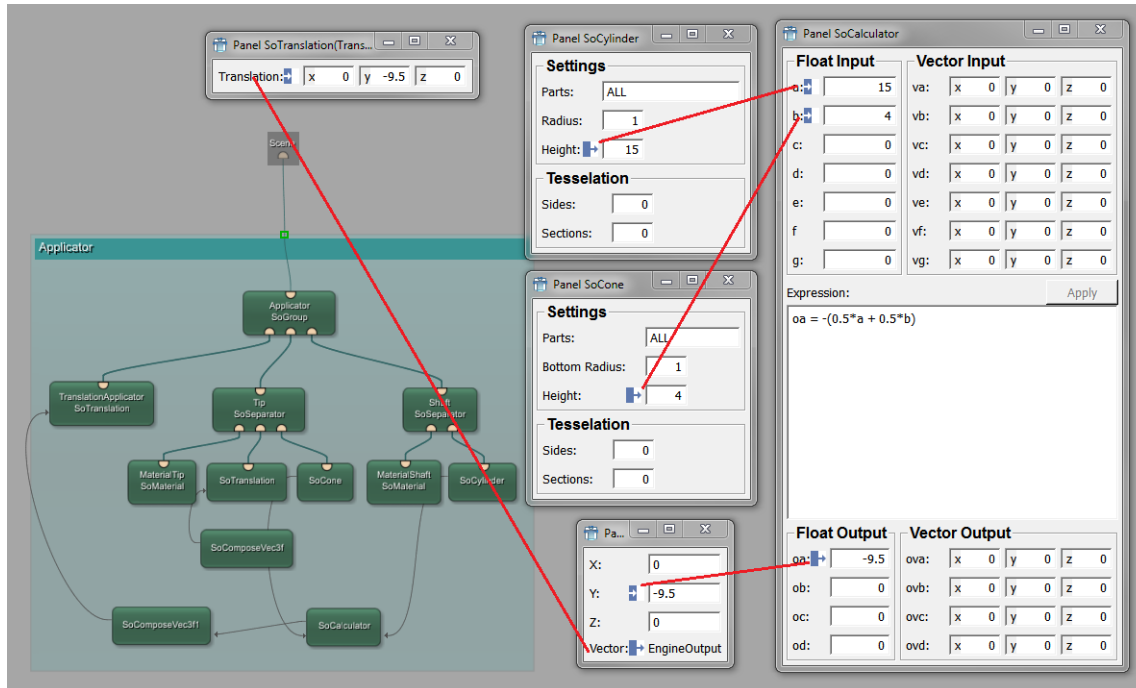


Tip

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 10.19. 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 main advantage of using script is that the setting of parameter field values or the triggering of a (re-)computation is much more controlled. Using parameter field connections can easily lead to unwanted notification avalanches.

- 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.



Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Examples/Modules/GettingStarted/ApplicatorMacroExample`, source files in `$(InstallDir)Packages/MeVisLab/Examples/Sources/GettingStarted/ApplicatorMacroExample`). The module can be added via quick search.

Chapter 11. GUI Design in MeVisLab

The following chapter introduces the concept and the formatting possibilities of GUI design in MeVisLab.

The panel design is given in detail, from a look at MDL basics and principles down to the implementation and scripting of controls.

For the visual appearance of panels, two major options are given: MDL styles and Qt style sheets. Both of them are discussed and illustrated.

1. Panel design with MDL, see [Section 11.1, “MeVisLab Definition Language \(MDL\)”](#).
2. Panel formatting with styles and prototypes in MDL, see [Section 11.3, “MDL Styles”](#).
3. Adding new MDL controls, see [Section 11.3.3, “Creating Custom MDL Controls”](#).
4. Panel formatting with Qt styles (CSS), see [Section 11.4, “Customize GUI Appearance Using Qt Style Sheets \(CSS\)”](#).

The base of every GUI in MeVislab is the panel designed in the MeVisLab Definition Language (MDL). The MDL is described in detail in the MDL Reference.

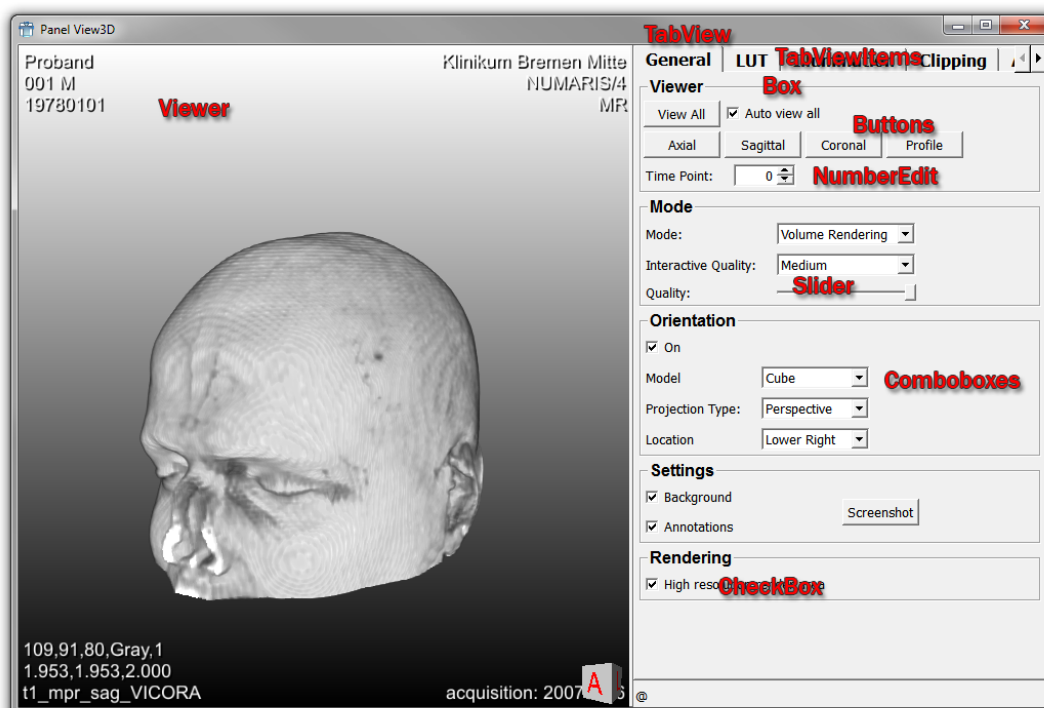


Tip

Examples for GUI designs with the MDL are available in MeVisLab, just enter “Test” into the quick module search (to be able to find the test modules the “Test” module group must be enabled in **Preferences** → **Module Groups**).

The part of the MDL in relation to the GUI is to define the structure of a panel, the included elements and buttons, and the way those elements are arranged. Available elements are, for example, lists, sliders, thumb wheels, text fields, check boxes, buttons, and many more. For arranging the elements, group controls are available, like tables, grids, boxes, and tabs.

Figure 11.1. View3D Panels as Example for GUI Elements



11.1. MeVisLab Definition Language (MDL)

The MDL is more than just a GUI definition language.

- It is a configuration and layout language.
- It is implemented based on the architecture pattern Model-View-Control (MVC) (see the [Wikipedia entry about MVC](#) for the general concept).
- It is a declarative language with a focus on the logic, not the processing (see the [Wikipedia entry about declarative programming](#) for the general concept). It focuses on the hierarchical structure of the content, and offers a MLABTree node interface that can be addressed from scripting and used for error reporting.
- It offers a simple preprocessor (`#ifdef/#include`).
- It is an application-specific language, tailored to the needs of MeVisLab. It adds a strong decoupling of GUI and C++ modules and provides the basis for extensibility to MeVisLab.
- It is used for GUI layout, calling script methods, installer scripts, `.prefs` files, and more.
- The GUI part was inspired by HTML/JavaScript, which is mirrored by MDL/Python in MeVisLab. Both combine a declarative language with an imperative language that adds the actual control flow.

In the following sections, a few interesting and important facts about the MDL are listed that will help in using its full potential.



Tip

The integrated text editor MATE supports MDL syntax and Python with syntax highlighting and auto-completion.

11.1.1. MDL Validator

The validation of MDL files is done with an MDL validator.

- An MDL file can contain any content.
- The validator defines what the MDL tree has to look like.
- The validator takes:
 - An MDL tree to validate.
 - An MDL tree that defines the expected structure (typically `MDLValidator.def`).
 - The MDL style definitions (which can add prototypes, see [Section 11.2.9, "Prototypes for Controls"](#)).
- The validator traverses the tree and prints errors/warnings if the tree does not match the expected structure.
- The tree that defines the expected structure is also written in MDL and is validated by itself.
- The validator file defines what groups are allowed in each group (recursively) and what name/value pairs are allowed.
- It knows the expected value types and can warn for non-existent files, check Integers, Floats, field names, etc.

Excerpt of the validator for the general module definition:

```
Group _Module {
  allowTags {
    comment      = STRING
    author       = AUTHORS
    exampleNetwork = MLABFILE
    ...
  }
  allowChildren {
    Interface = ""
    Commands  = ""
    Description = ""
    ...
  }
}
```

The validators of the specific module types then relate to the general module definition.

```
Group MLModule {
  value = NAME
  allow = _Module      <- allowed groups
  allowTags {          <- allowed tags (make sure to use the right data types)
    class = STRING
    DLL   = DLLNAME
  }
}

Group InventorModule {
  value = NAME
  allow = _Module
  allowTags {
    class          = STRING
    hasGroupInputs = BOOL
    hasViewer      = BOOL
    ...
  }
}
```

For other validators like for fields, panels, etc., the principles work accordingly.

Usually, a developer does not have to deal with the validator aspect of the MDL. However, once a new control is to be implemented, a validator should be written for that to avoid error messages, see [Section 11.3.3, “Creating Custom MDL Controls”](#).

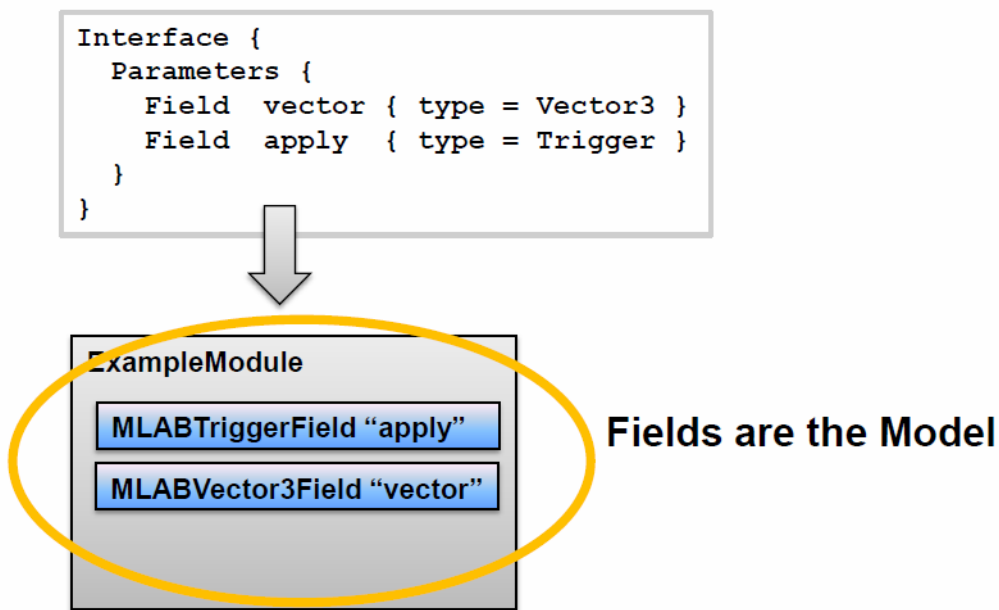
11.1.2. MDL Controls

- MDL controls are derived from `MLABWidgetControl`, a C++ class.
- They create and control their `QWidgets` (the Qt/C++ base class of widgets).
- They provide a Python scripting interface.
- Are reparented to the `QWidget` they create, so that the controllers are automatically destroyed when their `QWidget` is destroyed.
- Custom MDL controls can be created, see [Section 11.3.3, “Creating Custom MDL Controls”](#).

An important effect of the Model-view-controller pattern is the separation of the interfaces (as fields) and the actual controls defined in the windows section of an modules GUI definition.

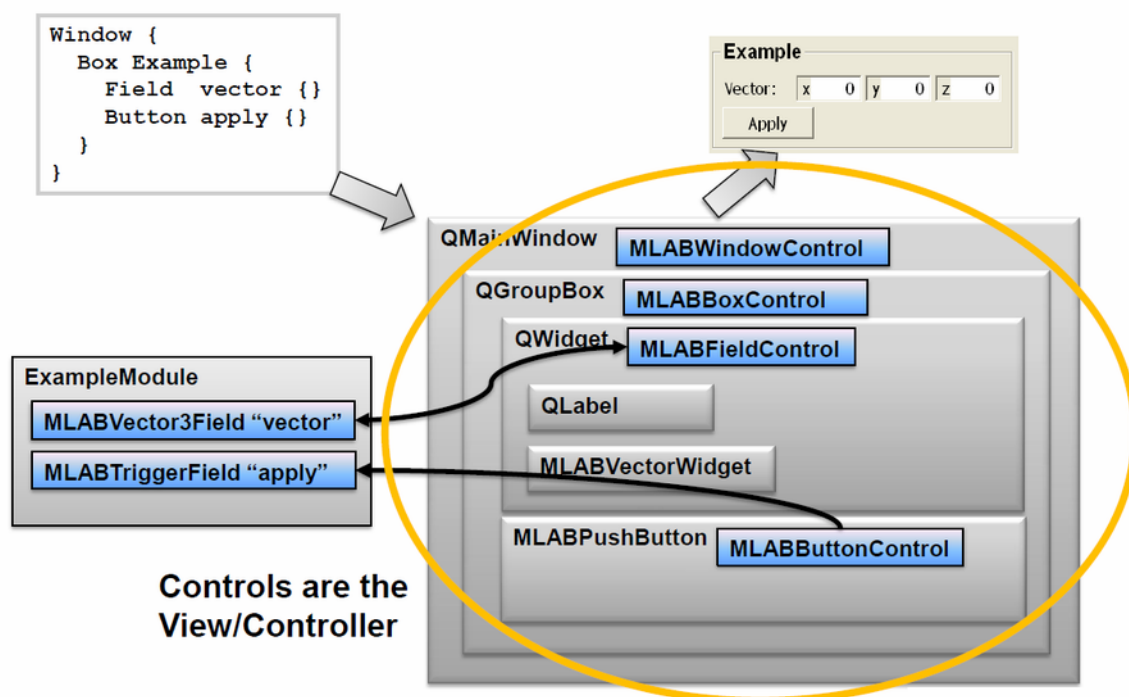
The fields defined in the interface sections are the models.

Figure 11.2. Fields as Model

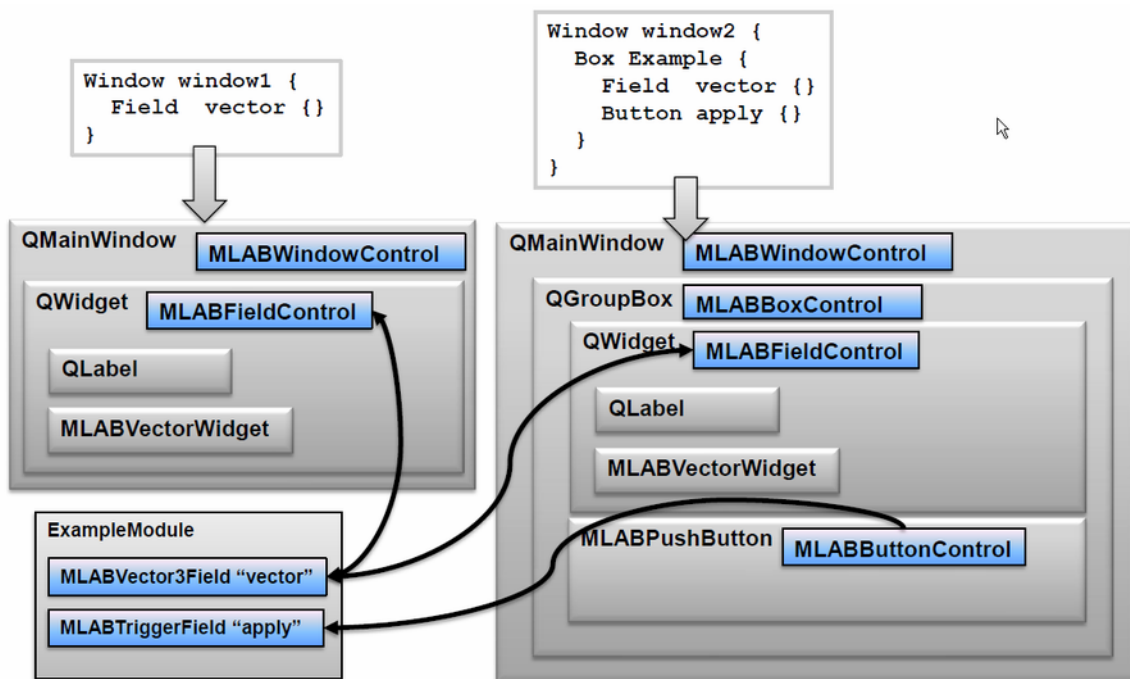


The controls in the GUI are the view and also the controller.

Figure 11.3. Controls as View/Controller



For a set of fields/models, many different views can be built.

Figure 11.4. Controls as Views/Controller

It is important to keep this Model-view-control design in mind, otherwise it is easy to mix up the definition of fields (Interface section) with the GUI control of the field (Window section).

1. Model: defines a new field of type `Vector3` named `vector`:

```

Interface {
  Parameters {
    Field vector { type = Vector3 }
    Field apply { type = Trigger }
  }
}

```

2. View/Controller: defines a GUI control that shows the value of the existing `vector` field:

```

Window {
  Box Example {
    Field vector {}
    Button apply {}
  }
}

```

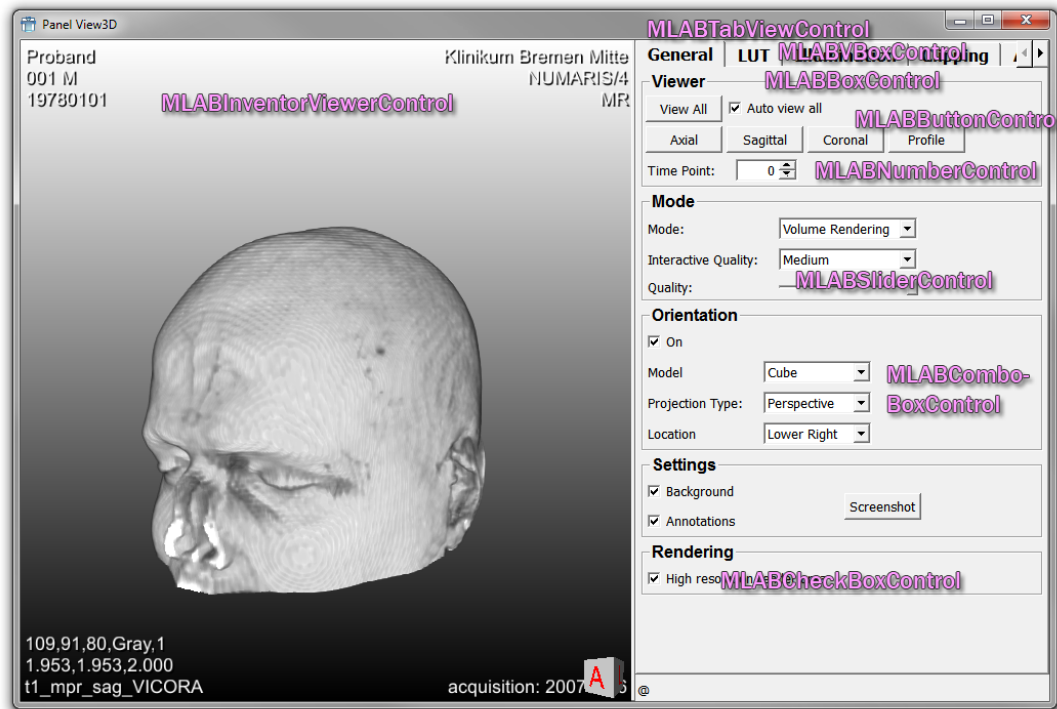


Note

Admittedly, it's a bit misleading that the `Field` is also used in the `Window` section for the field controls.

From the point of view of the MDL controls, the `View3D` panel looks like this:

Figure 11.5. View3D Panel with C++ Class Names of Included MDL Controls for Scripting



11.1.3. MDL GUI definition

The GUI definition of a module is written into the `.script` file of a module.

The **Interface** section is foremost used for Macro modules to declare fields. It is also used to declare extra fields of C++ modules or fields that should be kept persistent:

- input fields
- output fields
- parameter fields

The fields defined here may also forward existing fields of sub modules (internal fields) instead of defining new ones.

Declaring parameter fields of a C++ module has the advantage that these fields can be made persistent to save their values along with the network. (Normally, internal module states are not saved when the network is saved.)

The **Description** section is optional and can be used for all module types. It contains definitions for:

- Parameter field ranges (min/max)
- Persistence
- Editability

The **Commands** section is used to add Python script files and commands. (For details of the module initialization, see MDL Reference, chapter “Commands”.)

The **Window** section allows defining the GUI for the available Interface elements. The possible controls can be split into the following groups:

- Input controls for viewing and editing field values (Field, CheckBox, etc.)
- Layout controls (Vertical, Horizontal, Box, etc.)
- Decoration controls (Label, Image, Separator, etc.)

See the MDL Reference for a complete list of available controls.

The number of Window sections — which is the number of panels for a module — is unlimited. See View3D as an example for multiple panels.

11.1.4. A Note on Fields in Scripting Interfaces

“Field” has two meanings in MeVisLab:

1. The Field in the Parameters section declares a field for a module.

```
Interface {
  Parameters {
    Field fieldName { type = String }
  }
}
```

2. The Field control in the Window section defines a GUI control.

```
Window {
  Category {
    Field fieldName {}
  }
}
```

In scripting these objects have different class names:

1. MLABField
2. MLABFieldControl

Fields from a Scripting Perspective

Fields can be accessed from scripting. Take this integer field as an example:

```
Interface {
  Parameters {
    Field intFieldName { type = Int }
  }
}
```

Getting and setting the value:

```
ctx.field("intFieldName").value = 13
```

Getting and setting the value as a string (typically used for serialization purposes):

```
ctx.field("intFieldName").stringValue = "13"
```

To force notification of all field listeners:

```
ctx.field("intFieldName").touch()
```

FieldListener

The FieldListener binds scripting commands to a field. The command will always be called when the field issues a notification to its listeners.



Note

There is an important difference between FieldListeners defined in the Commands section and the ones defined in the Window section: The former is created when the module is created, and thus is always available. The latter is only created when the window is created, and it is destroyed when the window is destroyed. This will be explained in the examples below.

Example:

```
// triggerButton Field is already defined in Interface/Parameter
Interface {
  Parameters {
    Field triggerButton { type = Trigger }
  }
}

Commands {
  source = $(LOCAL)/ExampleToggleButton.py
  FieldListener triggerButton { command = callGlobal }
}
```

Touching the trigger field in Python will cause a notification and the field listener will call the given command, which is “callGlobal” in this example:

```
ctx.field("triggerButton").touch()
```

When “callGlobal” is called, there is no association to a window. The current window ID is 0, which means there is no current window. This means if you call `ctx.control("controlName")` in it, then MeVisLab will print an error that it cannot find the control.

