FW4SPL Documentation

Release 0

IRCAD-IHU

April 04, 2016

Contents

CHAPTER 1

Introduction

1.1 Repositories

The fw4spl project is organized organized around three repositories :

- fw4spl: main repository, contains the core libraries and bundles.
- fw4spl-ext: extension of fw4spl repository, contains additional functionalities and proofs of concept
- fw4spl-ar: extension of fw4spl, contains functionalities for augmented reality (video tracking)

Each of this repository needs the associated deps repository

- fw4spl-deps
- fw4spl-ext-deps
- fw4spl-ar-deps

1.2 fw4spl

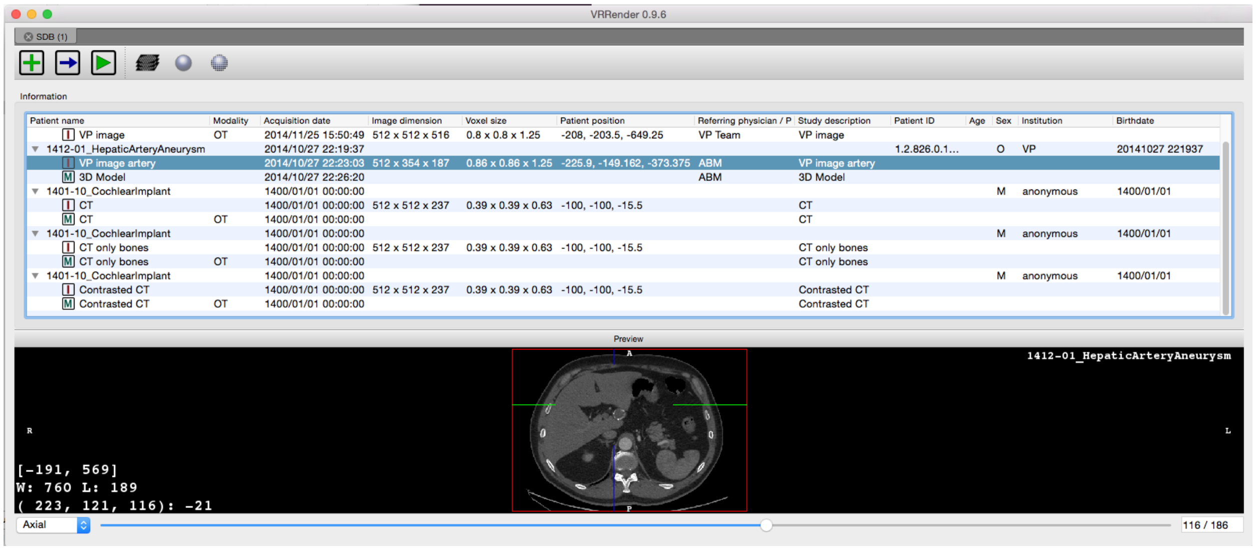
This repository contains the core libraries and bundles.

1.2.1 Features

- **Reader/Writer**
  - VTK (images and meshes)
  - DICOM
  - ITK
  - atoms (in-houte data format)
- **Visualisation**
  - 2D and 3D multi-planar reconstruction
  - volume rendering
  - 3D meshes

1.2.2 Application

VRRender is an application containing all the previous features.



Patient name	Modality	Acquisition date	Image dimension	Voxel size	Patient position	Referring physician / P	Study description	Patient ID	Age	Sex	Institution	Birthdate
I VP image	OT	2014/11/25 15:50:49	512 x 512 x 516	0.8 x 0.8 x 1.25	-208, -203.5, -649.25	VP Team	VP image					
▼ 1412-01_HepaticArteryAneurysm		2014/10/27 22:19:37						1.2.826.0.1...		O	VP	20141027 221937
I VP image artery		2014/10/27 22:23:03	512 x 354 x 187	0.86 x 0.86 x 1.25	-225.9, -149.162, -373.375	ABM	VP image artery					
M 3D Model		2014/10/27 22:26:20				ABM	3D Model					
▼ 1401-10_CochlearImplant		1400/01/01 00:00:00								M	anonymous	1400/01/01
I CT		1400/01/01 00:00:00	512 x 512 x 237	0.39 x 0.39 x 0.63	-100, -100, -15.5		CT					
M CT	OT	1400/01/01 00:00:00					CT					
▼ 1401-10_CochlearImplant		1400/01/01 00:00:00								M	anonymous	1400/01/01
I CT only bones		1400/01/01 00:00:00	512 x 512 x 237	0.39 x 0.39 x 0.63	-100, -100, -15.5		CT only bones					
M CT only bones	OT	1400/01/01 00:00:00					CT only bones					
▼ 1401-10_CochlearImplant		1400/01/01 00:00:00								M	anonymous	1400/01/01
I Contrasted CT		1400/01/01 00:00:00	512 x 512 x 237	0.39 x 0.39 x 0.63	-100, -100, -15.5		Contrasted CT					
M Contrasted CT	OT	1400/01/01 00:00:00					Contrasted CT					