If the field listener is defined inside of a window, it is only active when the window is actually created.

```
Window {
  Category {
    FieldListener triggerButton { command = callLocal }
  }
}
```

When “callLocal” is called, the current window ID is set to the window in which the field listener is defined. Calling `ctx.control("controlName")` in it will look inside this window for the control. This also means, that control names must only be unique inside a window, and can be reused in multiple windows of the same module.

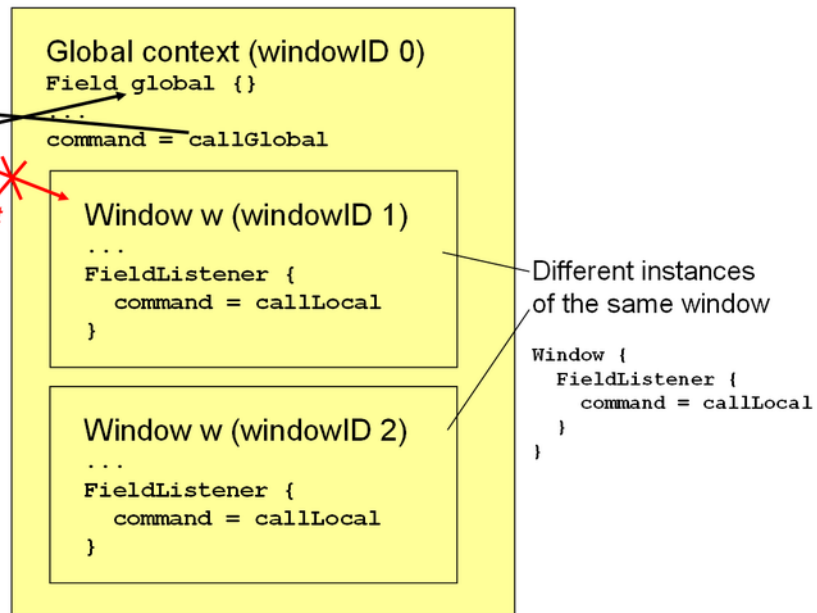
Figure 11.6. Command Execution Context

Command execution context

Python scripting

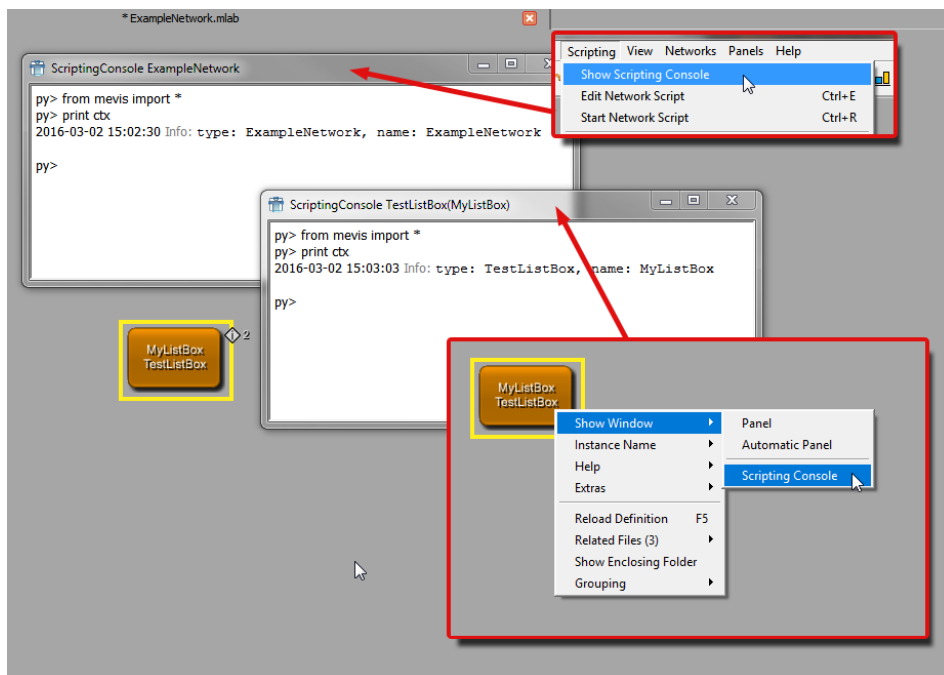
```
def callGlobal():
    ctx.field("global")
    ctx.control("w")
```

*not known in
global context*

**Accessing Controls from the Scripting Console**

From the scripting console, looking up a control only works when the correct context is set. First, the module context must be right. The scripting console must be opened from the context menu of a module to have it as the current context. The scripting console that can be opened from the **Scripting** → **Show Scripting Console** menu has the current network and not the selected module as context. Second, the window ID is 0 by default. To look up a control the window ID must be set accordingly. For debugging purposes, the function `ctx.controlDebug("controlName")` can be used in the scripting console as an alternative to setting the current window ID. It looks for the control in all open windows.

The following figure demonstrates that the global scripting console is not associated to a selected module in the network. Although the panel of the `TestListBox` module is open, the control cannot be found. It can be found in the scripting console of the module itself.

Figure 11.7. Contexts of the Scripting Console

11.2. Developing the ExampleToggleButton

In the following section, we will create a new Macro module with a simple on/off button. This is a standard use case for toggling parameters (visible or invisible).

- [Section 11.2.1, “Creating the Macro Module”](#)
- [Section 11.2.2, “Defining the Interfaces”](#)
- [Section 11.2.3, “Programming the Button Action in Python”](#)
- [Section 11.2.4, “Referencing the Command in the MDL Script”](#)
- [Section 11.2.5, “Persistent Field Values”](#)

11.2.1. Creating the Macro Module

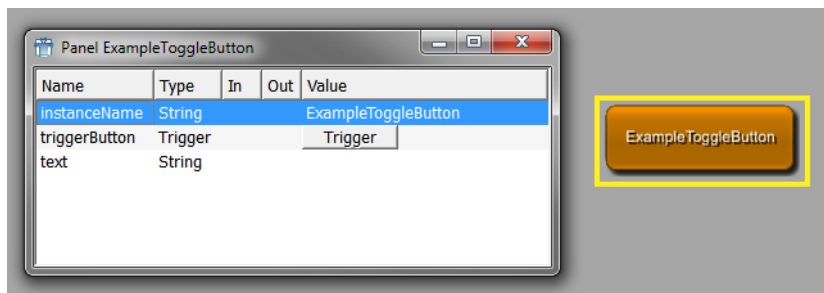
1. First of all, make sure that you have a user package defined as described in [Section 8.2, “Creating a User Package for Your Project”](#) or create it now.
2. Then run the Project Wizard and select the link **Macro Module**. This starts the Wizard for Macro Modules. Enter the following:
 - **Name:** ExampleToggleButton
 - **Keyword:** examples
 - **Target Package:** Example/General
 - **Project:** ExampleToggleButton
3. Click **Next** to proceed.
4. On the dialog **Macro Module Properties**, click **Add Python file** to have one created.

5. On the dialog **Module Field Interface**, the fields of the module can be defined (more fields can be added later).
 - a. Click **New** to create a new field, then enter the following:
 - **Field Name:** triggerButton
 - **Field Type:** Trigger
 - **Field Comment:**
 - **Field Value:**
 - b. Click again on **New** to create a second field:
 - **Field Name:** text
 - **Field Type:** String
 - **Field Comment:**
 - **Field Value:**
6. Click **Create** to create the module.

In the default file browser of your system, the folder `{packagePath}/Modules/Macros/ExampleToggleButton` is opened.

7. Click **Close** to finish the creation of the macro and to reload the module database to make the new module available. Type “Exam” to have the search deliver a list of available modules, and instantiate `ExampleToggleButton`. Double-click the module to open its automatic panel.

Figure 11.8. ExampleToggleButton



11.2.2. Defining the Interfaces

The automatic panel shows all defined fields. In the next step, we will edit the additional panel.

Right-click the module and look at **Related Files**. Click the script file `ExampleToggleButton.script` to open it in the integrated text editor MATE.

The script file looks as follows:

```
Interface {
  Inputs {}
  Outputs {}
  Parameters {
    Field triggerButton {
      type = trigger
    }
  }
}
```

```

    Field text {
        type = String
        value = ""
    }
}

Commands {
    source = $(LOCAL)/ExampleToggleButton.py
}

```

In a first step, we add the `Window` section to create a visible panel.

```

Window {
    Category {
        Box {
            Button triggerButton { title = "On/Off" }
        }
    }
}

```

After any changes in the `.script`, save the file, select the module in MeVisLab, and press **F5** to reload the module. After the first addition of the `Windows` section, double-click the module to open the new panel. Panels that are already open are automatically updated upon reload with **F5**.

The box draws a simple frame around the element, usually with a title. By default, the title is the tag value of the `Box` tag.

```

Box "A button" {
}

```

It can also be substituted by an explicit title element:

```

Box "A button" {
    title = "Yes, a button"
}

```

At this point, the button has no effect yet. The action tied to the button will be added as a the Python command in the next section. The `Label` uses the `text` field for its title.

```

Window {
    Category {
        Box {
            Button triggerButton { title = "On/Off" }
            Label { titleField = text }
        }
    }
}

```

11.2.3. Programming the Button Action in Python

In the Python script, the toggling of the button would need to have an effect, in this case it changes the label text.

Initially, the variable `toggleState` is `False`; upon pressing of the button it is toggled. If the `toggleState` gets `True` by pressing the button, then the label text is set to "On", otherwise it is set to "Off".

```

# Variable
toggleState = False

# Called when button pressed
def buttonPressed():

```

```
global toggleState
toggleState = not toggleState
if toggleState:
    ctx.field("text").value = "On"
else:
    ctx.field("text").value = "Off"
```

The value of the text field, which is used as the title field of the label, is set depending on the `toggleState` variable.

11.2.4. Referencing the Command in the MDL Script

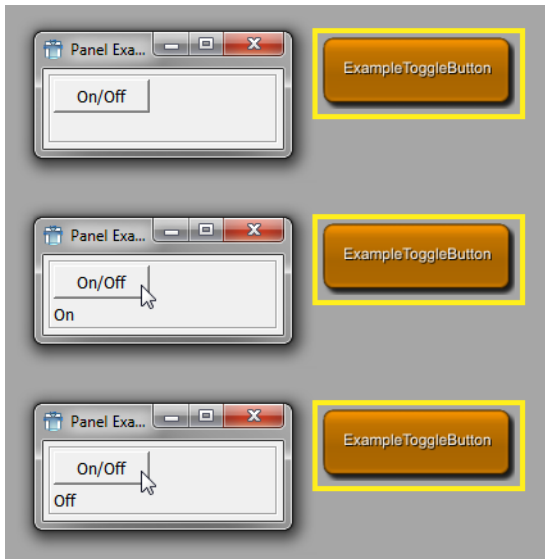
There are two ways to handle the button press. First, it is possible to use a trigger field and a `FieldListener`. Note that a `FieldListener` in the `Commands` section would not have the required window context to access any controls in the window. However, since the `buttonPressed()` function only accesses a field, this would also work in this case:

```
Window {
  FieldListener triggerButton {
    command = buttonPressed
  }
  Category {
    Box {
      Button {
        triggerButton { title = "On/Off" }
        title = "On/Off"
      }
      Label { titleField = text }
    }
  }
}
```

Second, the button press can be directly handled without a field:

```
Window {
  Category {
    Box {
      Button {
        title = "On/Off"
        command = buttonPressed
      }
      Label { titleField = text }
    }
  }
}
```

The example works now. However, the `toggleState` value is not persistent. If the network is closed and reloaded later, it is again initialized with `False`. The next section will explain how values can be made persistent.

Figure 11.9. ExampleToggleButton

11.2.5. Persistent Field Values

Assume that the value of the variable `toggleState` needs to be persistent, i.e., its last value should be restored when the network is loaded at another time. We can use a persistent field for this. The fields current value will be stored when the MeVisLab network is saved. In a first step, a Bool field for it is added to the `Interface` section:

```
Interface {
  Inputs {}
  Outputs {}
  Parameters {
    Field text { type = String value = Off}
    Field toggleState { type = Bool value = false}
  }
}
```

For the Python script, it would mean a rewrite resulting in:

```
# Called when button pressed
def buttonPressed():
    toggleState = ctx.field("toggleState")
    toggleState.value = not toggleState.value
    if toggleState.value:
        ctx.field("text").value = "On"
    else:
        ctx.field("text").value = "Off"
```

The value of the `toggleState` field is now persistent — if `ExampleToggleButton` would be used in a network, its last state would be saved with the network.

11.2.6. Implementing a Keyboard Shortcut

For a button, a keyboard shortcut could be implemented by adding an `Accel` control. In our example, we add the key combination **ALT+Q**.

```
Interface {
  Inputs {}
  Outputs {}
  Parameters {
    Field text { type = String value = "" }
  }
}
```

```

    Field toggleState { type = Bool value = false}
    Field triggerField { type = Trigger }
  }
}

Commands {
  source = $(LOCAL)/ExampleToggleButtons.py
}

Window {
  Accel {
    key = ALT+Q
    field = triggerField
    command = buttonPressed
  }
  Category {
    Box {
      Button {
        name = triggerButton
        title = "On/Off"
        command = buttonPressed
      }
      Label { titleField = text }
    }
  }
}

```

The key has to be defined before the other GUI controls for which it should be used, so it is best entered in the beginning of the `Windows` section. The panel has to be active for the shortcut to have an effect.

11.2.7. Arranging Multiple Buttons

In a `Box` control, the default layouter is a `Vertical` control:

```

Box {
  Button {
    name = triggerButton
    title = "On/Off"
  }
  Button {
    name = trigger2Button
    title = "Blue/Green"
  }
  Button {
    name = trigger3Button
    title = "Big/Small"
  }
}

```

The layouter can be changed, see the MDL Reference for a list of possible layouters. For example, a grid can be set:

```

Box {
  layout = grid
  Button {
    name = triggerButton
    title = "On/Off"
    x = 0 y = 0
  }
  Button {
    name = triggerButton
    title = "Blue/Green"
    x = 1 y = 1
  }
}

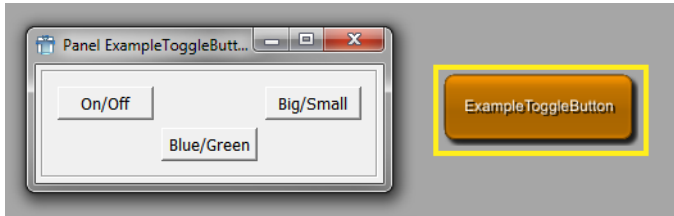
```

```

Button {
  name = trigger3Button
  title = "Big/Small"
  x = 2 y = 0
}

```

Figure 11.10. Buttons in a Grid



11.2.8. Auto Layouting with the AlignGroups Control

Have a look at the module `TestLayouter!`

11.2.9. Prototypes for Controls

It is possible to define overwrite default values for MDL controls. Prototypes are existing controls with different default values. For example, if all following occurrences of the Vertical control should by default expand in horizontal and vertical directions, the prototype declaration would look like this:

```

Vertical {
  style {
    Prototype Vertical {
      expandX = yes
      expandY = yes
    }
  }

  // the following vertical now has the defaults as given above
  Vertical {
    Label { title = "test" }
  }
}

```

Prototypes can be defined for all MDL elements.

Prototypes do not inherit from each other, so if you overwrite, e.g., Vertical, you loose all the default tags that are defined in the default prototype.

Styles inherit the prototypes from the style they are derived from, so you can overwrite individual prototypes without affecting other prototypes from the default style.

See the module `TestPrototypes` for an example.

11.2.10. Designing Larger GUIs

On the automatic panel of a module, all parameters are listed. Therefore, there is no necessity to add all parameters as fields to your GUIs. Focus on those fields that the user needs to set or see.

If the module has a high number of fields, the controls can be arranged on the panel, e.g., by using tabs or sub-panels.

As modules may have multiple windows, the GUI can be split into various panels. This is recommended for settings that are possible, but do not relate strongly to other, more important settings of the module.

For sharing parts of the GUI between panels, the Panel control can be used. It clones a defined subregion of a module's Windows section.

Excerpt from the View3D script file:

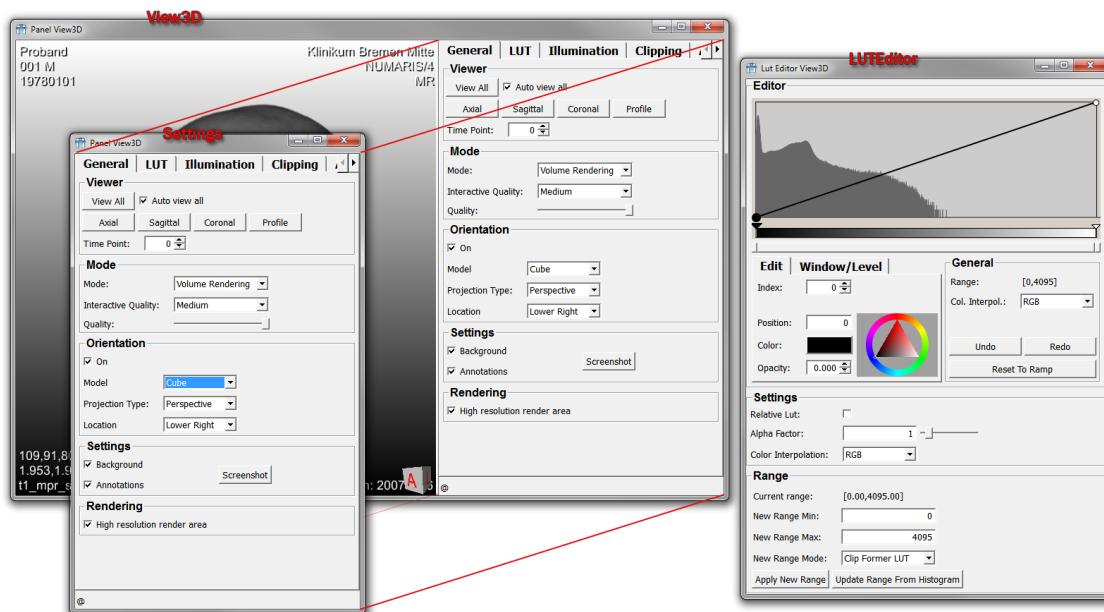
```
Window View3D {
  Vertical {
    expandX = NO
    panelName = Settings
    TabView {
      TabViewItem General {
        Box Viewer {
          Horizontal {
            expandX = no
          }
        }
      }
    }
  }
}
```

The thus defined panel “Settings” can be reused in a panel of its own.

```
Window Settings {
  Panel {
    panel = Settings
  }
}
```

The Panel control also clones all FieldListeners contained in the cloned code, so that a cloned panel should work like the original one. The window one gets when calling window() in the context of the cloned script will be the window in which the Panel is, in this case the View3D window.

Figure 11.11. View3D Panels with the Panel Control



11.3. MDL Styles

As every panel needs some kind of style — or visual theme — for display, MeVisLab provides the concept of styles. Styles are color and font schemes that can be derived from each other, and the base of all styles is the `_default` style.

First, it is important to know that there are two modes which affect what style is chosen: the `Panel` mode and the `Application` mode. If you open panels inside of MeVisLab by double clicking on a module, the panels are created in `Panel` mode. The `Application` mode is used when running a macro module as an application, either via **Scripting** → **Start Network Script** or from the commandline.

Two default styles are predefined, one for each mode. Both are directly derived from `_default`:

- `Panel.default` is used in `Panel` mode (gray style).
- `Application.default` is used in `Application` mode (blue/green style).

The predefined style names from above consist of a prefix (“Panel”/“Application”) and the actual style name (“default”). This allows for using the same style name for two different modes. The actual prefix is detected by MeVisLab, this will be explained below.

11.3.1. How to Use MDL Styles

Styles can be set and locally derived in MDL controls. They affect the current control and also recursively all its children.

For example, to specify the style for a single Window control the `style` tag can be used like this:

```
Window {
  style = ExistingStyleName
  ...
}
```

To locally derive from an existing style, one can write:

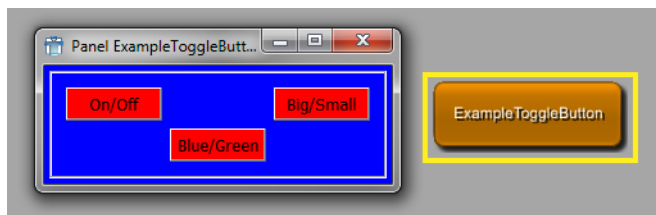
```
Window {
  style NewLocalStyle {
    derive = Application.default // inherit the default Application style
    colors {
      bg = blue // background color
      fg = yellow // foreground color
    }
  }
  ...
}
```

The `derive` tag can be omitted to inherit from the style that is currently active, regardless of which that is (see also the style stack, which is mentioned below in “How MeVisLab Applies the Styles”). It is also possible to omit the style prefix, in which case MeVisLab uses the detected one.

A local style can also be anonymous:

```
Window {
  style {
    colors { bg = blue button = red }
  }
  Box {
    layout = Grid
  }
  ...
}
```

Figure 11.12. Redesigned Panel




Tip

MATE offers auto completion for style attributes. See the figure below.

Figure 11.13. Entering Style Settings

```
Window {
  style {
    colors { bg = bl button = white }
  }
  Box {
    layout = Grid
    Button {
```


How MeVisLab Applies the Styles

Before a window is created, MeVisLab initializes a stack with the default style. The name of it is “<style prefix>.default”, e.g., `Application.default` (see below how MeVisLab determines the style prefix). Then it recursively creates all MDL controls, beginning with the Window control.

For each control, MeVisLab checks if either a style is specified or if a style is derived locally (local styles are explained below). If one of both is true, then it pushes this style onto the stack and applies it to the widgets that are created by the control. This style is also applied to all sub controls, unless they push another style themselves. After the control is created, the style is popped again.

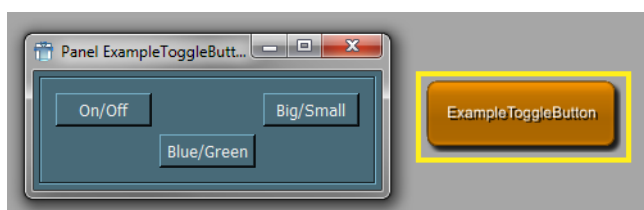
MeVisLab looks up styles as follows:

1. If the style name does not contain a dot, it prepends the style prefix and uses it if it exists.
2. If the style name does not contain a dot, it prepends the fallback style prefix and uses it if it exists.
3. Use the given style name as it is to look up the style.

How MeVisLab Determines the Style Prefix

In `Application` mode, the style prefix is the name of the application macro module. The fallback style prefix is either the value of the MDL preferences variable `ApplicationStyle`, or “Application” if `ApplicationStyle` is not specified.

In `Panel` mode the style prefix is either the value of the MDL preferences variable `PanelStyle`, if it is given, or “Panel”. The fallback style prefix is always “Panel”.

Figure 11.14. ExampleToggleButton with Application Style Panel

To style a window like above for testing, use the `style` tag as follows:

```
Window {
  style = Application.default

  Category {
    Box {
      ...
    }
  }
}
```

11.3.2. Defining Global Styles**How to Define MDL Styles**

A style is defined globally using the `DefineStyle` tag. The style name may include a prefix separated by a dot from the actual name, but it is not required (see “How MeVisLab Applies the Styles” above on how MeVisLab looks up styles by name). There are two possibilities to position the definition of a named style:

- if globally defined in any `*.def` file, this style will be available to all windows of all modules under the given name.
- if defined inside of a window, this style is only available inside of that window.

11.3.2.1. How to Define a Global Style

```
DefineStyle AnyStyleName {  
    derive = _default  
    colors {  
        bg = black  
        fg = black  
        button = black  
    }  
}
```

11.3.2.2. How to Define a New Default Style for Application Macro Modules

It is possible to define new default styles for application macro modules. The style definition does not have to be in the same `.def` file. You can have one global `.def` file where you define your styles. If you want to use a style as default in multiple application macro modules, you can derive default styles with the macro module names as prefix:

```
DefineStyle BaseStyle {  
    derive = _default  
    ...  
}  
  
DefineStyle Macro1.default {  
    derive = BaseStyle  
}  
  
DefineStyle Macro2.default {  
    derive = BaseStyle  
}  
...
```

11.3.3. Creating Custom MDL Controls

Custom MDL controls can be created.

The following steps would be necessary to create a MDL control:

1. Define the control in a `.def` file (MDL) under the `Modules` directory of a package. This will make the control available and extend the existing MDL validator.
2. Implement the control, either in C++ (`.h`, `.cpp`) or in Python (`.py`). Put the files under the `Sources` directory of a package.

Examples for this are in the `MeVisLab/Examples` package. It includes the controls

- `ColorChooserExampleControl`,
- `DiagramExampleControl`,
- `PythonControlExample`, and
- `DoubleSpinBoxExample`

Their definitions exist in `Examples/Modules/Controls/`, their implementations in `Examples/Sources/Controls/` (C++) and `Examples/Modules/Scripts/python/` (Python). The modules `ColorChooserExampleControlTest`, `DiagramExampleControlTest`, and `PythonExampleControlsTest` demonstrate using the controls.

For example, the `ColorChooserExample` files are:

- The `WidgetControl` definition, which includes a reference to the DLL that contains the compiled C++ implementation:

```
Examples/Modules/Controls/ColorChooserExampleControl.def
```

- The C++ implementation:

```
Examples/Sources/Controls/MLABColorChooserExampleControl/
mlabColorChooserExampleControl.h
```

```
Examples/Sources/Controls/MLABColorChooserExampleControl/
mlabColorChooserExampleControl.cpp
```

The `PythonControlExample` files are:

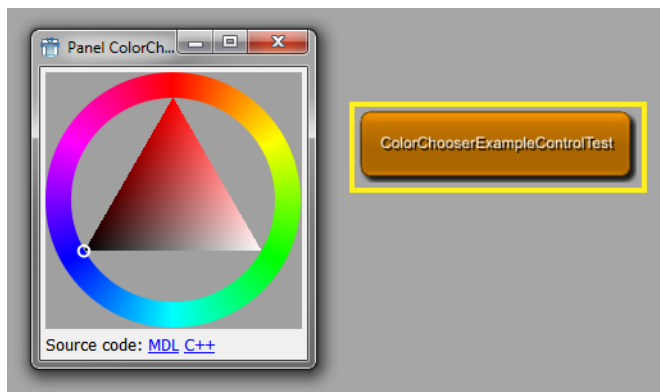
- The `WidgetControl` definition, which includes a reference to the implementing Python module:

```
Examples/Modules/Controls/PythonControlExamples.def
```

- The Python implementation:

```
Examples/Modules/Scripts/python/PythonControlExample.py
```

Figure 11.15. Color Chooser Example Control



11.4. Customize GUI Appearance Using Qt Style Sheets (CSS)

By referencing Qt Style Sheet files in the MDL, the underlying Qt widgets can be styled from the MDL, including tab bars, radio buttons, list view items, etc. For a list of available GUI elements, see [Qt Widgets](#).

The method has two major drawbacks:

- The developer needs to learn something about the underlying Qt widgets.
- The solution somewhat depends on the underlying implementation of the MDL, which could change over time, for example by using different Qt Widgets internally or just by using a newer Qt version.

The styling works for Qt widgets that are derived from `QWidget`.

Note that it does not work for the MDL controls themselves, because they have no own visual representation, but they aggregate Qt widgets.

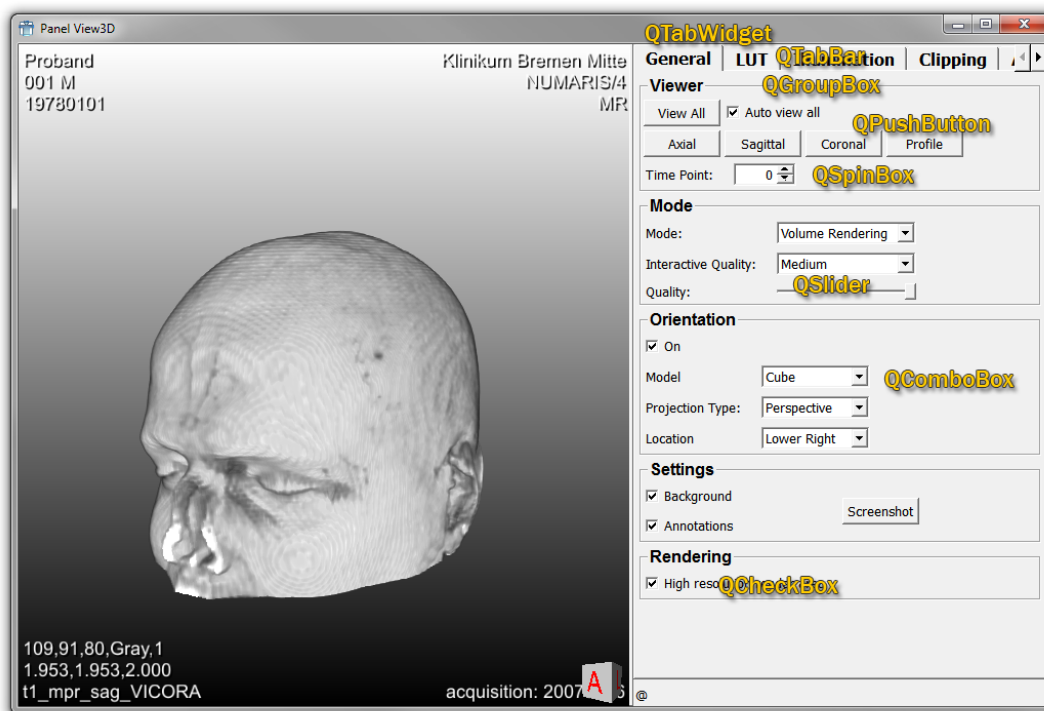


Note

Avoid mixing the MDL styles and the CSS styles within the same MDL controls, because MDL styles manipulate the QPalette of the underlying widgets and the style sheets override the QPalette.

Let's have a look at the panel we already know from its element and control names.

Figure 11.16. View3D Panel with Qt Widgets



Like for all modules, a module using the CSS style sheets would have a `.script` in which the elements of the GUI are defined. In addition, the stylesheet is referenced.

```
Interface {
  Parameters {
    Field name {
      type = string
    }
  }
}

Window {
  styleSheetFile = $(LOCAL)/TestStyleSheets.css
  Vertical Tab1 {
    expandY = yes
    Box Box {
      Field name { }
    }
    Box Buttons { layout = Horizontal
      Button { title = Something }
      Button { title = Testing }
    }
    Box ListView { layout = Horizontal
```

```
    ListView {
        values = "Column1,Column2$Value1,Value2$Value3,Value4"
        columnSeparator = ,
        rowSeparator = $
    }
}
SpacerY {}
}
Vertical Tab2 { expandY = yes
}
}
```

The styling of the elements is then done in the style sheet.

Excerpt from `TestStyleSheets.css`:

```
QTabWidget {
    background: white;
}

QStackedWidget {
    background: white;
}

QTabWidget::pane { /* The tab widget frame */
    border-top: 2px solid #C2C7CB;
}

QTabWidget::tab-bar {
    left: 5px; /* move to the right by 5px */
}

/* Style the tab using the tab sub-control. Note that
   it reads QTabBar _not_ QTabWidget */
QTabBar::tab {
    border-image: url("${LOCAL}/style/tab1.png") 10 10 2 10;
    border-top: 10px transparent;
    border-right: 10px transparent;
    border-bottom: 0 transparent;
    border-left: 10px transparent;
    min-width: 8ex;
    padding: 2px;
}

QTabBar::tab:selected, QTabBar::tab:hover {
    border-image: url("${LOCAL}/style/tab2.png") 10 10 2 10;
}

QPushButton {
    border-image: url("${LOCAL}/style/button.png") 6 10 6 10;

    border-top: 6px transparent;
    border-bottom: 6px transparent;
    border-right: 10px transparent;
    border-left: 10px transparent;
}
```

To inspect the QWidgets of a panel you can use the Widget Explorer. It shows the widget hierarchy of all windows and displays some information about the widgets, for example if the widget is directly owned by a control. The Widget Explorer also allows for dynamically altering the style sheets.

Note that you cannot use the control names for [CSS ID selector](#), because it does not set the object name on the widget, but on the control. You can set the object name of the widget with the MDL tag `widgetName`.

Chapter 12. Excursion: Image Processing in ML

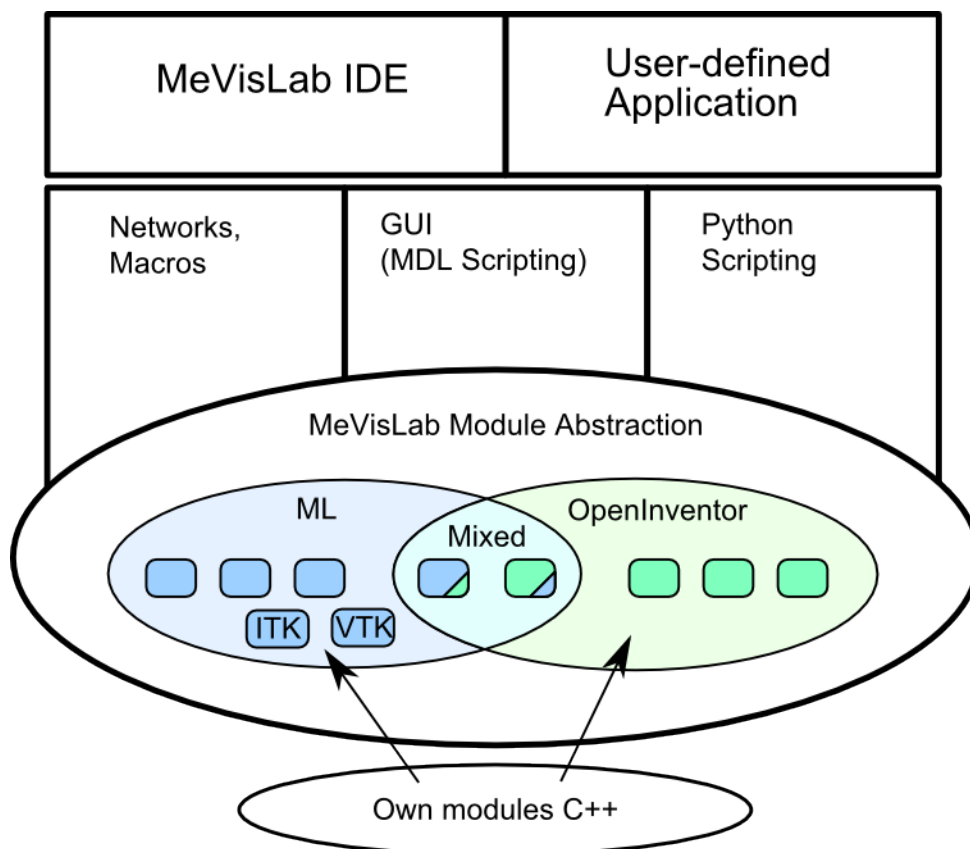
12.1. Some Advanced Information on Image Processing

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 ML Guide, chapter “Image Processing Concepts”. Please refer to this document for further information.

12.2. Structure of MeVisLab

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

Figure 12.1. 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 language Python with full access to the modules and GUI widgets, including the ability to generate new modules or widgets.

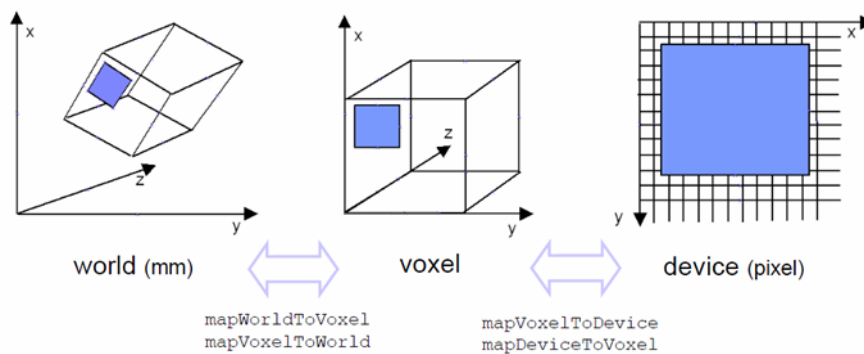
Based on these functionality one can build, test and evaluate own applications with the integrated development environment and — with the proper license — generate own installers with standalone applications.

12.3. Coordinate Systems

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

- World coordinates
- Voxel coordinates
- Device coordinates

Figure 12.2. 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)

12.4. 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 12.3. 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

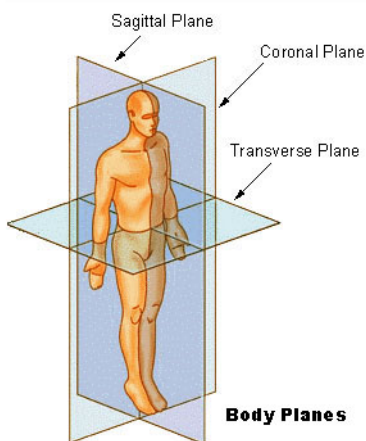
- Computing the voxel volume: `getVoxelSize()` returns voxel spacing in x, y and z. The product of these values is not the voxel volume if the voxel-to-world-matrix is not orthogonal. Solution: Use the absolute value of the matrix determinant instead.
- Inventor using row vector conventions: ML and MeVisLab 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.

12.5. 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 12.4. 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:

- `DirectDicomImport` is a module for DICOM import that reads images directly from slices, without an intermediate representation.

It has a lot of options to control the import process, which can, e.g., determine which slices are combined into an image stack.

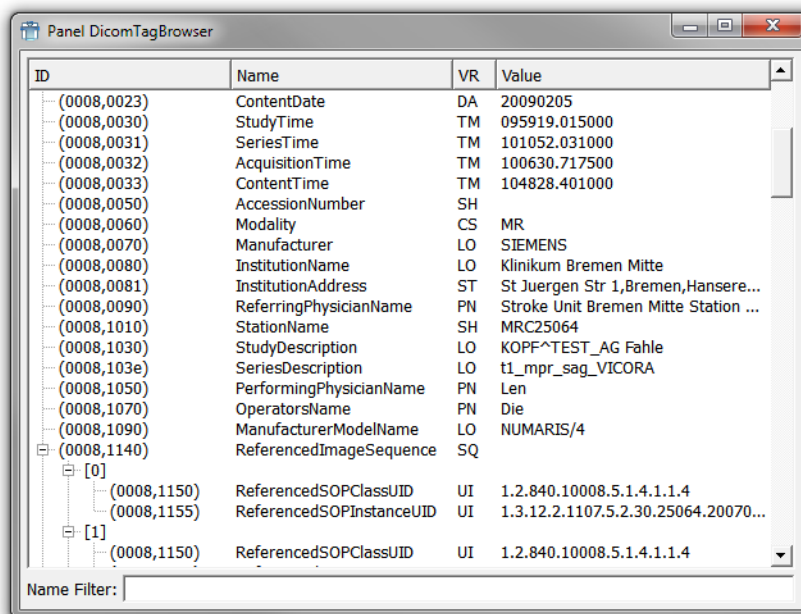
- `DicomImport` is a new module for DICOM import. The new implementation does not yet provide all known functionalities from `DirectDicomImport`, most of them will be added in future releases. Its main advantage is that the import process is faster and happens asynchronously.
- You can view the image-wide DICOM tags with the module `DicomTagBrowser`.
- 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**.
- Saving of loaded DICOM data to the filesystem or sending to a PACS (Picture Archiving and Communication System) is possible with the `DicomTool` macro module.
- Basic support for querying and receiving DICOM data from a PACS is available through the `DicomQuery` and `DicomReceiver` modules.



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 12.5. The DICOM Tag Browser



12.6. 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 12.6. Image Properties for an Ideal Image

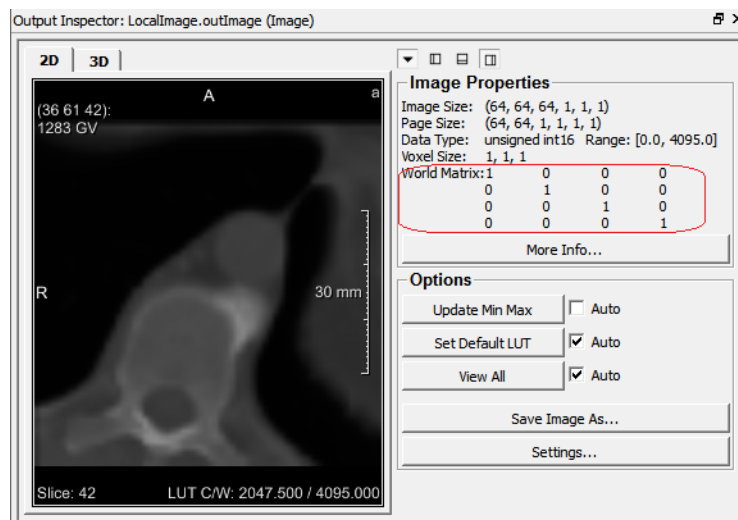
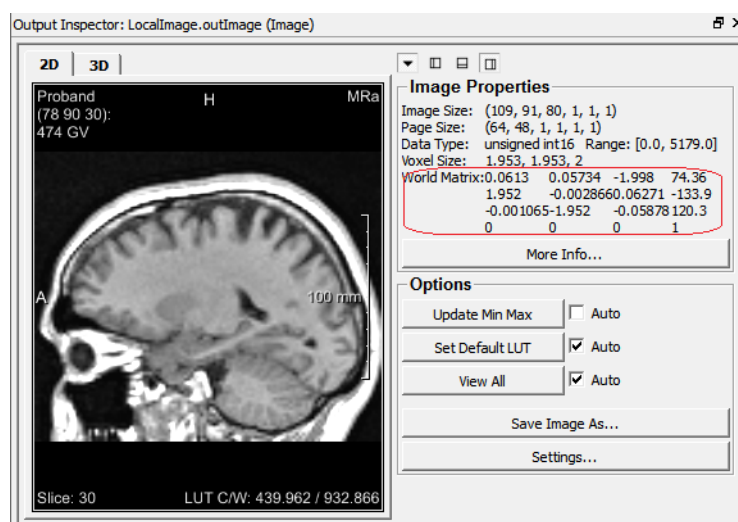


Figure 12.7. Image Properties for a Sagittal Image



Note

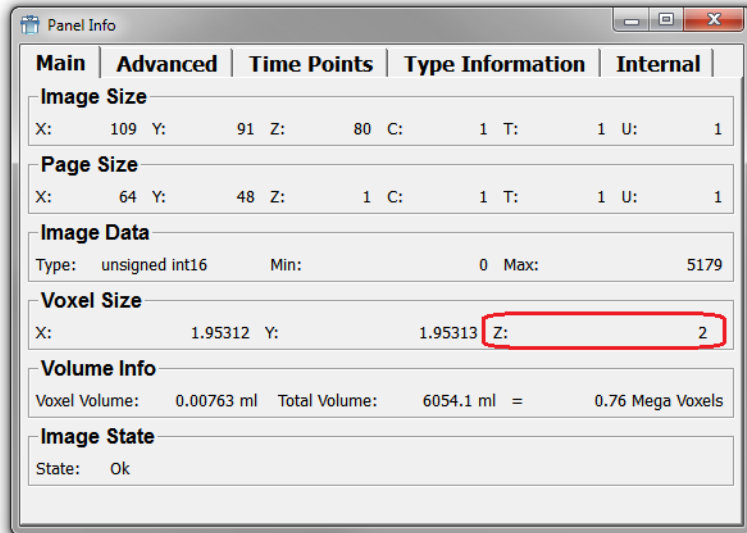
In DICOM, the voxel thickness does not necessarily correspond to the distance between slices. In MeVisLab however, the calculated voxels close the slice distance.



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 12.8. Image Properties in the `Info` Module



12.7. 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 `dcmdump` 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 `dcmtool` (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 (for example 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.



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.

12.8. 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 ML Guide, chapter “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 128x128x1x1x1x1. 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 13. 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. In the following chapter, you will find information on:

- [Section 13.1, “Module and Connection Specifics on the C++ Level”](#)
- [Section 13.2, “Some Tips for Module Design”](#)
- [Section 13.3, “Programming Examples”](#)

13.1. Module and Connection Specifics on the C++ Level

ML modules on the C++-level:

- Image processing modules are objects derived from class `Module` defined in the ML library and therefore are also called ML modules.
- Image inputs and outputs are connectors to objects of class `PagedImage`, 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 Module 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 PagedImage defined in the ML library.

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

13.2. Some Tips for Module Design

13.2.1. 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.

13.2.2. 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 an Inventor scene. Examples: `VoxelizeInventorScene`, `SoExaminerViewer` (hidden functionality).

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

13.2.3. Tips for Module Testing

First, test your modules and networks with the MeVisLab TestCenter, see [Chapter 16, Using the TestCenter](#) for an introduction and the TestCenter Reference for further information.

After being done with the 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 offer a well-designed panel for future users
- should come with a useful help and example network

13.3. 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/Examples/Sources/Examples/ML/...`
- **Modules:** `Packages/MeVisLab/Examples/Modules/Examples/ML/...`

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



Tip

See the chapter [Section 14.3, “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.

- **AddImagesExample** (Class: `AddImagesExample`; DLL: `AddImagesExample`)
Startup example for ML module programming.
- **ProcessAllPagesExample** (Class: `ProcessAllPagesExample`; DLL: `ProcessAllPagesExample`)
Shows how to process all pages using multi-threading and the `TypedOutputHandler`.
- **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.

- **AsyncProcessAllPagesExample** (Class: `mlAsyncProcessAllPagesExample`; DLL: `MLBackgroundTasksExamples`)

This module (like some others in the same DLL) demonstrates how image processing tasks can be performed in the background of the main process so that the GUI stays responsive.

- **ReadDICOMTagExample** (Class: `mlReadDICOMTagExample`; DLL: `MLReadDICOMTagExample`)

Shows how to read DICOM tags from the internal image representation.

- **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.

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

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



Tip

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

Chapter 14. 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 14.1, “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 14.2, “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 14.3, “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 ML Guide.



Note

Developing C++ modules requires a C++ development environment being available on your computer, for example Visual C++ on Windows or Xcode on Mac OS X.



Note

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

14.1. 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 14.1.1, “Creating the Basic ML Module with the Project Wizard”](#)
- [Section 14.1.2, “Preparing the Project”](#)
- [Section 14.1.3, “Programming the Functions of the ML Module”](#)
- [Section 14.1.4, “GUI Creation/Optimizing”](#)
- [Section 14.1.5, “Creating an Example Network and Help File”](#)



Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Examples/Modules/GettingStarted/MLSimpleAddExample`, source files in `$(InstallDir)Packages/Examples/Sources/GettingStarted/MLSimpleAddExample`). The module can be added via quick search.

14.1.1. 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 8.2, “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
- **Target Package:** Example/General
- **Project:** SimpleAdd

Click **Next** to proceed.