Fig. 1.1: Main VRRender view.

1.2.3 Tutorials

You can find some tutorials to explain fw4spl concept.

Name	Concept
Tuto01Basic	Basic application
Tuto02DataServiceBasic	Simple image reading and rendering
Tuto02DataServiceBasicCtrl	Simple image reading and rendering without XML configuration
Tuto03DataService	Image reading and rendering with signal communication
Tuto04SignalSlot	Scene point of view synchronisation with signal communication
Tuto05Mesher	Simple mesher from a 3D image
Tuto06Filter	Simple image filter
Tuto08GenericScene	Scene with multi-object rendering
Tuto09MesherWithGenericScene	Scene with multi-object rendering and simple mesher
Tuto10MatrixTransformInGS	Example of matrix transformation
Tuto11LaunchBasicConfig	Example to launch XML config in application
Tuto12Picker	Example of scene picker
Tuto13Scene2D	Example using the ‘‘scene2d‘‘bundle
Tuto14MeshGenerator	Mesh features (point/cell color, normals, ...)
Tuto15Multithread	Example of multi-threading using fw4spl worker
Tuto15MultithreadCtrl	Second example of multi-threading using fw4spl worker
TutoGui	Example of fw4spl gui feature (toolbar, menu, action)
TutoPython	Example of pyhton binding in fw4spl
TutoTrianConverterCtrl	Utility converting .trian meshes to .vtk