Figure 14.1. Entering the ML Module Properties

Modules (C++)/ML Module

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: Image Choose ☐ Add reference to example network

Select Target Package

Package: * Example/General

Project Properties

Directory Structure: Classic

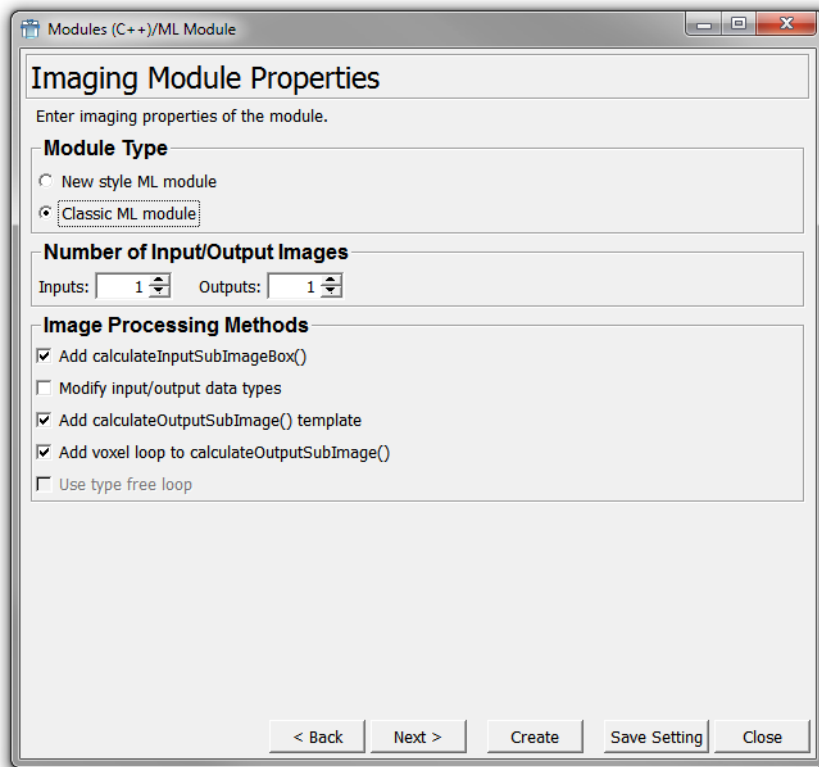
Project: * SimpleAdd Prefix: ML Select

☒ Include project files

* : Required fields

< Back Next > Create Save Setting Close

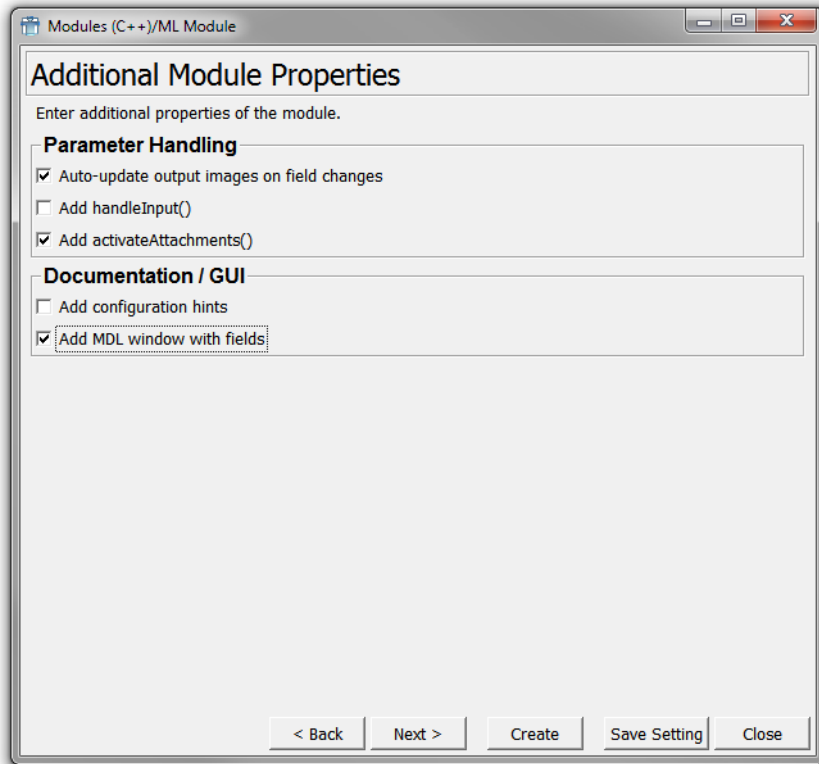
3. On the dialog **Imaging Module Properties**, the inputs and outputs as well as possible sample code can be added to the ML module.

Figure 14.2. Entering the Imaging Module Properties

Select the Module Type **Classic ML Module**. For information on the differences, see the MeVisLab Reference Manual, chapter “ML Wizard”.

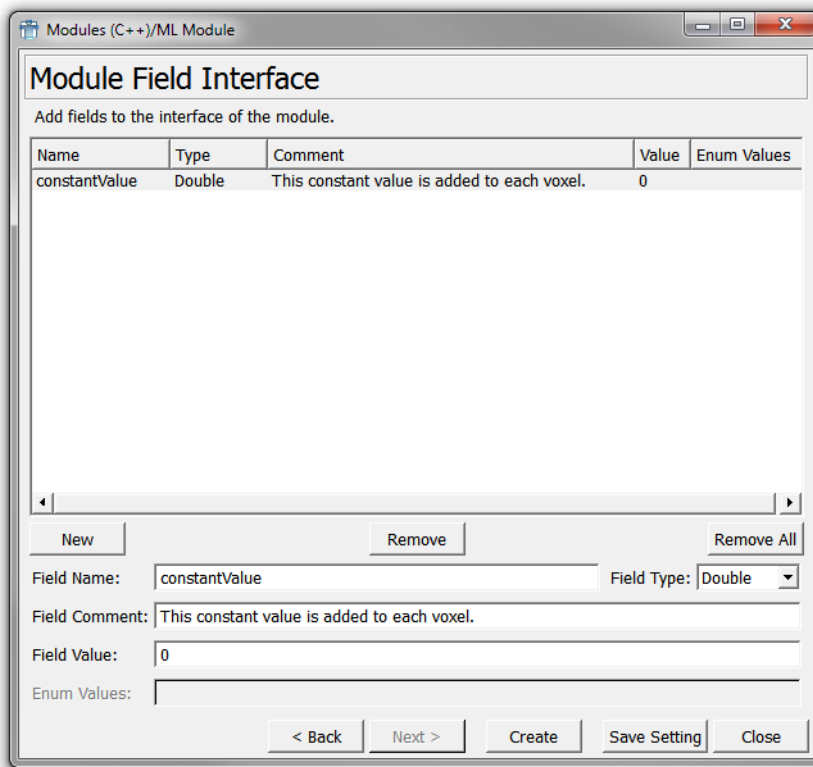
Enter the following settings:

- **Inputs:** 1
 - **Outputs:** 1
 - Check **Add calculateOutputSubImage() template**.
 - Check **Add voxel loop to calculateOutputSubImage()**.
4. On the dialog **Additional Module Properties**, the inputs and outputs as well as possible sample code can be added to the ML module.

Figure 14.3. Additional Module Properties

Make the following settings:

- Check **Auto-update output images on field changes** (adds `handleNotification`).
 - Check **Add activateAttachments()**.
 - Check **Add ML window with fields**.
5. 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 14.4. Entering the ML Module Properties — Fields

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.

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

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

- folder with the source code: `{packagePath}/Sources/ML/MLSimpleAdd`
- folder with the module's GUI definition: `{packagePath}/Modules/ML/MLSimpleAdd`



Note

For a full list of all created files and their contents, refer to the MeVisLab Reference Manual, chapter “ML Module — Created Files”.

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.

14.1.2. Preparing the Project

The Project Wizard creates a `CMakeLists.txt` file that describes the typical projects settings and used source files. This file can be translated manually with the [CMake tool](#) into a project file for your preferred C++ development tool. But most Integrated Development Environments (IDEs) nowadays can open CMake files directly.

Just make sure that the `MLAB_ROOT` environment variable is set on your system and points to the `Packages` directory of your MeVisLab installation, because this is used to resolve the reference to the 'MeVisLab' project.

For further documentation about our use of CMake see: [CMake for MeVisLab - Documentation](#).

14.1.3. Programming the Functions of the ML Module

Open the file `mlSimpleAdd.cpp`.



Note

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

14.1.3.1. Implementing `calculateOutputImageProperties`

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::calculateOutputImageProperties (int outIndex, PagedImage* outImage)
{
    // get the constant add value
    const MLdouble constValue = _constValueFld->getDoubleValue();

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

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



Note

`outIndex` is the index number of the output connector.

14.1.3.2. Implementing `calculateOutputSubImage`

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 from `*outVoxel = *inVoxel0;` to `*outVoxel = *inVoxel0 + constantValue;` so that the constant value is added to the value of the input voxel:

```
// Process all row voxels.
for (; p.x <= rowEnd; ++p.x, ++outVoxel, ++inVoxel0)
{
    *outVoxel = *inVoxel0 + constantValue;
}
```

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.

14.1.4. 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). Therefore, it is recommended to edit MDL files primarily with MATE. (More information on MATE can be found in the MeVisLab Reference Manual, chapter “MATE”.)
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 {
        margin = 3
        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

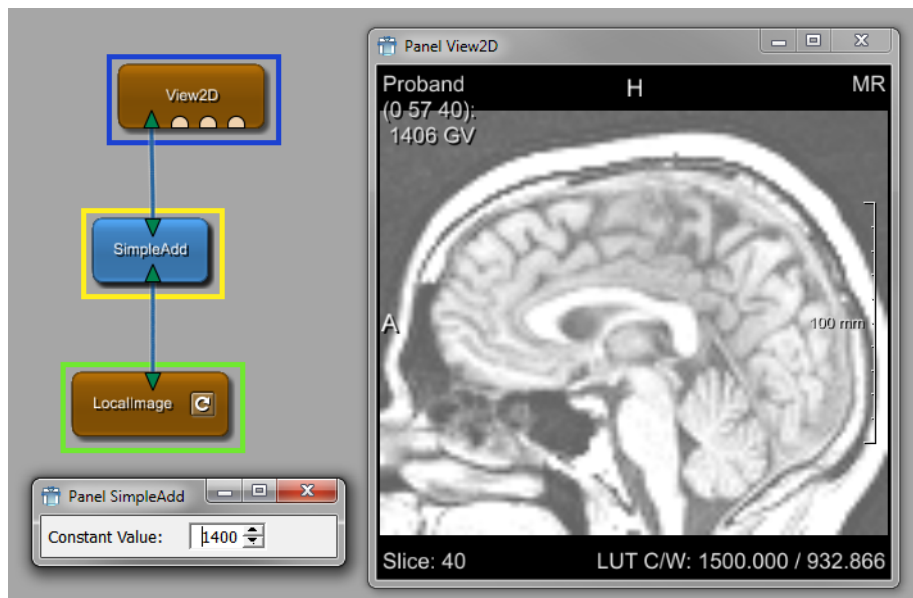
Congratulations, you have now implemented your first page-based and demand-driven ML image processing module!

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

14.1.5. 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 14.5. Example Network for SimpleAdd



4. To create the help, right-click the new module and select **Edit Help** from the context menu. This opens the integrated text editor MATE in a mode to edit a module's help file. See Section 27.9, "Module Help Editor" for more information.

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.

14.2. 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 14.2.1, "Creating the Basic ML Module with the Project Wizard"](#)

- [Section 14.2.2, “Editing the Header File of `SimpleAverage`”](#)
- [Section 14.2.3, “Editing the CPP File of `SimpleAverage`”](#)



Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Examples/Modules/GettingStarted/MLSimpleAverageExample`, source files in `$(InstallDir)Packages/MeVisLab/Examples/Sources/GettingStarted/MLSimpleAverage`). The module can be added via quick search.

14.2.1. 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 14.1.1, “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. **Target Package:** `Example/General`
 - f. **Project:** `SimpleAverage`

Click **Next** to proceed.

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

Select the Module Type **Classic ML Module**. For information on the differences, see the MeVisLab Reference Manual, chapter “ML Wizard”.

Enter the following settings:

- **Inputs:** 1
- **Outputs:** 1
- Check **Add calculateOutputSubImage() template**.
- Check **Add voxel loop to calculateOutputSubImage()**.



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 **Additional Module Properties**, check **Add MDL window with fields**.

Click **Next** to proceed.

- 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: {packagePath}/Sources/ML/MLSimpleAverage
- folder with the module's GUI definition: {packagePath}/Modules/ML/MLSimpleAverage



Note

For a full list of all created files and their contents, refer to the MeVisLab Reference Manual, chapter “ML Module — Created Files”.

- Reload the module database.
- Prepare the project, as described in [Section 14.1.2, “Preparing the Project”](#).

14.2.2. Editing the Header File of simpleAverage

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

```
//!
NotifyField* _updateFld;

size_t _numVoxels;
MLdouble _sumVoxelValues;
```

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

- Remove the following lines.

```
//! Sets properties of the output image at output outIndex.
virtual void calculateOutputImageProperties(int outputIndex,
                                           PagedImage* outputImage);
```

The virtual function calling `calculateOutputImageProperties` 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 `calculateOutputSubImage` template is necessary.)

14.2.3. 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.

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

```
SimpleAverage::SimpleAverage () : Module(1, 0)
```

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

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

```
// Handle changes of module parameters and input image fields here.
if (field == _updateFld)
{
    _numVoxels = 0;
    _sumVoxelValues = 0;

    processAllPages(-1);

    MLdouble result = 0;

    if (_numVoxels > 0)
    {
        result = _sumVoxelValues / static_cast<MLdouble>(_numVoxels);
    }
    _voxelValueAverageFld->setDoubleValue(result);
}
```

The code includes the important ML Module 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()`. The provided parameter '-1' causes the input image to be read-only for optimization reasons. For further information, see the ML Guide.

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

```
//-----
void SimpleAverage::calculateOutputImageProperties(int /*outputIndex*/,
                                                  PagedImage* outputImage)
{
    // Change properties of output image outputImage here whose
    // defaults are inherited from the input image 0 (if there is one).
}
```

4. In the method `calculateOutputSubImage(...)`, **remove** `outputSubImage` and `outputIndex` from the method's signature. Result:

```
template <typename T>
void SimpleAverage::calculateOutputSubImage (TSubImage<T>* ,
                                             int ,
                                             TSubImage<T>* inputSubImage0
                                             )
```

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

5. Replace the line:

```
const SubImageBox validOutBox = outputSubImage->getValidRegion()
```

with the line:

```
const SubImageBox inBox = inputSubImage0->getValidRegion();
```

Resulting in:

```
// Compute subimage of output image outIndex from input subimages.
const SubImageBox inBox = inputSubImage0->getValidRegion();
```

6. Remove the line

```
T *outVoxel = outputSubImage->getImagePointer(p);
```

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

8. Replace the line

```
*outVoxel = *inVoxel0;
```

with the lines:

```
_sumVoxelValues += static_cast<MLdouble>(*inVoxel0);
++_numVoxels;
```

Remove the `++outVoxel`, from the inner for-loop over the voxel row.

Resulting in:

```
// Process all row voxels.
for (; p.x <= rowEnd; ++p.x, ++in0Voxel) {
    _sumVoxelValues += static_cast<MLdouble>(*in0Voxel);
    ++_numVoxels;
}
```

9. At last, compile the project. Then restart MeVisLab so that the new module is registered and added to the module database.
10. In MeVisLab, instantiate the new module, right-click it and open the module's .script file.

In the .script file, enter the following lines before the `Window` section:

```
Description {
    Field voxelValueAverage { editable = No }
}
```

Setting the `editable` of a field to `No` (or `False` or `0`) has two consequences: firstly, the field is not editable by the user which makes sense, because the field should be set from the C++ code only, and secondly, the value of the field is not saved with the .mlab file which makes sense, because the value needs to be calculated in a live network.

14.2.4. 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 rounding error in the display.

**Tip**

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

14.3. 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 14.3.1, “Copying the Source Files”](#)
- [Section 14.3.2, “Editing and Recompiling the `CMakeLists.txt` File”](#)
- [Section 14.3.3, “Editing the Project in the Development Environment”](#)
- [Section 14.3.4, “Editing the Module Definition \(`.def`\)”](#)
- [Section 14.3.1, “Copying the Source Files”](#)

Per project, one DLL (`.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.

**Note**

The source code of this example is not delivered with MeVisLab, as it would lead to module name collisions with the examples above. If you want to implement this example, make sure to change the module names or to move the original modules to another folder.

14.3.1. Copying the Source Files

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

14.3.2. Editing and Recompiling the `CMakeLists.txt` File

1. Open the `CMakeLists.txt` of your project in any text editor.
2. Add `mlSimpleAverage.h` and `mlSimpleAverage.cpp` to the `target_sources` statement.
3. Resulting code (excerpt):

```
target_sources(MLSimpleAddExample PRIVATE
    mlSimpleAddExample.cpp
    mlSimpleAddExample.h
    mlSimpleAverage.cpp
    mlSimpleAverage.h
    MLSimpleAddExampleInit.cpp
    MLSimpleAddExampleInit.h
    MLSimpleAddExampleSystem.h
)
```

4. Re-translate the `CMakeLists.txt` file.

14.3.3. Editing the Project in the Development Environment

Open the `SimpleAdd` project in your development environment.

14.3.3.1. Editing `SimpleAverage.h`

1. Open `SimpleAverage.h`.
2. Exchange the line

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

by

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

Resulting in:

```
// Local includes
#include "MLSimpleAddSystem.h"
```

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

```
MLSIMPLEAVERAGE_EXPORT
```

by

```
MLSIMPLEADD_EXPORT
```

Resulting in:

```
//! Computes the average voxel value of an image.
class MLSIMPLEADD_EXPORT SimpleAverage : public Module
```

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

14.3.3.2. Editing `MLSimpleAddInit.cpp`

1. Open `MLSimpleAddInit.cpp`.
2. Add the line

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

below the line

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

Resulting in:

```
// Include all module headers ...
#include "mlSimpleAdd.h"
#include "mlSimpleAverage.h"
```

3. Add the line

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

below the line

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

Resulting in:

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

This registers the classes to the ML runtime type system.

4. Recompile the project.



Note

mlSimpleAverage.cpp does not have to be edited.

14.3.4. Editing the Module Definition (.def)

1. Copy the file SimpleAverage.script to the folder containing the file SimpleAdd.def.
2. 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 (})

3. 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"
```

Resulting code:

```
MLModule SimpleAdd {
  DLL                = MLSimpleAdd
  genre              = ""
  author             = "JDoe"
  comment            = "Adds a constant double value to each voxel."
  keywords           = ""
  seeAlso            = Arithmetic1
  exampleNetwork     = $(LOCAL)/networks/SimpleAddExample.mlab
  externalDefinition = $(LOCAL)/SimpleAdd.script
}

MLModule SimpleAverage {
  DLL                = MLSimpleAdd
  genre              = ""
  author             = "JDoe"
  comment            = "Computes the average voxel value of an image."
  keywords           = "Statistics Average"
  seeAlso            = ""
  exampleNetwork     = $(LOCAL)/networks/SimpleAverageExample.mlab
  externalDefinition = $(LOCAL)/SimpleAverage.script
}
```

14.3.5. 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, for example via a quick search. However, you will find that in the About information, the same DLL will appear for both modules.

Chapter 15. Developing a Base Communication

In the following chapter, we will develop an ML module owning a Base object in combination with an Open Inventor module that will display the contents of the Base object.

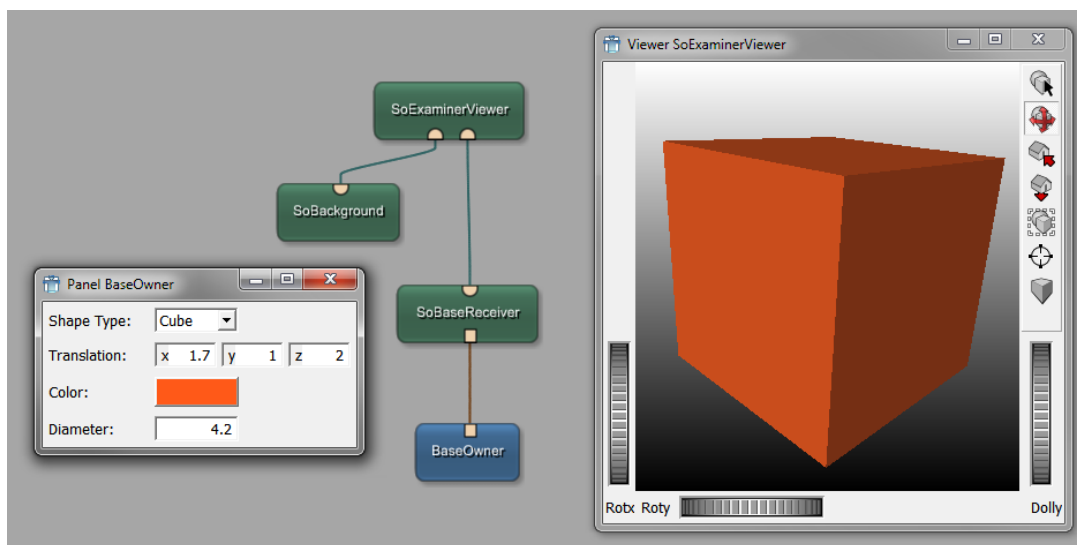
Purpose of this example:

- shows how to implement an ML module without any image processing functionality (no input/output image, hence no `calculateOutputSubImage` etc.).
- shows the use of an object derived from `Base` for communication between two modules.
- shows how to use an object derived from `Base` in an ML module and in an Open Inventor module.

The class `Base` is briefly referred to in the ML Guide, chapter “Base Objects”.

This will be our resulting network:

Figure 15.1. Example Network for ML Module and an Open Inventor Module



The data processing works as follows:

- The ML module offers fields for parameterizing a simple scene.
- The parameters are 'transported' by a Base object to another module.
- The receiving Open Inventor module renders a simple scene on base of the parameters set in the module with fields.

The example will be implemented with these elements:

- an `MLBaseOwner` module
- a `BaseMessenger` class
- a `SoBaseReceiver` module

Why this way of implementation?

- Separating the owner module from the receiver/visualization module is a lot more flexible than integrating the functionality — this way, it is possible to use several receivers/visualization modules with one messenger.
- The separation also means that the receiver does not have to know anything about the owner.
- Having an class derived from `Base` enables wrapping it as a scripting object so it can be used in Python directly. Have a look at the scripting example for wrapping the `BaseMessenger`.

The example is created in the following chapters:

- [Section 15.2, “Developing the `MLBaseOwner` Module and the `BaseMessenger` Class”](#)
- [Section 15.3, “Developing the `SoBaseReceiver` Module”](#)

15.1. A Note on Base Types Checks

15.1.1. Base Connectors and Field Types

When drawing a connection in MeVisLab, it is checked whether the basic connector type fits (`MLImage`, `Base`, `Open Inventor`). If not, a “blocked” sign appears.

As `Base` objects can be of different derived types, it is possible to connect `Base` fields with incompatible types, which might result in errors. To prevent this, the allowed `Base` object type(s) can be added to the `Base` field in the C++ source of the module. The allowed `Base` types are then displayed in the mouse-over information of a `Base` connector. This is especially helpful in cases where more than one `Base` connector is available.

Figure 15.2. Mouse-over Information for Base Connectors

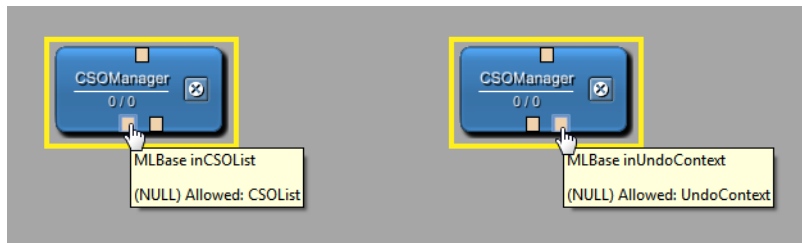
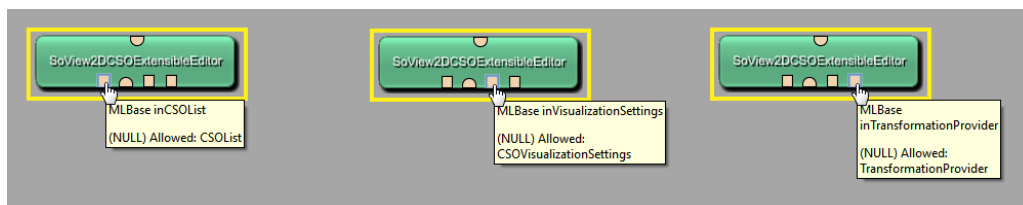


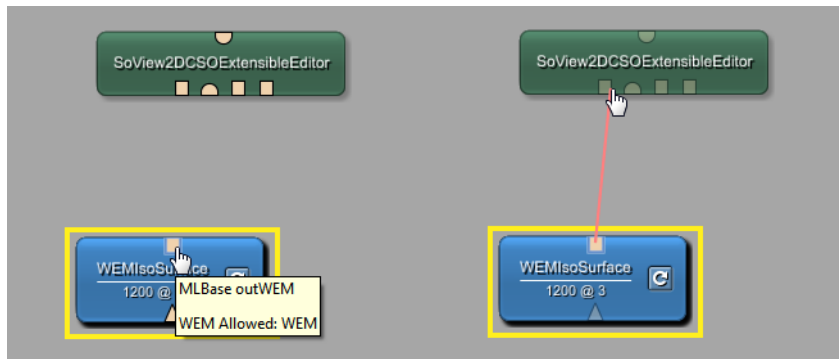
Figure 15.3. Mouse-over Information for Different Base Connectors in One Module



When connecting `Base` fields, the allowed types are checked and the connection is only possible for types in the allowed-list. This check also happens when connecting fields across macro modules, as the input/output fields of a macro “inherit” the allowed-list of the connected module fields.

While drawing a connection, the incompatible connectors are grayed out; if the connection is about to be dropped on an incompatible connector, the intermediate connection rendering is displayed in red.

Figure 15.4. Base Field Connection Checked for Type Compatibility



15.1.2. Overriding Base Type Checks

Sometimes it is useful to establish a connection although the Base field types are not compatible. For example, if the allowed types are set incorrectly in C++ when the module is still in development.

Three override methods are available:

- In scripting: Drawing the Base connection in scripting always works. However, scripting functions are available to check whether the types match: `allowedTypesByString()` and `matchesTypes(MLABMLBaseField *field)`, see the `MLABMLBaseField` Class Reference.
- In the `.mlab` file: Base connections can always be created by editing the `.mlab` file manually.

15.1.3. Implementing Base Type Checks

Implementing the Base type check is done in the C++ source.

To add an allowed type to a Base field, add the following C++ template method:

```
_myBaseField->addAllowedType<CSOList>();
```

There is a convenience template function available to set the initial value and the allowed type at the same time, especially if the initial value is not NULL:

```
_myBaseField->setBaseValueAndAddAllowedType(&_myOutputValue);
```

This derives the allowed type from the argument, but the type can also be defined specifically:

```
_myBaseField->setBaseValueAndAddAllowedType<lutFunction>(&_mySpecialOutputLUT);
```

It is possible to write

```
_myBaseField->setBaseValueAndAddAllowedType<lutFunction>(NULL);
```

but this is basically the same as

```
_myBaseField->addAllowedType<lutFunction>();
```

To add an allowed type to a Base field of an Inventor module, add the following C++ template method:

```
_myBaseField.addAllowedType<CSOList>();
```

The addition of checks for allowed Base types is demonstrated in the following example, see [Section 15.2.8.1, “Adding the construction of a new BaseMessenger Object”](#) and [Section 15.3.4, “Editing SoBaseReceiver.cpp”](#).

15.2. Developing the `MLBaseOwner` Module and the `BaseMessenger` Class

The ML module is of the class `Module` which is the base class from which all C++-based image processing modules are derived. Usually, it is used to implement new algorithms for processing voxel images. In our example, it will only offer the fields to parameterize the simple scene.

The class `Module` is explained in more detail in the ML Guide, chapter “Deriving Your Own Module from `Module`”.

Technically, this module owns the object derived from `Base` we will implement later:

- it constructs and deconstructs the `Base` object in its constructor and destructor, respectively.
- it holds the pointer to the `Base` object.
- it parameterizes the `Base` object according to the values of its fields and touches the `BaseField` in order to notify the receiving module(s) that the scene needs to be updated.

15.2.1. Creating the `BaseCommunication` Project in the Wizard

To create the `MLBaseOwner` module in the `BaseCommunication` project, use the wizard.

First of all, make sure that you have a user package defined as described in [Section 8.2, “Creating a User Package for Your Project”](#) or create it now. Then run the Project Wizard and select the option **ML Module**. This starts the Wizard for C++/ML Modules.

1. On the dialog **Module Properties**, enter the following:
 - **Name:** `BaseOwner`
 - **Comment:** Module for setting parameters of a `Base` object via fields
 - **Keyword:** Example
 - **See Also:** `SoBaseReceiver`
 - **Target Package:** your package, for example “Example/General”
 - **Project:** `BaseCommunication`

Figure 15.5. Project Wizard — Module Properties

Modules (C++)/ML Module

Module Properties

Enter the general properties of the module.

General Module Properties

Name: * BaseOwner Author: * JDoe

Comment: Module for setting parameters of a Base object via fields.

Keywords: Example

See Also: SoBaseReceiver

Genre: Choose ☐ Add reference to example network

Select Target Package

Package: * Example/General

Project Properties

Directory Structure: Classic

Project: * BaseCommunication Prefix: ML Select

☒ Include project files

* : Required fields

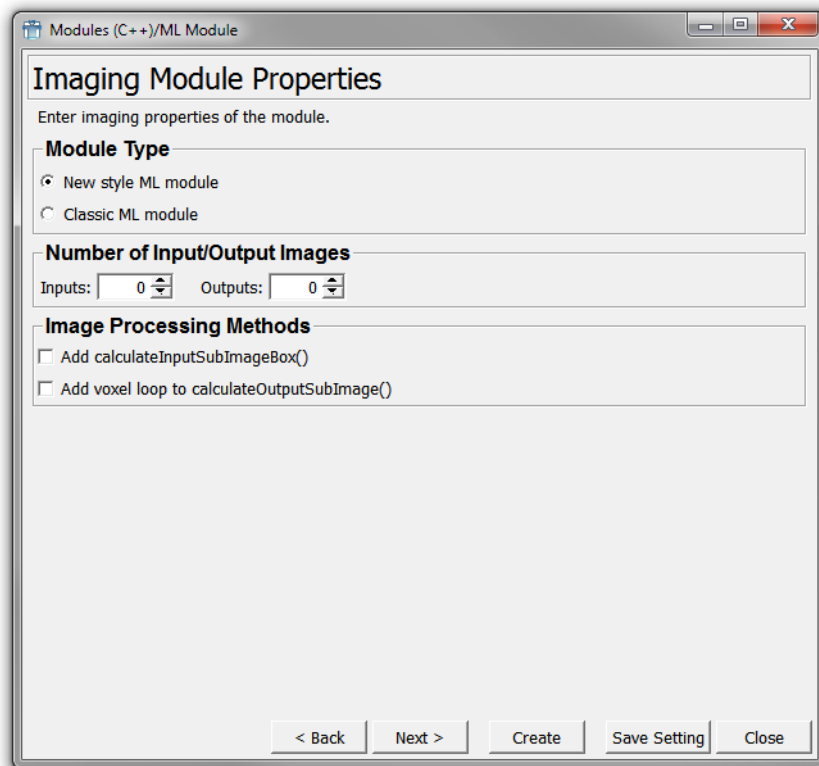
< Back Next > Create Save Setting Close

The project name is different to the module name here because the project will later include the module `MLBaseOwner` and the additional class `BaseMessenger`.

Click **Next** to proceed.

2. On the dialog **Imaging Module Properties**, all settings have to be removed as the module has no image input/outputs.
 - Keep **New Style ML Module**, as the setting is irrelevant for our example.
 - Change the settings to 0 inputs and 0 outputs.
 - Uncheck all options (**Add calculateInputSubImageBox** and **Add voxel loop to calculateOutputSubImage**).

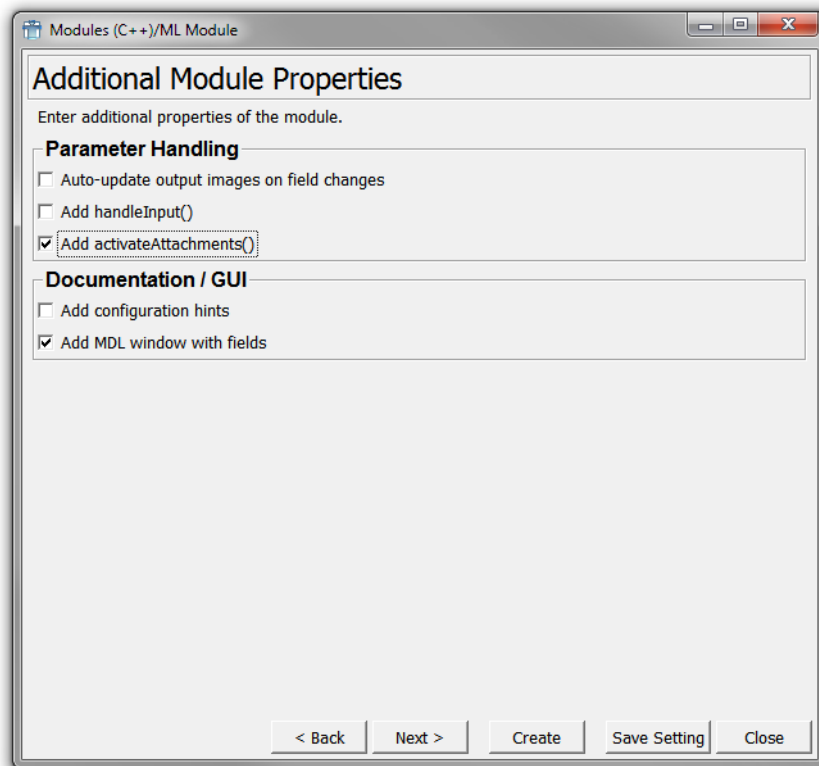
Figure 15.6. Project Wizard — Imaging Module Properties



Click **Next**.

3. On the dialog **Additional Module Properties**, the following settings are necessary:
 - Check **Add activateAttachments()**.
 - Check **Add MDL window with fields**.
 - Uncheck everything else.

Figure 15.7. Project Wizard — Additional Module Properties

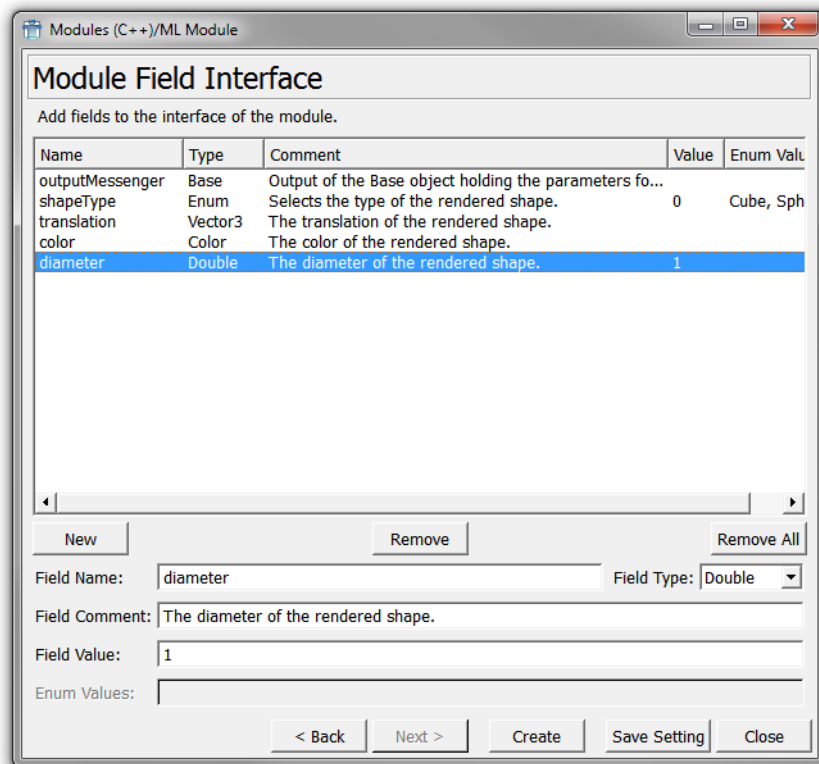


Click **Next**.

4. On the dialog **Module Field Interface**, add the following five fields (their sequence is not important):
 - Enter field:
 - **Field Name:** outputMessenger
 - **Field Type:** Base
 - **Field Comment:** Output of the Base object holding the parameters for the Inventor scene.
 - Enter field:
 - **Field Name:** shapeType
 - **Field Type:** Enum
 - **Field Comment:** Selects the type of the rendered shape.
 - **Field Value:** 0
 - **Enum Values:** Cube, Sphere
 - New field:
 - **Field Name:** translation
 - **Field Type:** Vector3
 - **Field Comment:** The translation of the rendered shape.

- New field:
 - **Field Name:** color
 - **Field Type:** Color
 - **Field Comment:** The color of the rendered shape.
- New field:
 - **Field Name:** diameter
 - **Field Type:** Double
 - **Field Comment:** The diameter of the rendered shape.
 - **Field Value:** 1

Figure 15.8. Project Wizard — Module Field Interface



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

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

- folder with the source code: {packagePath}\Sources\ML\MLBaseCommunication
- folder with the module's GUI definition: {packagePath}\Modules\ML\MLBaseCommunication



Note

For a full list of all created files and their contents, see the MeVisLab Reference Manual, chapter “ML Module (Wizard)”.

6. Close the Wizard.

The code resulting from the wizard is:

```
//-----
//! The ML module class BaseOwner.
//
// Module for setting parameters of a Base object via fields.
//-----

#include "mlBaseOwner.h"

ML_START_NAMESPACE

//! Implements code for the runtime type system of the ML
ML_MODULE_CLASS_SOURCE(BaseOwner, Module);

//-----

BaseOwner::BaseOwner() : Module(0, 0)
{
    // Suppress calls of handleNotification on field changes to
    // avoid side effects during initialization phase.
    handleNotificationOff();

    // Add fields to the module and set their values.
    _outputMessengerFld = addBase("outputMessenger", NULL);
    static const char * const shapeTypeValues[] = { "Cube", "Sphere" };
    _shapeTypeFld = addEnum("shapeType", shapeTypeValues, 2);
    _shapeTypeFld->setEnumValue(0);
    _translationFld = addVector3("translation", Vector3());
    _colorFld = addColor("color", 1,1,1);
    _diameterFld = addDouble("diameter", 1);

    // Reactivate calls of handleNotification on field changes.
    handleNotificationOn();
}

//-----

void BaseOwner::handleNotification(Field* field)
{
    // Handle changes of module parameters and input image fields here.
}

//-----

void BaseOwner::activateAttachments()
{
    // Update members to new field state here.
    // Call super class functionality to enable notification handling again.
    Module::activateAttachments();
}

ML_END_NAMESPACE
```

15.2.2. Adding New Files

Open your folder {packagePath}/Sources/ML/MLBaseCommunication and add two empty files:

- BaseMessenger.h
- BaseMessenger.cpp

These will be used for the `BaseMessenger` class that transmits the field values from the ML module to the Open Inventor module.

15.2.3. Adding References to the new Files in `CMakeLists.txt`

1. Open `CMakeLists.txt` of the project in a text editor.
2. Add references to the new files. Result:

```
target_sources(MLBaseCommunication PRIVATE
    BaseMessenger.cpp
    BaseMessenger.h
    MLBaseCommunicationInit.cpp
    MLBaseCommunicationInit.h
    MLBaseCommunicationSystem.h
    mlBaseOwner.cpp
    mlBaseOwner.h
)
```

3. Add the include paths for Base objects (`MLBase`) to the configuration:

```
find_package(MeVisLab COMPONENTS ML MLBase HINTS "$ENV{MLAB_ROOT}" REQUIRED)

target_link_libraries(MLBaseCommunication
    PUBLIC
    MeVisLab::ML
    MeVisLab::MLBase
)
```

4. Compile the `CMakeLists.txt` file and open the project in your development environment.

15.2.4. Adding Contents to `BaseMessenger.h`

Open `BaseMessenger.h` and enter the following code:

```
//-----
//! This class defines merely a parameter container for
//! visualization attributes and a shape enumeration.
//-----

#pragma once

#include <mlModuleIncludes.h>
#include <mlBase.h>
#include <mlLinearAlgebra.h>

// Local includes
#include "MLBaseCommunicationSystem.h"

ML_START_NAMESPACE

//-----

//! This enumeration lists all possible
//! shape types used in the owner and receiver modules.
enum MessengerShapeType
{
    ShapeTypeCube = 0,
    ShapeTypeSphere = 1
};
```

```
//-----

//! This class defines merely a parameter container for
//! visualization attributes and a shape enumeration.
class MLBASECOMMUNICATION_EXPORT BaseMessenger : public Base
{
public:

    //! Constructor.
    BaseMessenger();

    //! Copy constructor.
    BaseMessenger(const BaseMessenger& baseMessenger);

    //! Standard destructor.
    virtual ~BaseMessenger();

    //! \name Methods to retrieve attributes.
    //@{
    inline const Vector3&      getPosition() const { return _position; }
    inline const Vector3&      getColor()      const { return _color; }
    inline MLdouble           getDiameter()    const { return _diameter; }
    inline MessengerShapeType getShapeType()   const { return _shapeType; }
    //@}

    //! \name Methods to set attributes.
    //@{
    inline void setPosition(const Vector3& newPosition) { _position = newPosition; }
    inline void setColor(const Vector3& newColor)      { _color = newColor; }
    inline void setDiameter(MLdouble newDiameter)     { _diameter = newDiameter; }
    inline void setShapeType(MessengerShapeType newType) { _shapeType = newType; }
    //@}

private:

    //! \name Member variables.
    //@{
    Vector3 _position;
    Vector3 _color;
    MLdouble _diameter;
    MessengerShapeType _shapeType;
    //@}

    //! Implements interface for the runtime type system of the ML.
    ML_CLASS_HEADER(BaseMessenger)
};

//-----

ML_END_NAMESPACE
```

15.2.5. Add Contents to BaseMessenger.cpp

Open BaseMessenger.cpp and enter the following code:

```
// Local includes
#include "BaseMessenger.h"

ML_START_NAMESPACE

//! Implements code for the runtime type system of the ML
```

```
ML_MODULE_CLASS_SOURCE(BaseMessenger, Base);

//-----

BaseMessenger::BaseMessenger() : Base()
{
    // Set default values.

    _position.assign(0.0, 0.0, 0.0);
    _color.assign(1.0, 0.0, 0.0); // red
    _diameter = 1.0;
}

//-----

BaseMessenger::BaseMessenger(const BaseMessenger& baseMessenger) : Base()
{
    // Just copy values of the given object.

    _position = baseMessenger.getPosition();
    _color     = baseMessenger.getColor();
    _diameter = baseMessenger.getDiameter();
}

//-----

BaseMessenger::~BaseMessenger()
{
    // Not needed.
}

//-----

ML_END_NAMESPACE
```

15.2.6. Editing MLBaseCommunicationInit.cpp

Add the initialization of the `BaseMessenger` class (runtime type system).

1. Open `MLBaseCommunicationInit.cpp`.
2. Add the include of `BaseMessenger.h`. Result:

```
// Include all module headers
#include "mlBaseOwner.h"
#include "BaseMessenger.h"
```

3. Add the initialization of `BaseMessenger.h`. Result:

```
int MLBaseCommunicationInit ()
{
    // Add initClass calls from all other modules here...
    BaseOwner::initClass();
    BaseMessenger::initClass();

    return 1;
}
```

At this point, the project should be compilable.

15.2.7. Editing mlBaseOwner.h

Open `mlBaseOwner.h`.

1. Add the include of `BaseMessenger.h`. Result:

```
// Local includes
#include "MLBaseCommunicationSystem.h"
#include "BaseMessenger.h"
```

2. Add a destructor to the class:

```
//! Constructor.
BaseOwner();

//! Destructor.
~BaseOwner();
```

3. Add a private member variable of type `BaseMessenger` pointer since this module is the owner of the Base object. Result:

```
private:

    //! \name Member variables.
    //@{
    BaseMessenger* _baseMessenger;
    //@}
```

4. Add a private method to set the value in the Messenger object to the module's field values:

```
// Implements interface for the runtime type system of the ML.
ML_MODULE_CLASS_HEADER(BaseOwner)

//! Set the field values to the output messenger.
void _setFieldValuesToMessenger();
```

15.2.8. Editing `mlBaseOwner.cpp`

15.2.8.1. Adding the construction of a new `BaseMessenger` Object

Open `mlBaseOwner.cpp` and add the construction of a new `BaseMessenger` object and its parameterization to the constructor of the `BaseOwner` module. Use `setBaseValueAndAddAllowed` to ensure that the base type will be checked when drawing connections in the user interface.

Also, add the destruction of the `_baseMessenger` object to the destructor.

Result:

```
BaseOwner::BaseOwner () : Module(0, 0)
{
    // Suppress calls of handleNotification on field changes to
    // avoid side effects during initialization phase.
    handleNotificationOff();

    // Allocate memory for the BaseMessenger object.
    // Delete the object in this module's destructor.
    ML_CHECK_NEW(_baseMessenger, BaseMessenger());

    // Add fields for the interface.
    // Set the pointer to the BaseMessenger object to the output field.
    _outputMessengerFld = addBase("outputMessenger");

    _outputMessengerFld->setBaseValueAndAddAllowedType(_baseMessenger);

    static const char * const shapeTypeValues[] = { "Cube", "Sphere" };
    _shapeTypeFld = addEnum("shapeType", shapeTypeValues, 2);
```

```

_shapeTypeFld->setEnumValue(0);
_translationFld = addVector3("translation");
_translationFld->setVector3Value(Vector3());
_colorFld = addColor("color");
_colorFld->setColorValue(1,1,1);
_diameterFld = addDouble("diameter");
_diameterFld->setDoubleValue(1);

_setFieldValuesToMessenger();

// Reactivate calls of handleNotification on field changes.
handleNotificationOn();
}

// ...

BaseOwner::~BaseOwner()
{
    ML_DELETE(_baseMessenger);
}

```

15.2.8.2. Editing handleNotification

Change `handleNotification` so that it touches the output Base field after setting the module's field values to the `BaseMessenger` object. Result:

```

void BaseOwner::handleNotification(Field* field)
{
    // Handle changes of module parameters and input image fields here.
    bool touchOutputs = false;

    if ((field == _shapeTypeFld) ||
        (field == _translationFld) ||
        (field == _colorFld) ||
        (field == _diameterFld))
    {
        touchOutputs = true;
    }

    if (touchOutputs)
    {
        // Set the current parameter values to the messenger object
        // and touch the output field so the receiver generates its
        // scene anew.