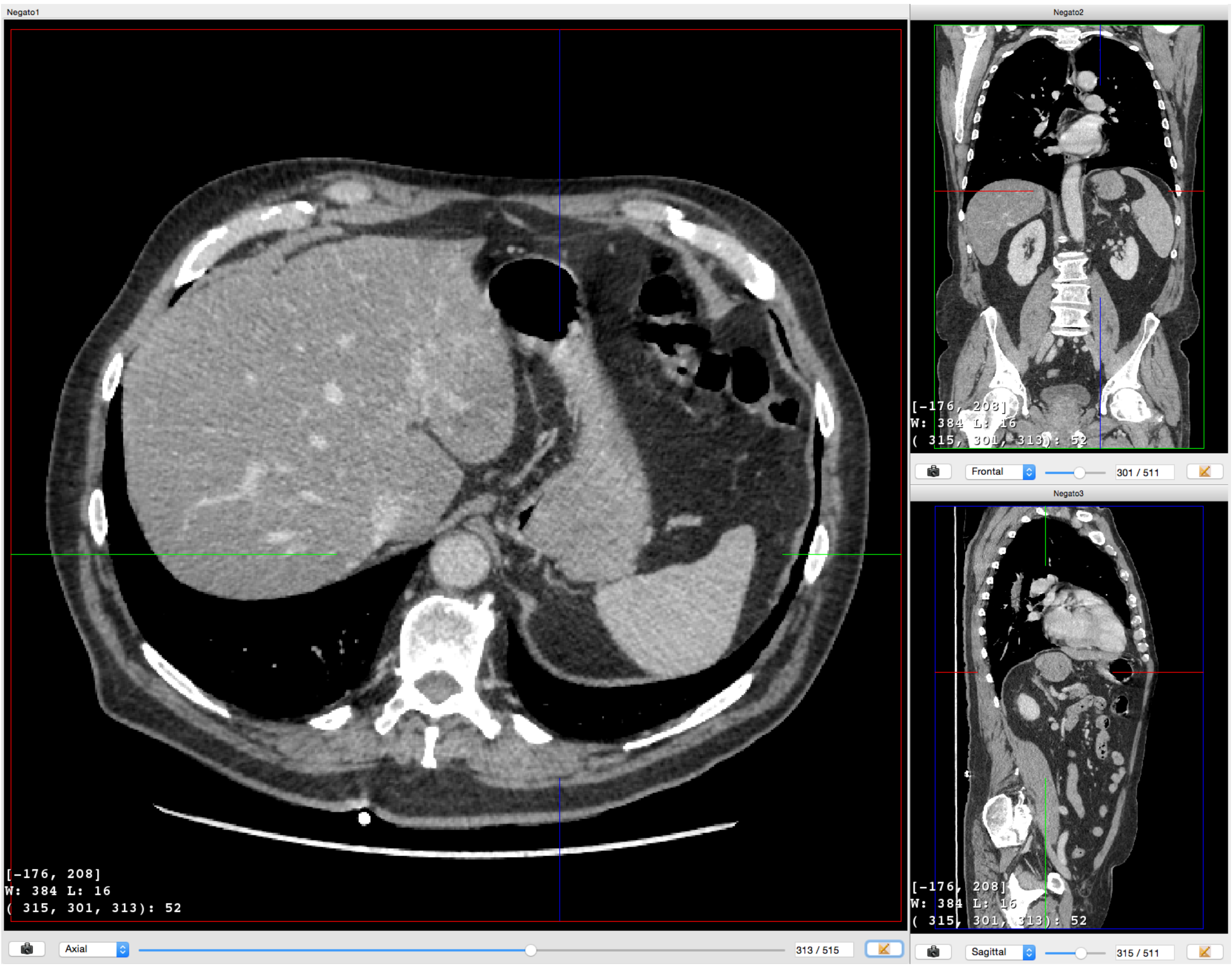


Fig. 1.2: MPR view of a medical 3D image.

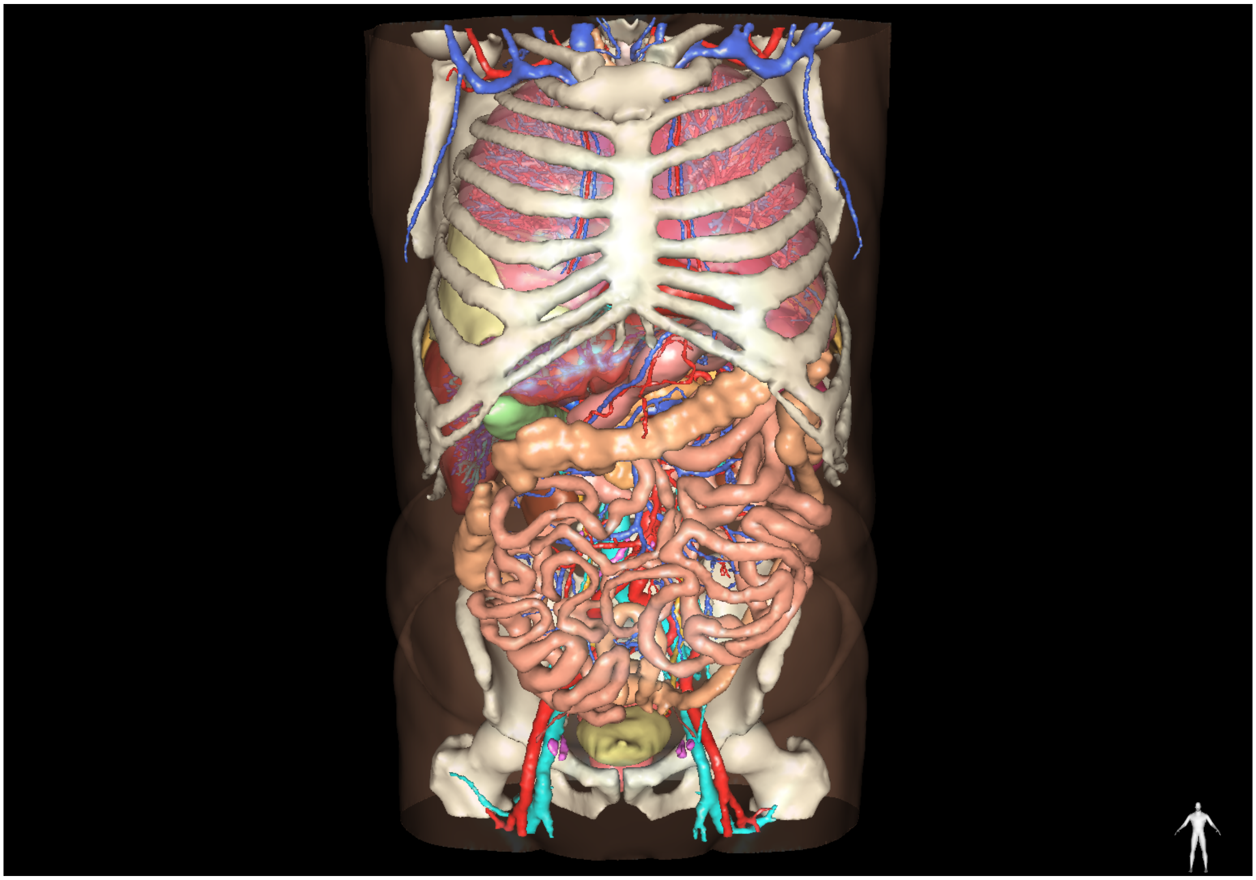


Fig. 1.3: 3D view of surfacic meshes.