        _setFieldValuesToMessenger();

        _outputMessengerFld->touch();
    }
}

```

15.2.8.3. Editing activateAttachments

After a saved network has been loaded and all the modules and their connection have been regenerated, the `activateAttachments` methods are called. We use this to regenerate the Base object with the saved parameters of the `BaseOwner` module.

Result:

```

void BaseOwner::activateAttachments()
{
    // Update members to new field state here.
    _setFieldValuesToMessenger();
}

```

```
_outputMessengerFld->touch();

// Call super class functionality to enable notification handling again.
Module::activateAttachments();
}
```

15.2.8.4. Implementing Setting the Parameters in BaseMessenger

Implement the setting of the parameters in the BaseMessenger according to the module's fields after the module has been loaded in a network (with restored field values) in activateAttachments(). Result:

```
void BaseOwner::activateAttachments ()
{
    // Update members to new field state here.
    _setFieldValuesToMessenger();

    // Call super class functionality to enable notification handling again.
    Module::activateAttachments ();
}
```

15.2.8.5. Implementing the method

`_setFieldValuesToMessenger()`

```
void BaseOwner::_setFieldValuesToMessenger()
{
    _baseMessenger->setPosition(_translationFld->getVector3Value());
    _baseMessenger->setColor(_colorFld->getVector3Value());
    _baseMessenger->setDiameter(_diameterFld->getDoubleValue());
    _baseMessenger->setShapeType(
        static_cast<MessengerShapeType>(
            _shapeTypeFld->getEnumValue()
        )
    );
}

ML_END_NAMESPACE
```

Save the file `mlBaseOwner.cpp`.

15.2.9. Making MLBaseCommunication classes known

Make the classes of the project MLBaseCommunication (and with it, the BaseMessenger) known to other projects:

1. Open the file `CMakeLists.txt` of the project in a text editor.
2. Change the last lines of the file, from:

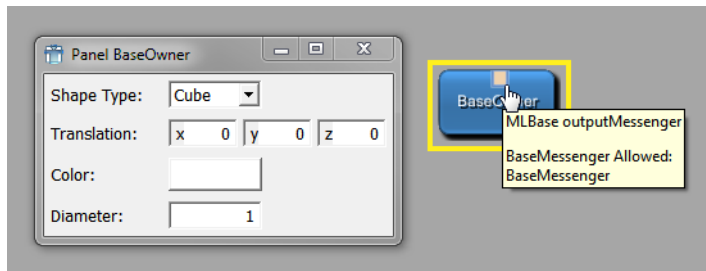
```
mlab_install(MLBaseCommunication NS MeVisLab)
```

to

```
mlab_install(MLBaseCommunication NS MeVisLab EXPORT)
mlab_install_headers(MLBaseCommunication)
```

Compile the project and restart MeVisLab. To check the final module, enter `BaseOwner` in the quick search and add it.

Figure 15.9. Resulting BaseOwner Module



Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Examples/Modules/GettingStarted/MLBaseCommunicationExample`, source files in `$(InstallDir)Packages/MeVisLab/Examples/Sources/GettingStarted/MLBaseCommunicationExample`). The module can be added via quick search.

15.2.10. Adding an object wrapper for MLBaseCommunication objects

To use the `MLBaseCommunication` in scripting, an object wrapper can be implemented. How this is done is explained here.

15.3. Developing the soBaseReceiver Module

In this section, we will develop the Open Inventor module that is necessary to display the output of `MLBaseOwner`.

Technically, this module receives the Base object and constructs a simple Open Inventor scene internally on base of the parameter and attribute values in the received Base object.

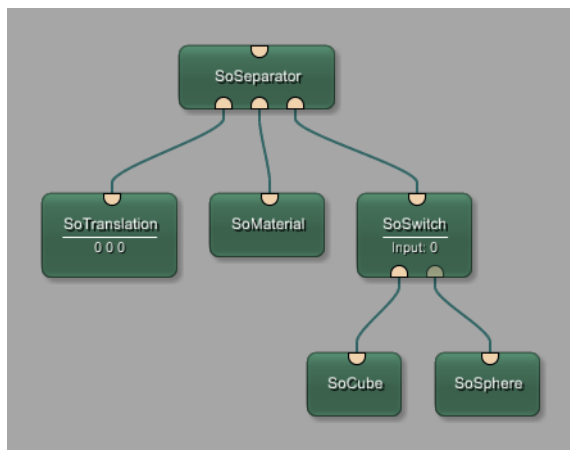


Tip

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

The internal scene graph of this module could also be built as a network in MeVisLab:

Figure 15.10. soBaseReceiver Module Alternative

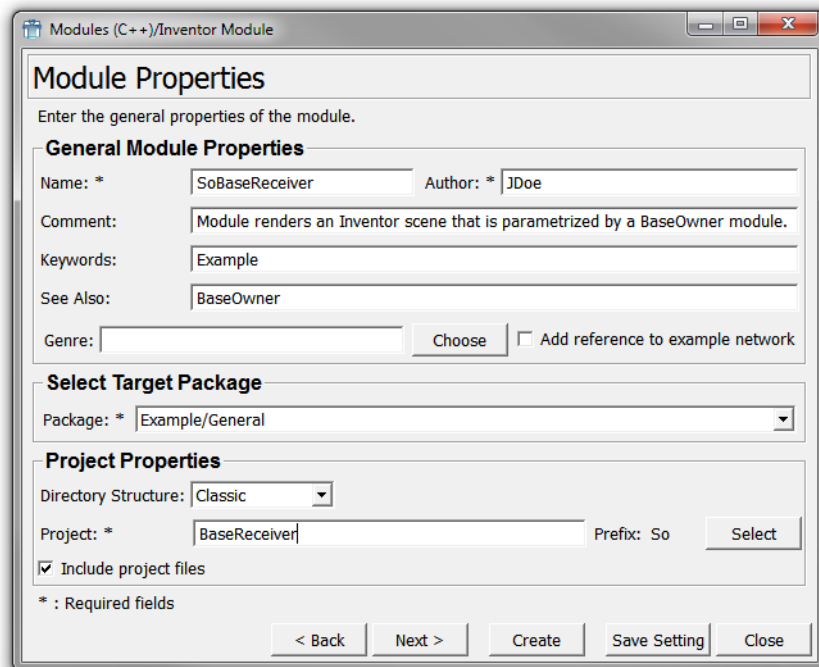


As you can see, `SoBaseReceiver` is essentially an Open Inventor separator module which has the advantage that it comes with its own viewer. The other modules deliver the translation, the color and the actual shape.

15.3.1. Creating the New Open Inventor Module with the Wizard

1. First of all, make sure that you have a user package defined as described in [Section 8.2, “Creating a User Package for Your Project”](#) or create it now.
2. Then run the Project Wizard and select the link **Inventor Module**. On the dialog **Module Properties**, enter the following:
 - **Name:** (So)BaseReceiver
 - **Comment:** Module renders an inventor scene that is parameterized by a BaseOwner module.
 - **Keyword:** Example
 - **See Also:** BaseOwner
 - **Target Package:** your package, for example “Example/General”
 - **Project:** BaseReceiver (“So” is added automatically)

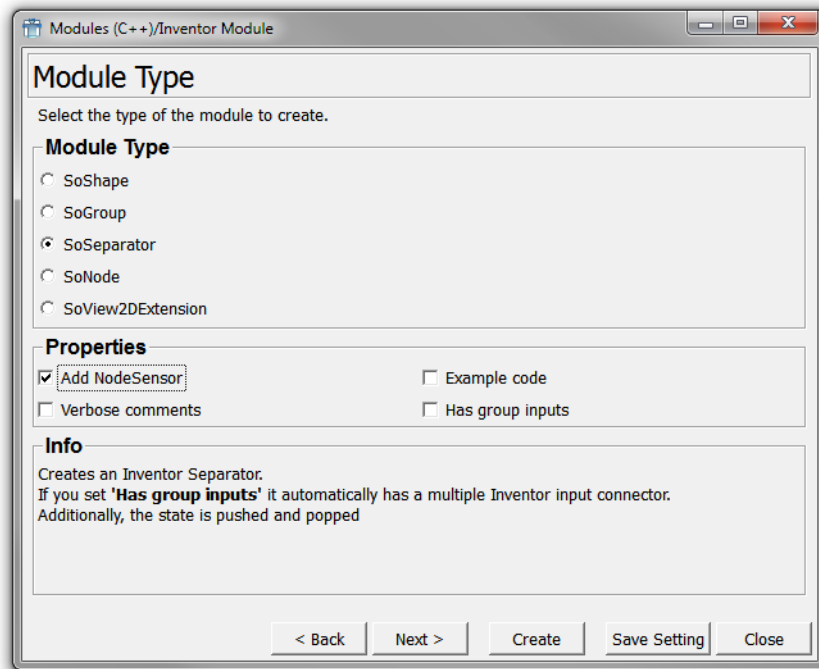
Figure 15.11. Project Wizard — General Module Properties



Click **Next** to proceed.

3. On the dialog **Module Type**, select **SoSeparator** and check the option **Add Node Sensor**.

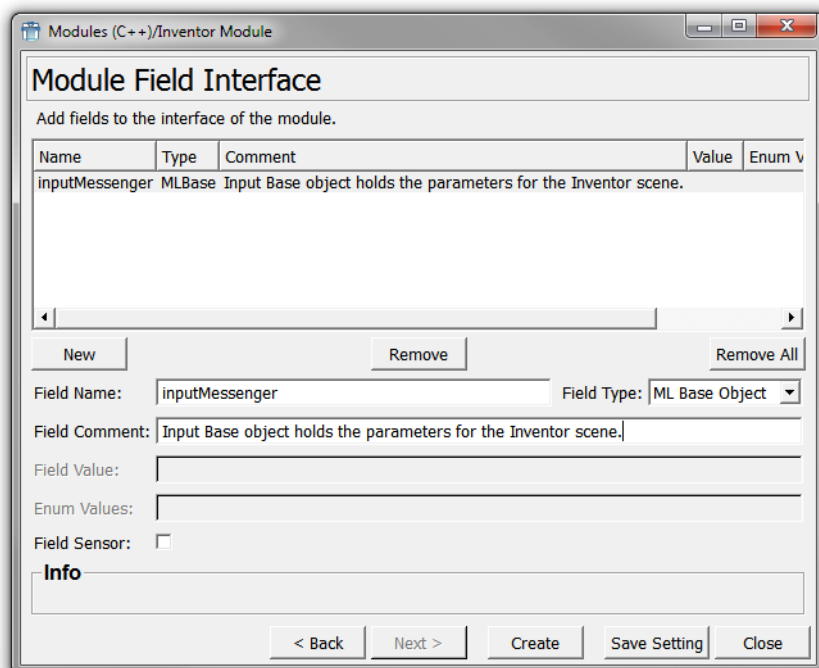
Figure 15.12. Project Wizard — Module Type



4. On the dialog **Module Field Interface**, enter one field:

- **Field Name:** inputMessenger
- **Field Type:** ML Base Object
- **Field Comment:** Input Base object holds the parameters for the inventor scene.

Figure 15.13. Project Wizard — Module Field Interface





Tip

Why using a node sensor instead of a field sensor? In our example, it would make no difference as we only have one input field. Usually, however, there will be more than one field, and as each field sensor will add redundant code to the module, using a node sensor that will react to any changes of the Open Inventor node is usually recommended.

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

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

- folder with the source code: {packagePath}\Sources\So\SoBaseReceiver
- folder with the module's .def file definition: {packagePath}\Modules\So\SoBaseReceiver.



Note

For a full list of all created files and their contents, see MeVisLab Reference Manual, chapter "ML Module (Wizard)".

6. Close the Wizard.

The code resulting from the wizard is:

```
//-----
//! The Inventor module class SoBaseReceiver
//
// Module renders an Inventor scene that is parametrized by a BaseOwner module.
//-----

#include "SoBaseReceiver.h"

#include <Inventor/elements/SoCacheElement.h>

SO_NODE_SOURCE(SoBaseReceiver)

void SoBaseReceiver::initClass()
{
    SO_NODE_INIT_CLASS(SoBaseReceiver, SoSeparator, "Separator");
}

SoBaseReceiver::SoBaseReceiver()
{
    // Execute Inventor internal code for node construction.
    SO_NODE_CONSTRUCTOR(SoBaseReceiver);

    SO_NODE_ADD_FIELD(inputMessenger, (NULL));
    // Create a sensor calling _nodeChangedCB if any field changes. Use a priority 0
    // sensor to be sure that changes are not delayed or collected.
    _nodeSensor = new SoNodeSensor(SoBaseReceiver::nodeChangedCB, this);
    _nodeSensor->setPriority(0);
    _nodeSensor->attach(this);
}

SoBaseReceiver::~SoBaseReceiver()
{
    // Remove the node sensor.
    delete _nodeSensor;
}
```

```

}

void SoBaseReceiver::nodeChangedCB(void* data, SoSensor* sensor)
{
    static_cast<SoBaseReceiver*>(data)->nodeChanged(
        static_cast<SoNodeSensor*>(sensor)
    );
}

void SoBaseReceiver::nodeChanged(SoNodeSensor* sensor)
{
    // Get the field which caused the notification.
    SoField* field = sensor->getTriggerField();
    // Handle changed fields here
}

```

As the module is already of type SoSeparator, no additional include has to be made for that.

15.3.2. Editing CMakeLists.txt of SoBaseReceiver

1. Open the CMakeLists.txt of the SoBaseReceiver project in a text editor.
2. Add the inclusion of the MLBaseCommunication project to the find_package and target_link_libraries calls. Result:

```

find_package(MeVisLab COMPONENTS ML MLABBase OpenGL InventorBinding MLBaseCommunication HINTS

target_link_libraries(SoBaseReceiver
    PUBLIC
        MeVisLab::MLBaseCommunication

        MeVisLab::ML
        MeVisLab::MLBase
        MeVisLab::OpenGL
        MeVisLab::InventorBinding
        OpenInventor::OpenInventor
)

```

3. Create a project file for your development environment out of the CMakeLists.txt file.

15.3.3. Edit SoBaseReceiver.h

1. Open SoBaseReceiver.h.
2. Add a forward declaration (in a doxygen comment group) between the includes and the class declaration. Forward declarations are used here because in the header file, it is not necessary to know the actual classes because only pointer are declared here. The definition of the classes is used in the .cpp file where the according header files of the used classes must be included.

```

#include "mlAPI.h"

//! \name Forward declarations
//@{
class SoMaterial;
class SoTranslation;
class SoSwitch;
class SoSphere;
class SoCube;
//@}

```

3. Add private member variables to reference parts of the internal scene graph:

```
private:

    //! \name Member variables
    //@{
    //! The node providing the color properties to the output scene.
    SoMaterial* _material;
    //! The node providing the translation of the output scene.
    SoTranslation* _translation;
    //! A node to switch between the shapes 'cube' and 'sphere' as
    //! well as to turn off any output shape.
    SoSwitch* _shapeSwitch;
    //! The output shape: cube.
    SoCube* _cube;
    //! The output shape: sphere.
    SoSphere* _sphere;
    //@}
```

4. Add a private method to set the received parameters to the output scene graph:

```
    //! Parameterizes the internal scene graph.
    void _parameterizeSceneGraph();
};
```

15.3.4. Editing SoBaseReceiver.cpp

1. Open SoBaseReceiver.cpp.
2. Add includes. Result:

```
#include <Inventor/elements/SoCacheElement.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoTranslation.h>
#include <Inventor/nodes/SoSwitch.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoCube.h>

#include <BaseMessenger.h>
```

3. Change the constructor to generate the scene graph here. Set the allowed Base type to the input Base field.

Result:

```
// -----
//! Constructor, creates fields and scene graph
// -----
SoBaseReceiver::SoBaseReceiver()
{
    // Execute inventor internal stuff for node construction.
    SO_NODE_CONSTRUCTOR(SoBaseReceiver);
    SO_NODE_ADD_FIELD(inputMessenger, (NULL));
    inputMessenger.addAllowedType<ml::BaseMessenger>();
    // Create scene graph

    // Add nodes that influence the whole scene
    // independent on the actual shape
    _translation = new SoTranslation();
    addChild(_translation);

    _material = new SoMaterial();
    addChild(_material);

    // Create subgraph to switch the shapes
```

```

_shapeSwitch = new SoSwitch();
addChild(_shapeSwitch);

_cube = new SoCube();
_shapeSwitch->addChild(_cube);

_sphere = new SoSphere();
_shapeSwitch->addChild(_sphere);

// Create a sensor calling _nodeChangedCB if any field changes.
// Use a priority 0 sensor to be sure that changes are not
// delayed or collected.
_nodeSensor = new SoNodeSensor(SoBaseReceiver::nodeChangedCB, this);
_nodeSensor->setPriority(0);
_nodeSensor->attach(this);

// Update the parameters of the internal scene graph
// according to the connected BaseMessenger
_parameterizeSceneGraph();
}

```

4. Call the updating of the internal scene graph if the input field has changed. Result:

```

//-----
//! Called on any change on the node, field might by also NULL
//-----
void SoBaseReceiver::nodeChanged(SoNodeSensor* sensor)
{
    // Get the field which caused the notification.
    SoField* field = sensor->getTriggerField();

    // Handle changed fields here
    if (field == &inputMessenger)
    {
        _parameterizeSceneGraph();
    }
}

```

5. Implement the method that sets the parameters of the output scene according to the BaseMessenger's parameters. Result:

```

void SoBaseReceiver::_parameterizeSceneGraph()
{
    // check if the BaseMessenger is valid
    ml::BaseMessenger* baseMessenger =
        mlbase_cast<ml::BaseMessenger*>(inputMessenger.getValue());

    if (baseMessenger)
    {
        // set parameters for all shapes
        ml::Vector3 position = baseMessenger->getPosition();
        _translation->translation.setValue(position[0], position[1], position[2]);

        ml::Vector3 color = baseMessenger->getColor();
        _material->diffuseColor.setValue(SbVec3f(color[0], color[1], color[2]));

        const double diameter = baseMessenger->getDiameter();

        _cube->width = diameter;
        _cube->height = diameter;
        _cube->depth = diameter;

        _sphere->radius = diameter * 0.5;

        switch (baseMessenger->getShapeType())

```

```
{
  case ml::ShapeTypeCube:
    _shapeSwitch->whichChild.setValue(0);
    break;
  case ml::ShapeTypeSphere:
    _shapeSwitch->whichChild.setValue(1);
    break;
  default:
    _shapeSwitch->whichChild.setValue(-1);
    break;
}
}
else
{
  // no output scene
  _shapeSwitch->whichChild.setValue(-1);
}
}
```

The project should compile now, and both modules can be used in a network. The `BaseOwner` can parameterize a shape and the `SoBaseReceiver` renders a shape with that parameterization.



Tip

This example is delivered with MeVisLab (.def file in `$(InstallDir)Packages/MeVisLab/Examples/Modules/GettingStarted/SoBaseReceiverExample`, **source files** in `$(InstallDir)Packages/MeVisLab/Examples/Sources/GettingStarted/SoBaseReceiverExample`). The module can be added via quick search.

Chapter 16. Using the TestCenter

In the following chapter, we will introduce you to using the MeVisLab TestCenter.

- [Section 16.1, “Introduction to Testing in MeVisLab”](#)
- [Section 16.2, “Developing a Test Case”](#)

16.1. Introduction to Testing in MeVisLab



Note

In the following section, we only have a brief look at the concepts of the TestCenter. For detailed information and references, see the TestCenter Reference and the TestCenter Manual.

The testing of macro modules, networks, applications and scriptable functionality in MeVisLab is done with the TestCenter.



Tip

On the C++ level, GoogleTest can be used.

What makes a test? Possible definitions:

- A test compares results against expectations.
- A test uses a parameterized algorithm to generate data that is compared to expected results.
- A test is a specification that can easily be verified.

Two main categories of test cases can be created with the TestCenter:

- Generic test cases: Tests a larger set of modules by applying a test case generically.
- Functional test cases: Tests specific functionalities of a single module or network.

A test case in MeVisLab consists of

- a set of test functions
- input data
- (optional) a network



Tip

A network provides the context for a module to be tested, such as inputs/outputs/other modules for comparison. A network might be unnecessary for testing scripting functionality only, but might still be useful if you want to add a module, connect it to another module, and then remove again, etc.

TestCases are similar to macro modules, with two differences:

- They are not handled by the general module database but by a specific test case database, the `MLABTestCaseDatabase`.
- The TestCase database must be initialized explicitly.

TestCases are located in the TestCases directories of the packages, parallel to the folders “Modules” and “Sources”.

The test case creation and management is supported by the special macro module (`TestCaseManager`) which is implemented in the MeVisLab GUI and will be used in our example.



Note

Test cases cannot be deleted in the `TestCaseManager`. To delete a test case, delete the folder of the test case on your system and click **Reload All** in the `TestCaseManager`.

The name of a test function consists of three parts:

- One of the following predefined keywords to define the test type:
 - **TEST**: a function that is executed once.
 - **FIELDVALUETEST**: a test based on interactively predefined settings of fields and comparison of computed field values with expected results.
 - **ITERATIVETEST**: a test based on a list of given parameter and a function that is executed for each parameter.
- An arbitrary string used for sorting.
- An arbitrary name of the function for display purposes.

16.2. Developing a Test Case

In this section, we will develop a test case for the `Threshold` module. The `Threshold` module transforms the input image to a binary image with:

- voxel values below the threshold being set to the minimum image value.
- voxel values at or above the threshold being set to the maximum image value.

The `TestPattern` module will be used for the input image.

The `ImageStatistics` module will be used for verifying the test results.

16.2.1. Creating a New Test Case

1. Open the `TestCaseManager` via the menu bar, **File** → **Run Test Case Manager**.
2. Select the **Test Creation** tab.
3. Enter the following:
 - **Name**: `MyThresholdTest`
 - **Type**: the type of your module, this selects a sub-directory in the test directory.
 - **Package**: your package, for example “Example/General” (this is the example user package created in [Section 8.2, “Creating a User Package for Your Project”](#))
 - **Comment**: Tests the `Threshold` module.



Note

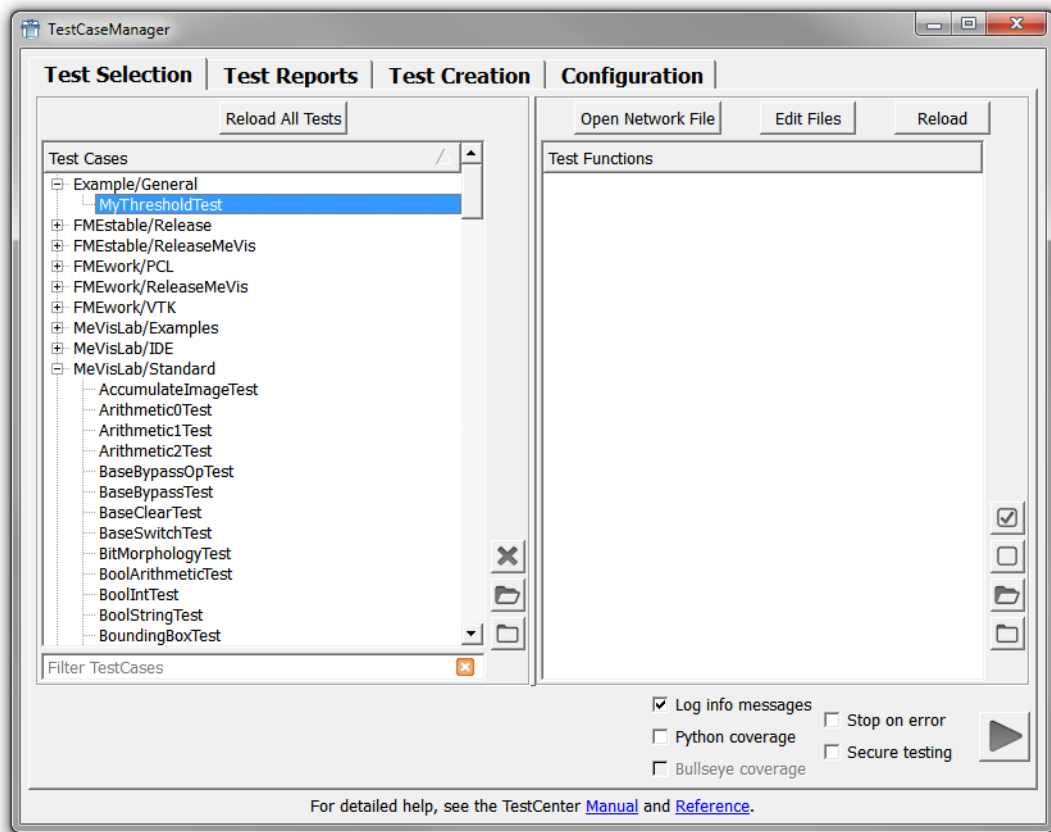
As we will build the network in the next step, start with an empty network here. If you already had a network that could be used as test case, you could import it here.

Figure 16.1. Creating a New Test Case

The screenshot shows the 'TestCaseManager' application window with the 'Test Creation' tab selected. The window has a title bar with standard OS controls. Below the title bar is a tabbed interface with four tabs: 'Test Selection', 'Test Reports', 'Test Creation' (active), and 'Configuration'. The 'Test Creation' tab is divided into two sections: 'General' and 'Test Network'. The 'General' section contains fields for 'Name' (MyThresholdTest), 'Type' (ML), 'Package' (Example/General), 'Directory Structure' (Classic), 'Project' (empty), and 'Comment' (Tests the Threshold module.). There are also buttons for 'Auto Fill Fields From Module...' and 'Reset Fields'. The 'Test Network' section has a 'Test Network' dropdown (Empty Network) and a 'Network' field with a 'Browse...' button. At the bottom right of the 'General' section is a 'Create' button. At the very bottom of the window, there is a footer text: 'For detailed help, see the TestCenter [Manual](#) and [Reference](#).'

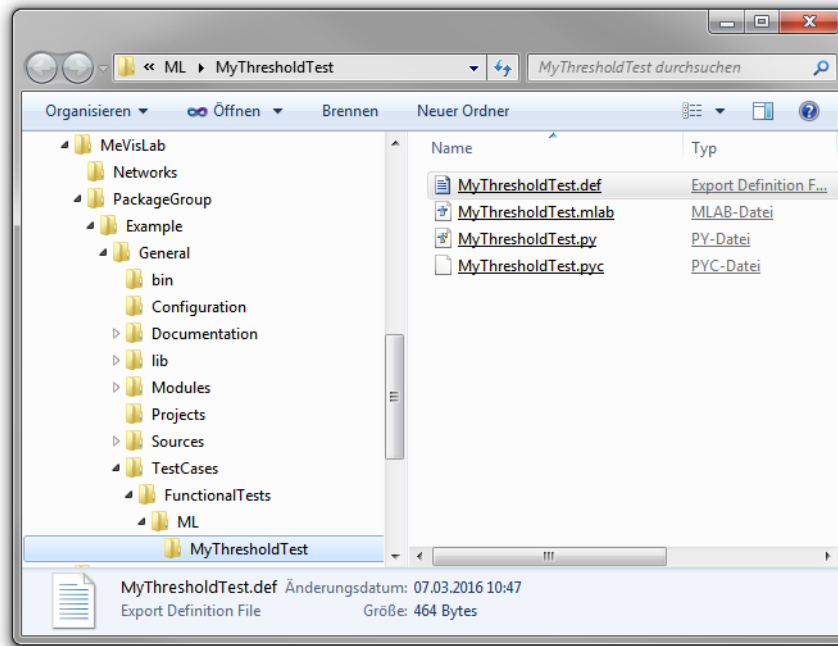
4. Click **Create** to create the test case.

The test case is created and the **Test Selection** tab is opened, where you can now find the new test case.

Figure 16.2. New Test Case in Test Selection

In your package path, a new folder `Testcases/FunctionalTests/MyThresholdTest` is created. `MyThresholdTest` contains the necessary files for the test case:

- `MyThresholdTest.def`: for the test case definition, similar to the `MeVisLab` module definition files. Contains the keyword "FunctionalTestCase", a timeout parameter and the reference to the script file, in this case `MyThresholdTest.py`.
- `MyThresholdTest.mlab`: the example network, empty so far
- `MyThresholdTest.py`: the Python scripting for the test case

Figure 16.3. New Test Case in the Package Path

16.2.2. Populating the Test Network

Our test case is associated with a test network, so in the next step, we need to add the necessary modules to the so far empty network.

1. In the TestCaseManager, select the new test case and click **Open Network File**. The empty network opens in MeVisLab.
2. Add the three required modules:
 - Threshold
 - TestPattern
 - ImageStatistics
3. Connect the modules as can be seen in [Figure 16.4, “Basic Test Case Setup”](#).
4. Save the network.

16.2.3. Editing the Module Settings

For the test case, a setup is necessary with which the function of the `Threshold` can be tested. This can easily be done when the voxel values in the image correspond to the position on the x-axis that is determined by the threshold value `n`.

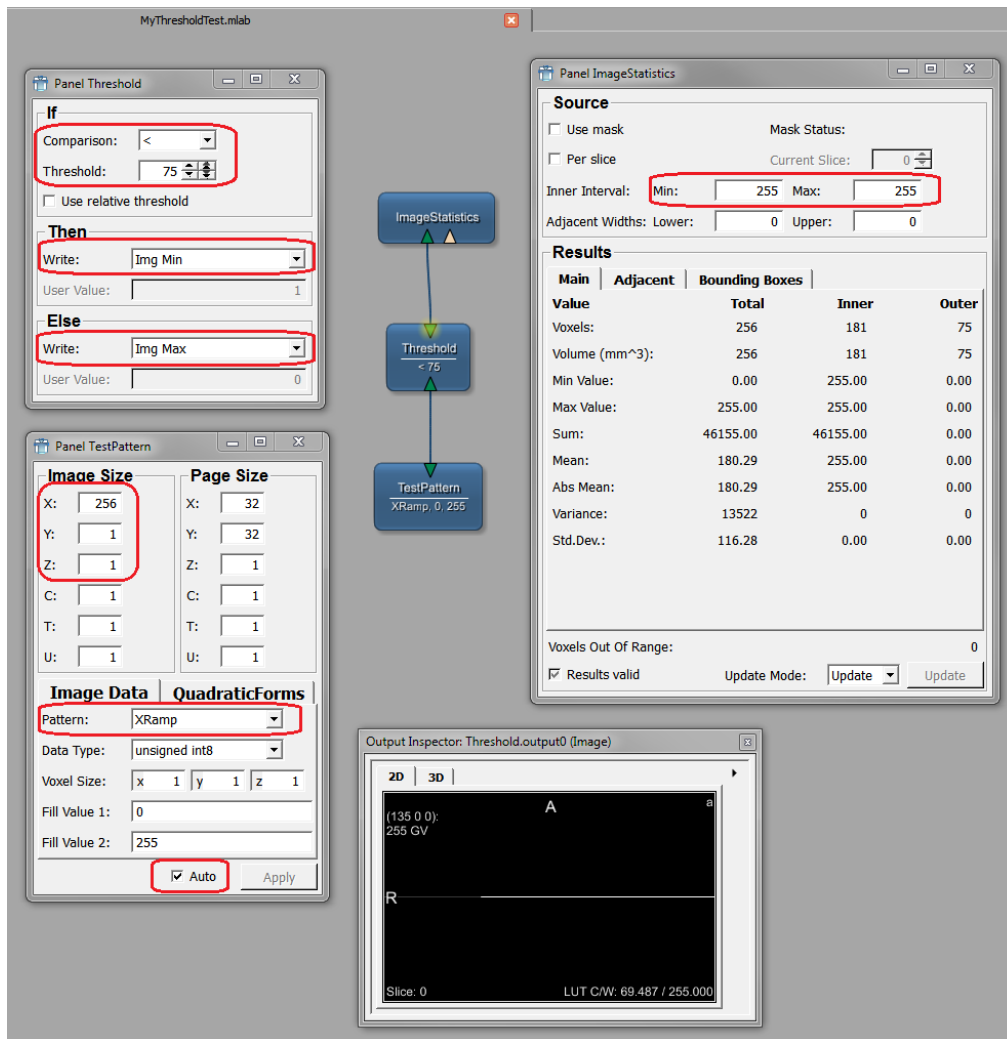
1. Modify the `TestPattern` parameters:
 - **ImageSize:** X = 256 and Y = Z = 1. This draws a horizontal line.
 - **Pattern:** XRamp. This creates a gradient from voxel value 0 to 255.
 - **Auto:** check this option to generate an output image automatically.
2. Modify the `Threshold` parameters:
 - **Comparison:** set this to < (less than).
 - **Then - Write:** set this to `ImgMin`.
 - **Else - Write:** set this to `ImgMax`.
3. Modify the `ImageStatistics` parameters, so that the **Inner Interval** is Min = Max = 255. This way, all voxels with the value = 255 will count as inner voxels. (Min and Max could also be set to 0; in this

case the voxels with value = 0 would count as inner voxels — the decision between inner and outer here is arbitrary and irrelevant as long as the correct fields are compared later.)

4. Save the network again.

With this setup, the voxel values in the created image are equal to the position on the x-axis. Voxels below the threshold are set to value = 0, voxels above the threshold are set to value = 255. For example, for a threshold of 75, 75 voxels are set to 0 (counting as outer Voxels) and 181 voxels above the threshold are set to 255 (counting as inner voxels), as can be seen in the results on the `ImageStatistics` panel.

Figure 16.4. Basic Test Case Setup



16.2.4. Creating a First Test Script with Manual Threshold Setting

In the next step, the actual test script needs to be programmed. In our case, the threshold needs to be set and the results have to be verified.

1. In the TestCaseManager, select the test case and click **Edit Files**. The files open in the integrated text editor MATE. The generated file `MyThresholdTest.py` looks like this:

```
#from TestSupport import Fields
#from TestSupport.Macros import *

#def TEST_exampleFunction():
```

```
#      """ Testing foobar """
#      Fields.setValue("MyModule.foo", 1)
#      EXPECT_EQ(2, Fields.getValue("MyModule.bar"))
```

2. Remove the comment symbol # from the lines.
3. Rename `exampleFunction` to something recognizable, for example "TEST_ManualTest_75". The tests are executed in alphabetic order. If you need to execute them in a certain order, you can add numbers to the tests, e.g. TEST001_, TEST002, etc. Note that this is generally considered bad practice - test cases should be independent of each other.
4. Add the actual function. Three actions are needed:
 - a. The threshold has to be set to a value, for example "75".
 - b. The `ImageStatistics` module has to be updated.
 - c. It has to be verified that the value for the outer voxels corresponds to the entered threshold value.

This is done with the following Python code:

```
Fields.setValue("Threshold.threshold", 75)
EXPECT_EQ(Fields.getValue("ImageStatistics.outerVoxels"), 75)
```



Tip

The function `EXPECT_EQ` checks whether two given values are equal. It is a Python function modeled after the macro of the same name in the GoogleTest library. For quick help, right-click the name in MATE and select **Show Help for 'EXPECT_EQ'**. Further information on the TestCenter macros and functions can be found in the TestCenter Reference.

5. Save the script. The resulting code for this manual (static) test is:

```
from TestSupport import Fields
from TestSupport.Macros import *

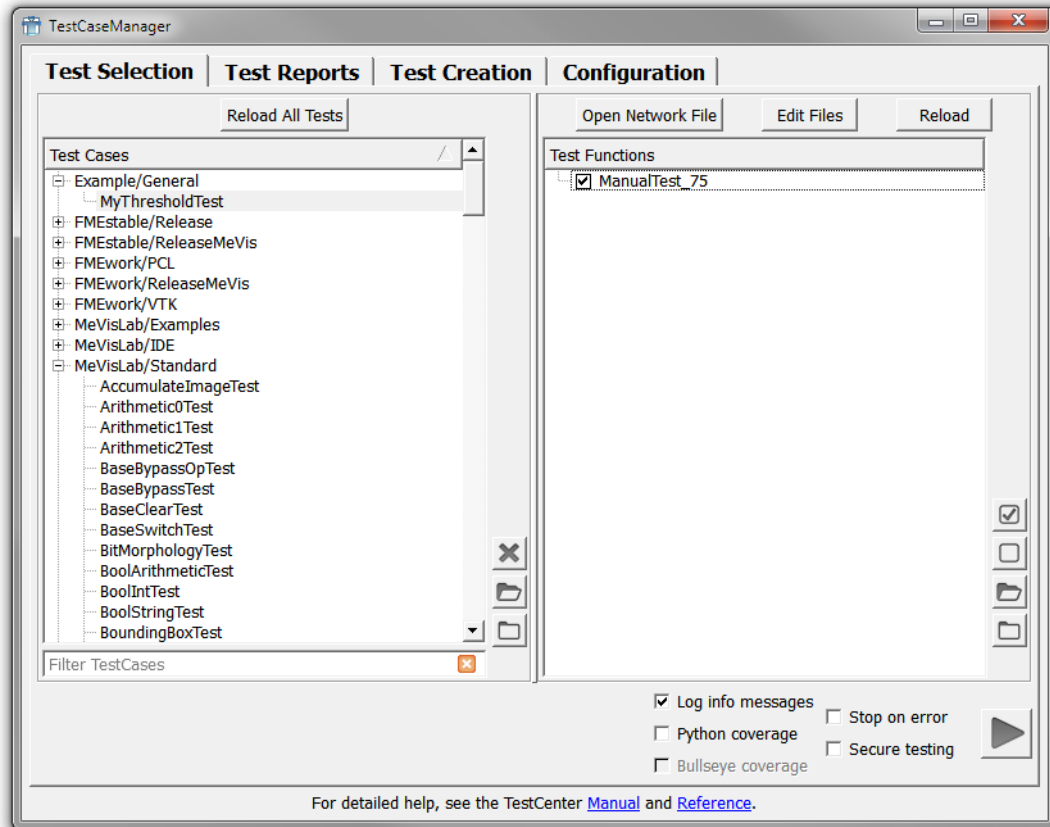
def TEST_ManualTest_75 ():
    """ -- Basic test for threshold values -- """
    Fields.setValue("Threshold.threshold", 75)
    EXPECT_EQ(Fields.getValue("ImageStatistics.outerVoxels"), 75)
```



Note

`TestSupport.Macros` provides a number of functions of the form `EXPECT_xxx` and `ASSERT_xxx`. Importing them using wildcard import is a convenient way to provide auto completion for all these functions. To avoid complains from your linter, you may want to import only the needed functions instead.

6. In the TestCaseManager, select the test case and click **Reload** to reload the test case. The new test function will be listed on the right.

Figure 16.5. Test Functions in the TestCaseManager**Tip**

When hovering over the test function with the mouse, the function's comment is displayed as a tool tip.

7. Finally, click on **Run** to run the test function. The option **Secure Testing** defines that the test case is run in another instance of MeVisLab; you might want to keep it checked.

The report should look as follows:

Figure 16.6. Report for ManualTest_75**Tip**

For defining the test functions status, the MeVisLab debug console is used (OK, Error, Warning), see also “ExampleTestCase1” in the test cases for the `MeVisLab/Standard` package.

Excursion: About Context and Fields

When using the **Scripting Assistant** (see MeVisLab Reference Manual, chapter “Scripting Assistant”), the following scripting line would be offered when setting the threshold value: `ctx.field("Threshold.threshold").value = 75`. The context “ctx” is the context from which the scripting is called up. When called up in an ML module, the context would be the ML module. If called up in a macro module, the context would be the macro module. The context also defines which context-sensitive help link is offered in the integrated text editor MATE.

For testing, however, using “ctx.field” is not the sensible approach because this way, the value for the field is directly set and will remain as set even after the closure of the test function and the start of the next test function. This might result in undefined conditions of the test case. The better solution here is to set the value with `Fields.setValue("Threshold.threshold", 75)`. This sets the value only for the currently running function and then sets it back to the saved value the field had before calling the function.

16.2.5. Automating the Test Case with the FieldValueTestCaseEditor

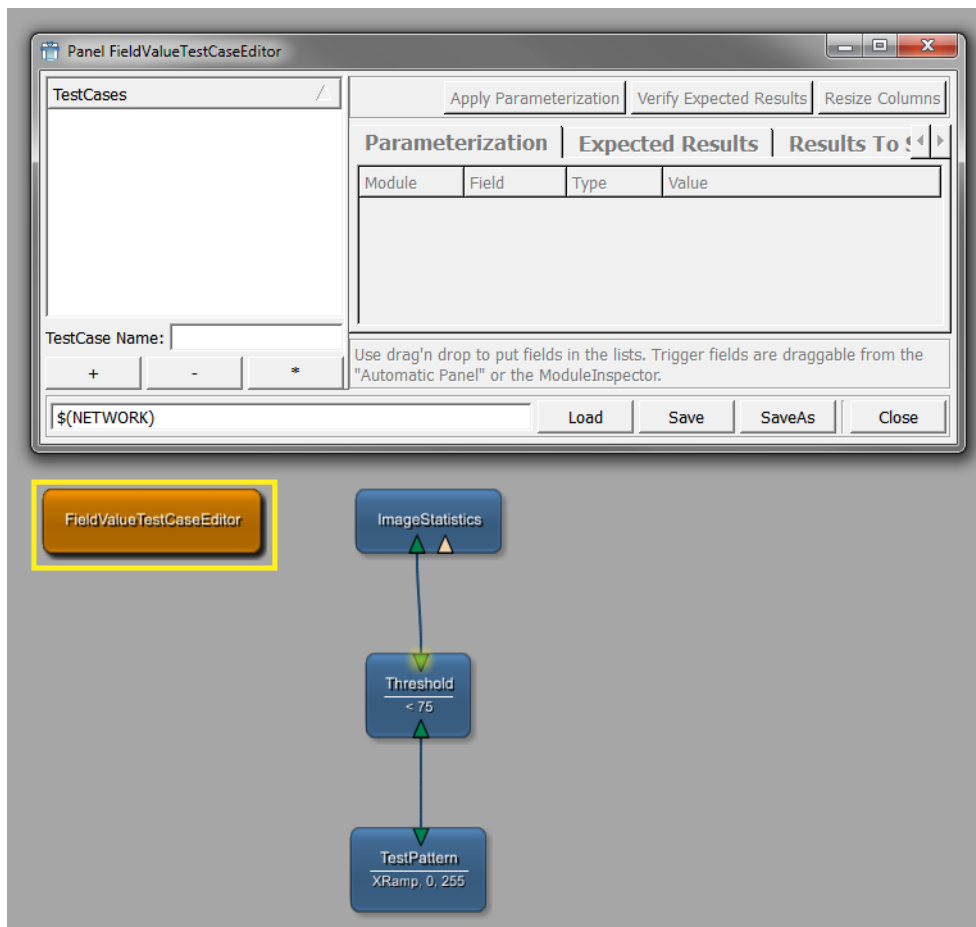
One possibility to automate our example test is to use the `FieldValueTestCaseEditor` module. With it, field-value test cases can be created.

**Tip**

Aside of the module described in the following chapter, other modules are available to handle field-value test cases, for example `FieldValueTestCaseGenerator` for the fully automated generation of test cases based on parameters and their permutations. Use the Quick Search to find more `FieldValueTestCase` modules.

Add the module `FieldValueTestCaseEditor` to your test network and save the network.

Figure 16.7. The `FieldValueTestCaseEditor` Panel



The user interface is split into three main parts:

- The FieldValue (FV) **Test Cases** list is on the left. There are three buttons to add (+), remove (-) and duplicate (*) test cases.
- The FV test case editing is done on the right. Here, test cases can be (re)named and parameterized.
- The listed FV test cases are saved as one set in an XML file, which is handled on the bottom of the window.

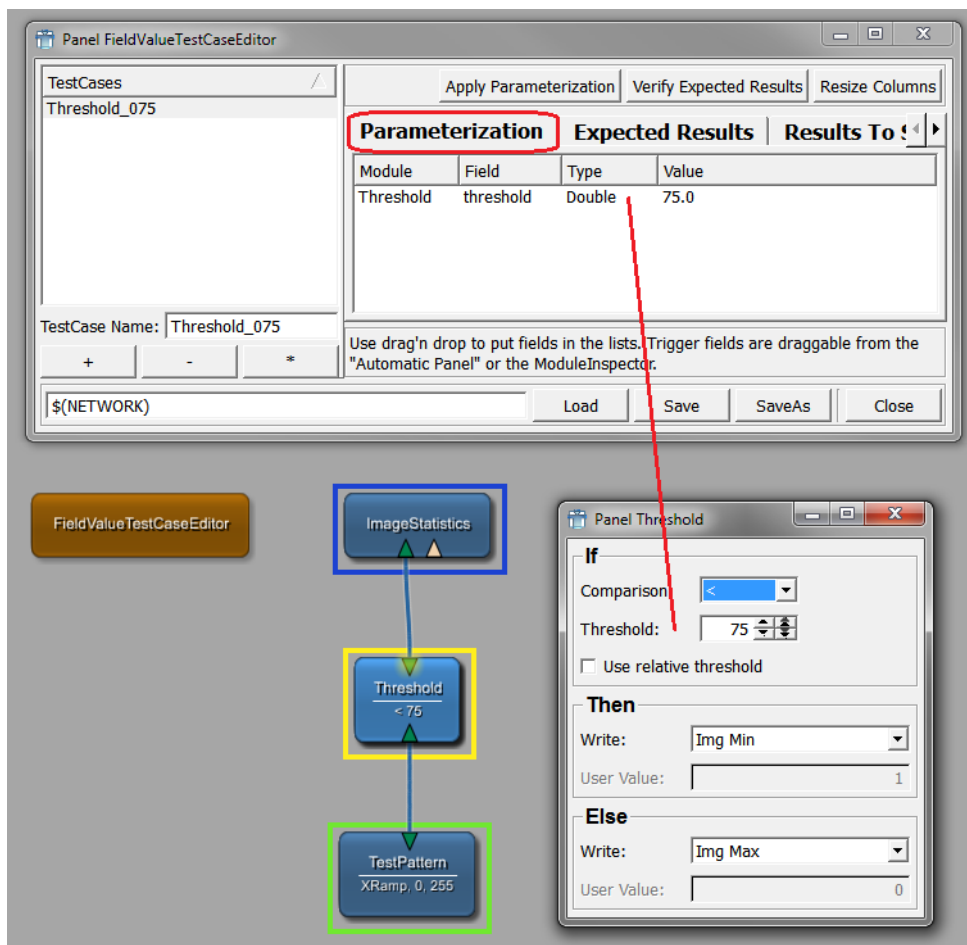


Note

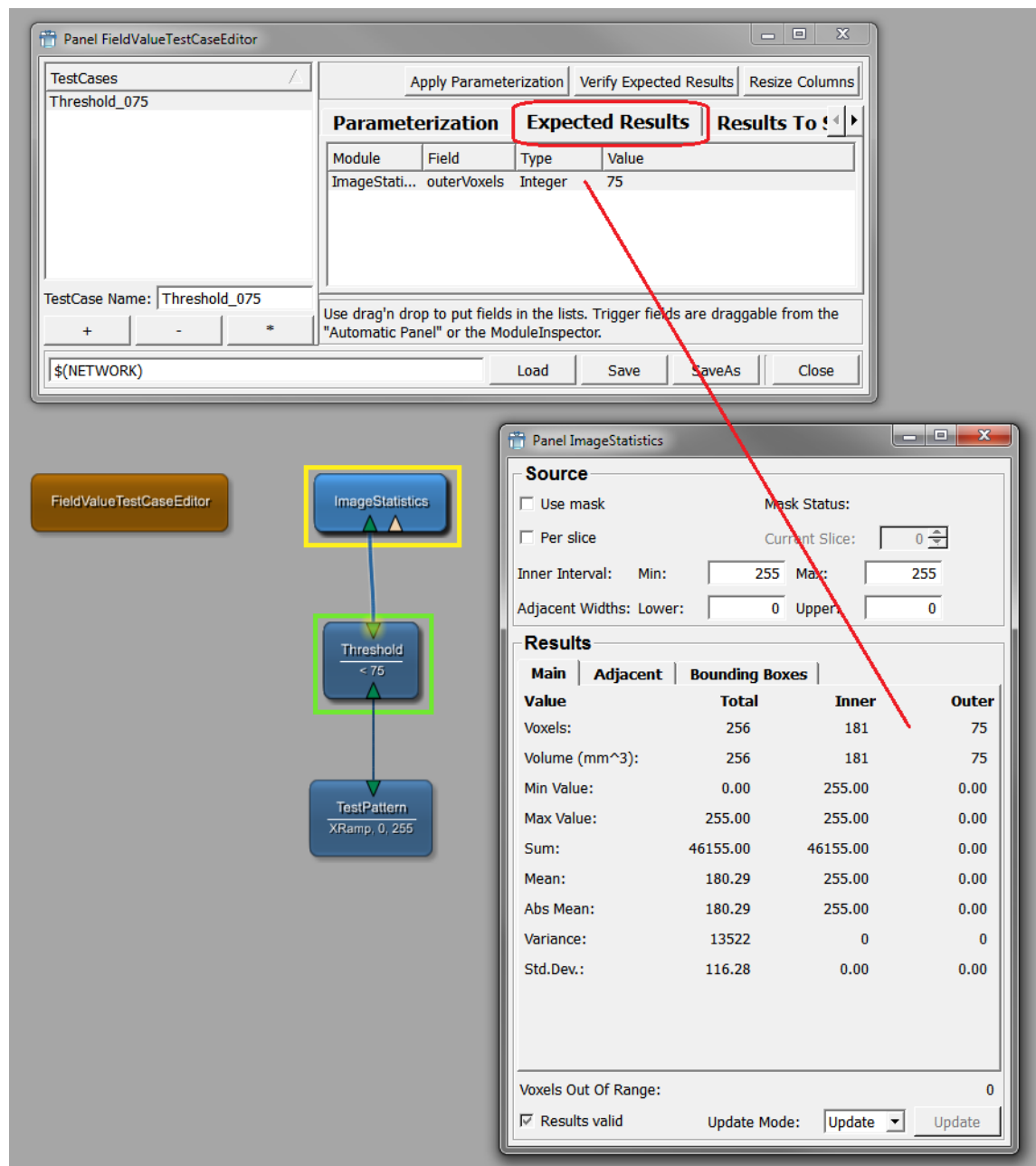
To save the FV test case set later, a data folder has to exist below the test case, for example `MyThresholdTest/data`. If no data folder exists yet, create it now.

To create a small set of three FV test cases for different threshold values, proceed as follows:

1. Click on the + button beneath the FV TestCases list to add a new test case.
2. Enter the FV TestCase name, for example "Threshold_75" and press **RETURN**.
3. Add the necessary parameters, in our case the threshold value of the module `Threshold`. To do this, drag the field from the module's panel onto the **Parameterization** tab.

Figure 16.8. Dragging Fields into the Parameter List

4. Click on the **Expected Results** tab to enter the expected result. In our case, it is a value of "75" for the *outerVoxels* parameter, so drag this parameter into the list and edit the value, if necessary.

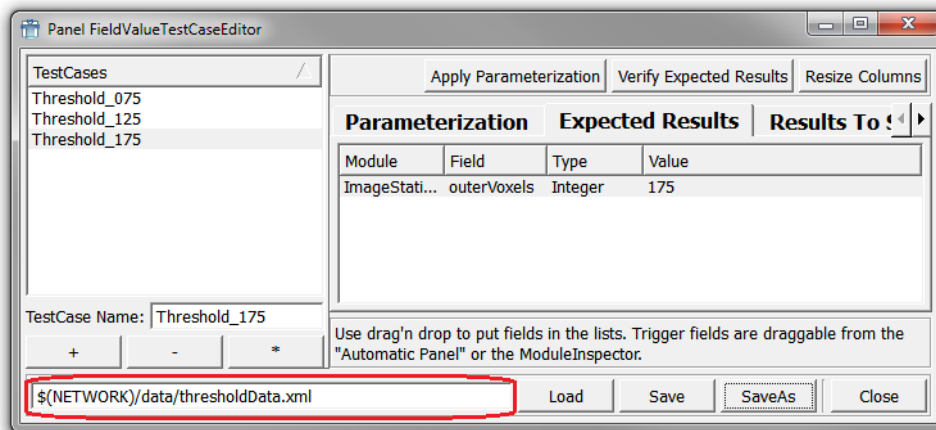
Figure 16.9. Dragging Fields into the Expected Results List

5. Select your FV test case and click the * button twice to duplicate the entry, as we need two further test cases for threshold = 125 and threshold = 175.
6. Edit each new FV test case by adapting the name of the function, the used threshold value, and the expected result value.

**Note**

The processing order is alphabetically, so for sorting the order of your test cases, enter the test case names accordingly.

7. In the field on bottom, enter the path and file name as \$(NETWORK)/data/thresholdData.xml, then click **Save**.

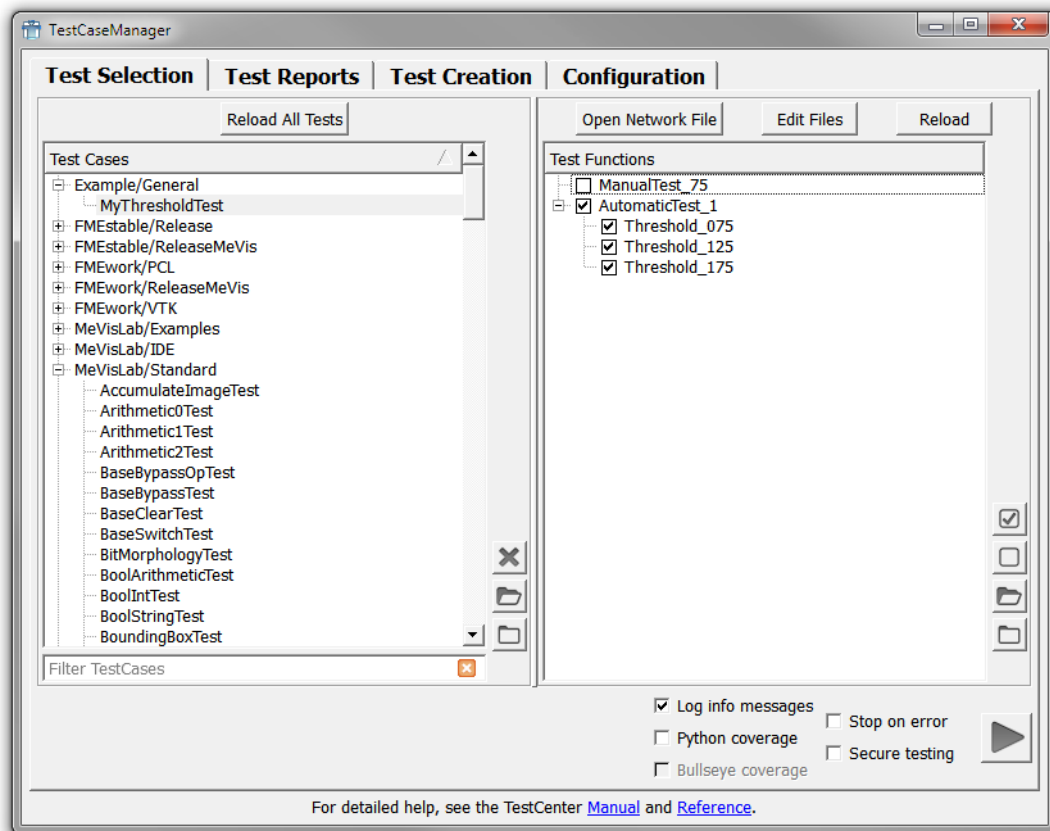
Figure 16.10. The Resulting Panel

8. For integrating the new FV test cases, add the following two things to your scripting code:
 - Add `import os` so that your function can use the Python functions for handling platform-dependent strings.
 - Add the new test function beneath the first:

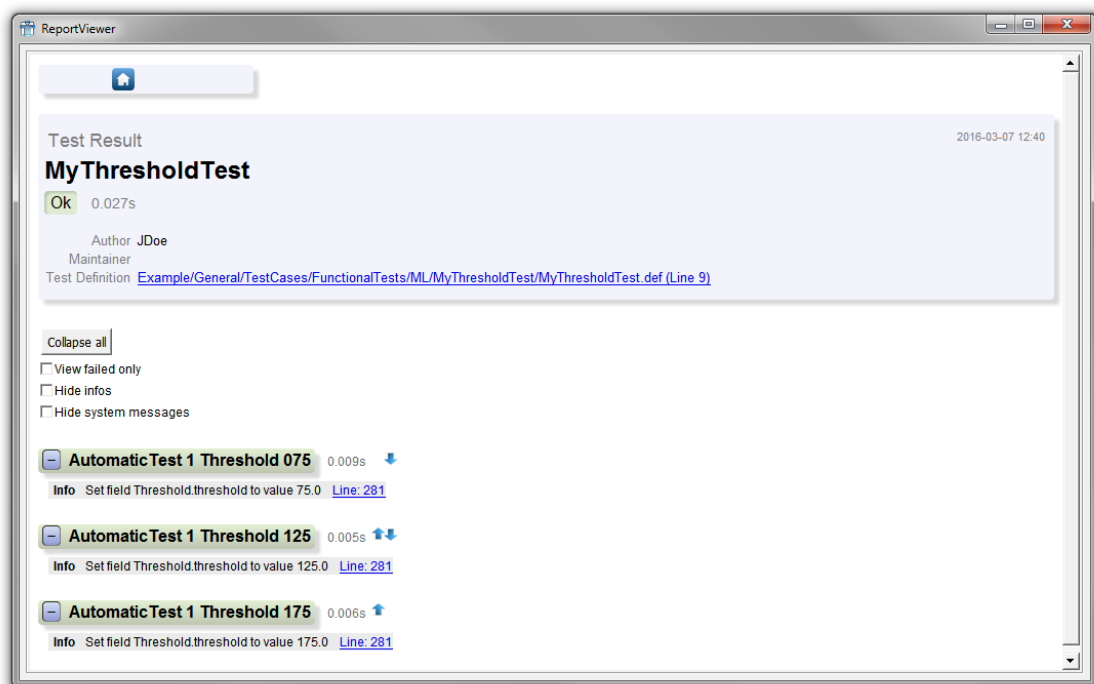
```
def FIELDVALUETEST_AutomaticTest_1():
    return os.path.join(Base.getDataDirectory(), "thresholdData.xml")
```

The return path expects the test case data file we just created.

9. In the TestCaseManager, reload the test case.

Figure 16.11. Our Automatic FieldValue Tests Added

10. Select “AutomaticTest_1” and run it. The report should look as follows:

Figure 16.12. Report for AutomaticTest_1

**Tip**

If you want to use only a subset of the field-value test cases, explicitly add the relevant subset at the end of the line, for example:

```
return os.path.join(Base.getDataDirectory(), "thresholdData.xml"), \
        ['Threshold_075', 'Threshold_175']
```

This way, only the test cases for threshold values of 75 and 175 would be run, while the test case for value 125 would be omitted.

**Tip**

For another field-test example, see “ExampleTestCase5” in the test cases for the MeVisLab/Standard package.

16.2.6. Automating the Test Case with an Iterative Test

For this, the test function we implemented first will be used with a parameter instead of a fixed threshold value, and the parameter is changed in the test function.

1. Add the new test function:

```
def ITERATIVETEST_AutomaticTest_2():
    return {'075':075,'125':125,'175':175}, computeVoxels
```

Instead of a simple list, we use the Python's dictionary class here to have a nicer listing.

**Note**

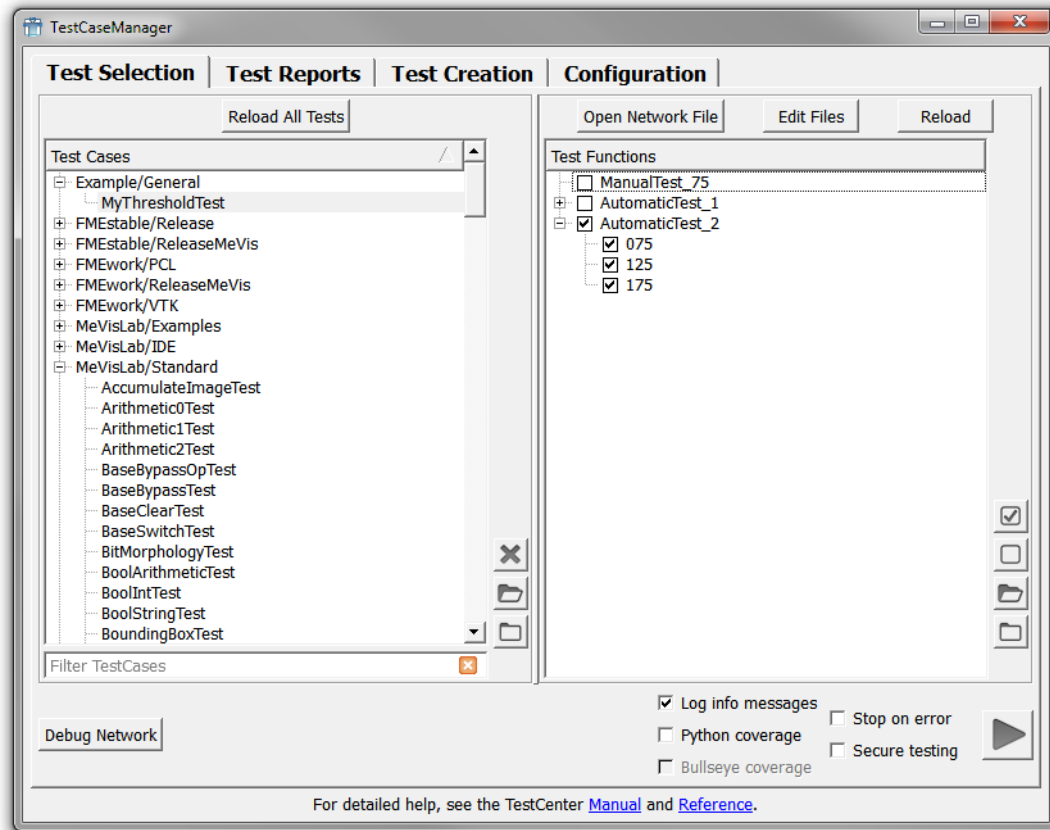
The processing order is alphabetically (and not given by the dictionary's order!), so for setting the order of your test cases here, enter the dictionary names accordingly.

2. Add the actual test:

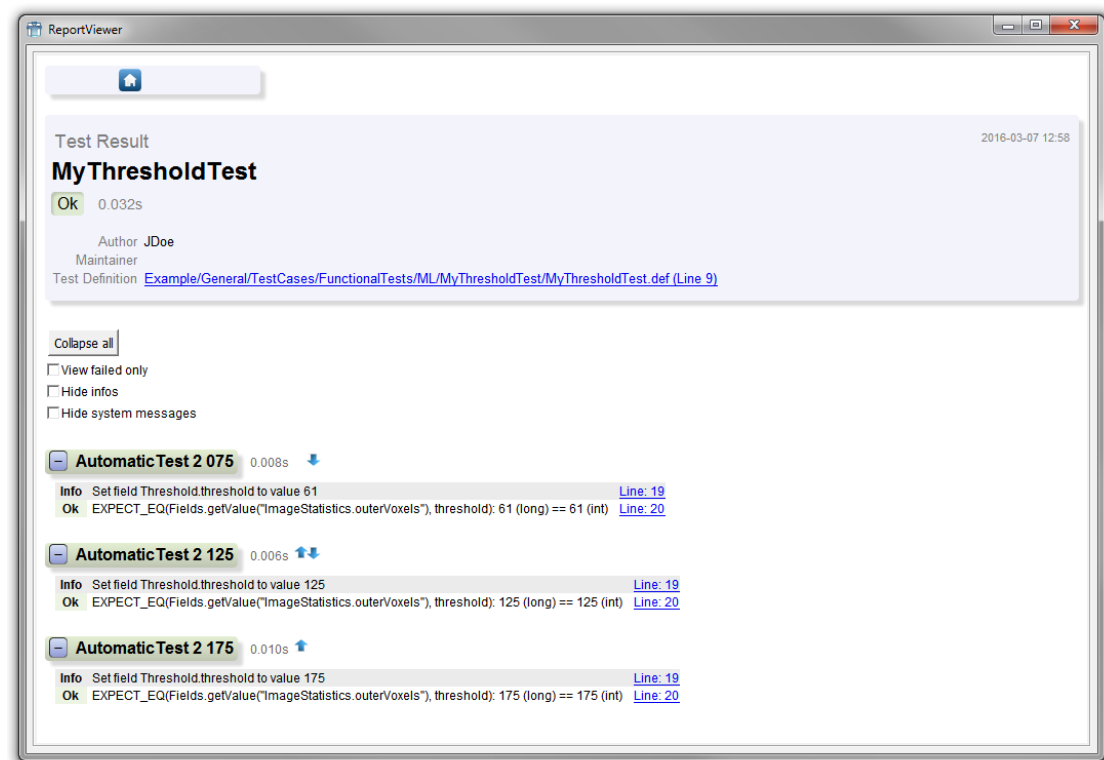
```
def computeVoxels(threshold):
    Fields.setValue("Threshold.threshold", threshold)
    EXPECT_EQ(Fields.getValue("ImageStatistics.outerVoxels"), threshold)
```

The `computeVoxels` function is essentially the same function as entered for the manual test case, but now using the parameter `threshold`. The function is called for every entry in the dictionary.

3. In the TestCaseManager, reload the test case.

Figure 16.13. Our Iterative Test in the Test Center

4. select "AutomaticTest_2" and click on **Run** to run the test function. The report should look as follows:

Figure 16.14. Report for AutomaticTest_2

16.2.7. Grouping Test Functions

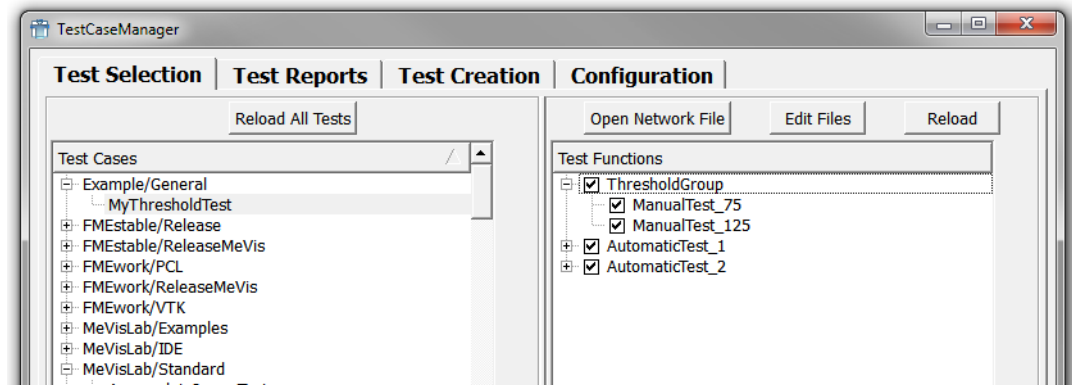
TEST functions can be grouped. This is useful for grouping tests in the Test function list.

1. For a quick example, simply copy "ManualTest_75" and change the "75" in name and value to "125". (In reality, nobody would want to group such redundant test cases but would make use of the automation approaches as described above.) Make sure to give the test a new number, so the resulting test function name might be "TEST004_ManualTest_125".

2. Add the group definition:

```
def GROUP_ThresholdGroup():
    return (TEST_ManualTest_75, TEST_ManualTest_125)
```

3. Save the scripting.
4. In the TestCaseManager, reload the test case. The new "ThresholdGroup" appears in the test functions list. It looks and works similar to the automatic tests.

Figure 16.15. Grouped Test Functions

16.2.8. Enhancing Test Reports with ScreenShots

Screenshots can easily be created with the `ScreenShot` method.

Here a quick example:

1. Create a new test case called "MyScreenShotTest".
2. To the example network, add the modules `LocalImage` and `View2D` and connect them. Save the network.
3. Then edit the scripting:

- Configure the `LocalImage` module by setting the image path:

```
Fields.setValue("LocalImage.name", "${DemoDataPath}/Bone.tiff")
```

- Configure the `View2D` module, for example by setting the slice:

```
Fields.setValue("View2D.startSlice", 0)
```

- Add the screenshot method and store the result in a variable:

```
result = ScreenShot.createOffscreenScreenShot("View2D.self", "screentest.png")
```

- Add two lines that make the result available in the report:

```
Logging.showImage("My screenshot", result)
Logging.showFile("Link to screenshot file", result)
```

The full code is:

```
from TestSupport import Fields, Logging, ScreenShot
from TestSupport.Macros import *

def TEST_Create_ScreenShot ():
    """ -- Creates a single screenshot -- """
    Fields.setValue("LocalImage.name", "${DemoDataPath}/Bone.tiff")
    Fields.setValue("View2D.startSlice", 0)
    result = ScreenShot.createOffscreenScreenShot("View2D.self", "screentest.png")
    Logging.showImage("My screenshot", result)
    Logging.showFile("Link to screenshot file", result)
```

4. Save it all and run the test function.

The report should look as follows:

Figure 16.16. Report for ScreenShot Example



Tip

For a more complex screenshot example, see “ExampleTestCase4” in the test cases for the MeVisLab/Standard package.

This was a short, practical introduction to the MeVisLab TestCenter. For further information, see the TestCenter Reference.

16.2.9. Disabling Test Functions

It may be desired to disable test functions when they always fail because of a known bug. To do so append the prefix “DISABLED_” to the function name.

Test functions can also be disabled depending on a condition using the `disableTestFunctionIf(condition)` decorator. `condition` can be a truth value or a callable.

```
from TestSupport.Base import disableTestFunctionIf
from TestSupport.Macros import *
```

```
def canCreateScreenShots():
    if [...]:
        return True
    else:
        return False

# Disable this test function if screenshots cannot be created:
@disableTestFunctionIf(not canCreateScreenShots())
def TEST_Create_ScreenShot():
    [...]

# Disable this test function if the platform is unix:
@disableTestFunctionIf(MLAB.isUnix)
def TEST_TestWithWin32API():
    [...]
```