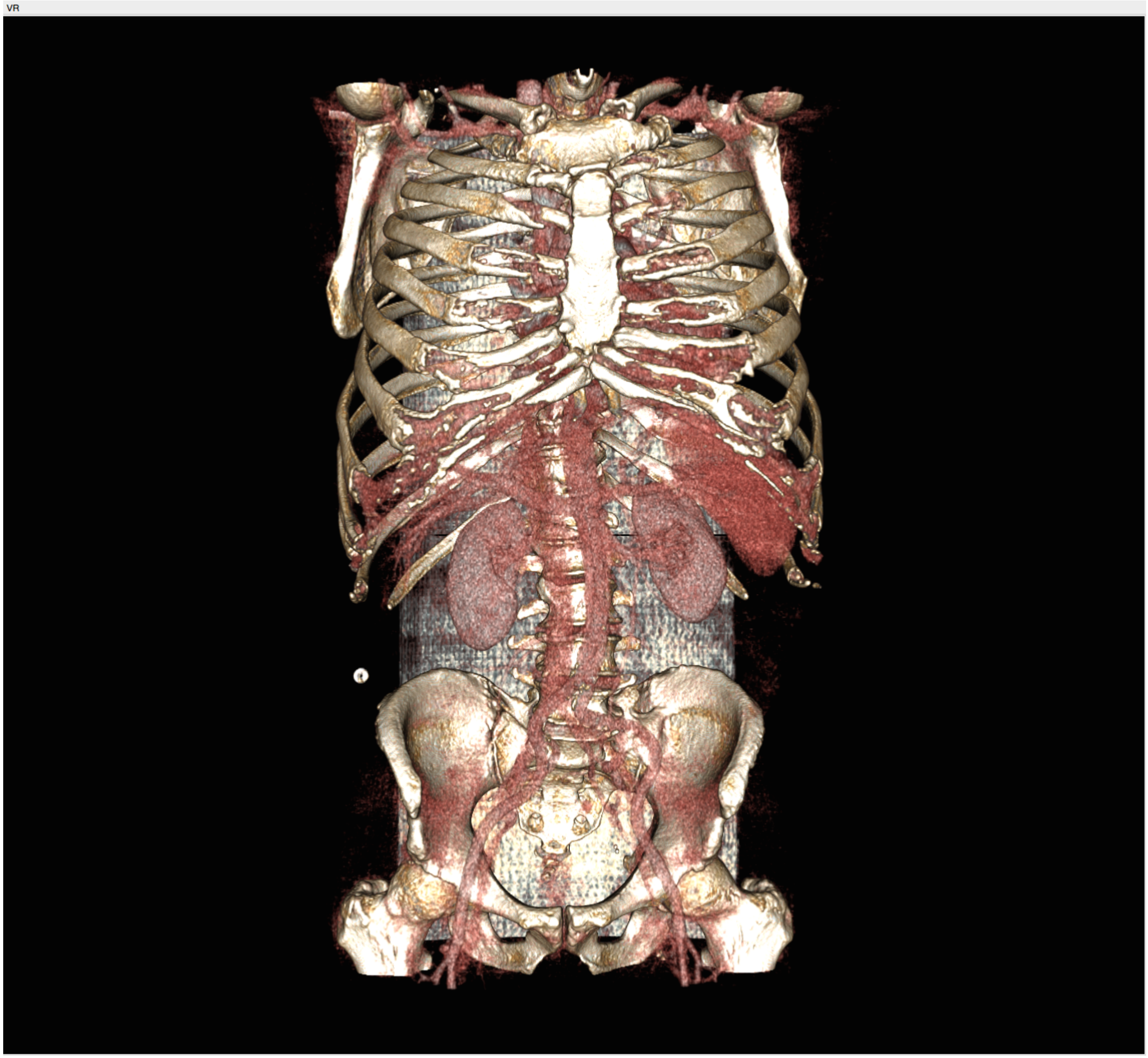


Fig. 1.4: Volume rendering

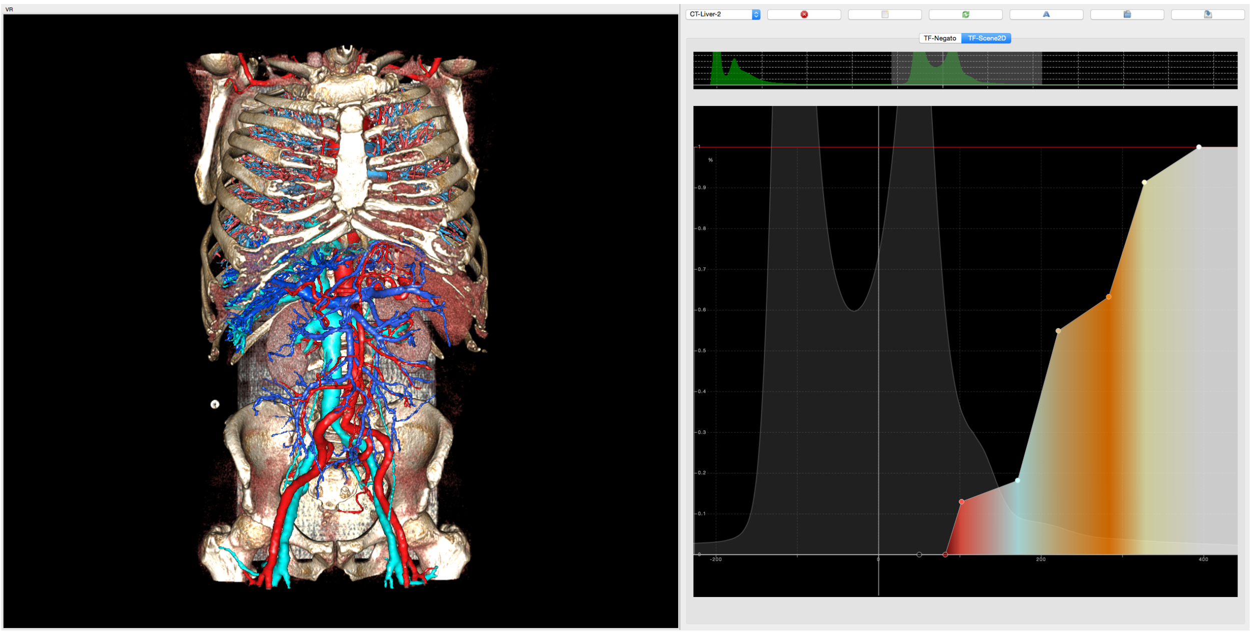


Fig. 1.5: Volume rendering mixed with 3D surfacic meshes.

1.3 fw4spl-ext

This repository contains additional functionalities and proofs of concept.

1.3.1 Features

- **additional DICOM reader/writer**
  - PACS connection
  - 3D mesh segmentation reader/writer
  - DICOM filter for reader
- navigation along a spline
- timeline
- network communication via openigtlink

1.3.2 Application

VRRenderExt is an application containing the **VRRender** features and also the additional fw4spl-ext features.

1.3.3 Proofs of concept

Name	Concept
PoC06Scene2DTF	Simple use of `scene2d` bundle
PoC07TimeLine	Timeline use with consumer/producer
PoC08Igtl	Network communication with openigtlink

1.3.4 Examples

Name	Concept
Ex01VolumeRendering	Example of volume rendering using transfer function
Ex02ImageMix	Example of image blend
Ex03Registration	Example of simple rigid image-mesh registration
Ex04ImagesRegistration	Example of simple rigid image-image registration

1.4 fw4spl-ar

This repository contains functionalities for augmented reality.

1.4.1 Features

- video calibration
- optical tag tracking

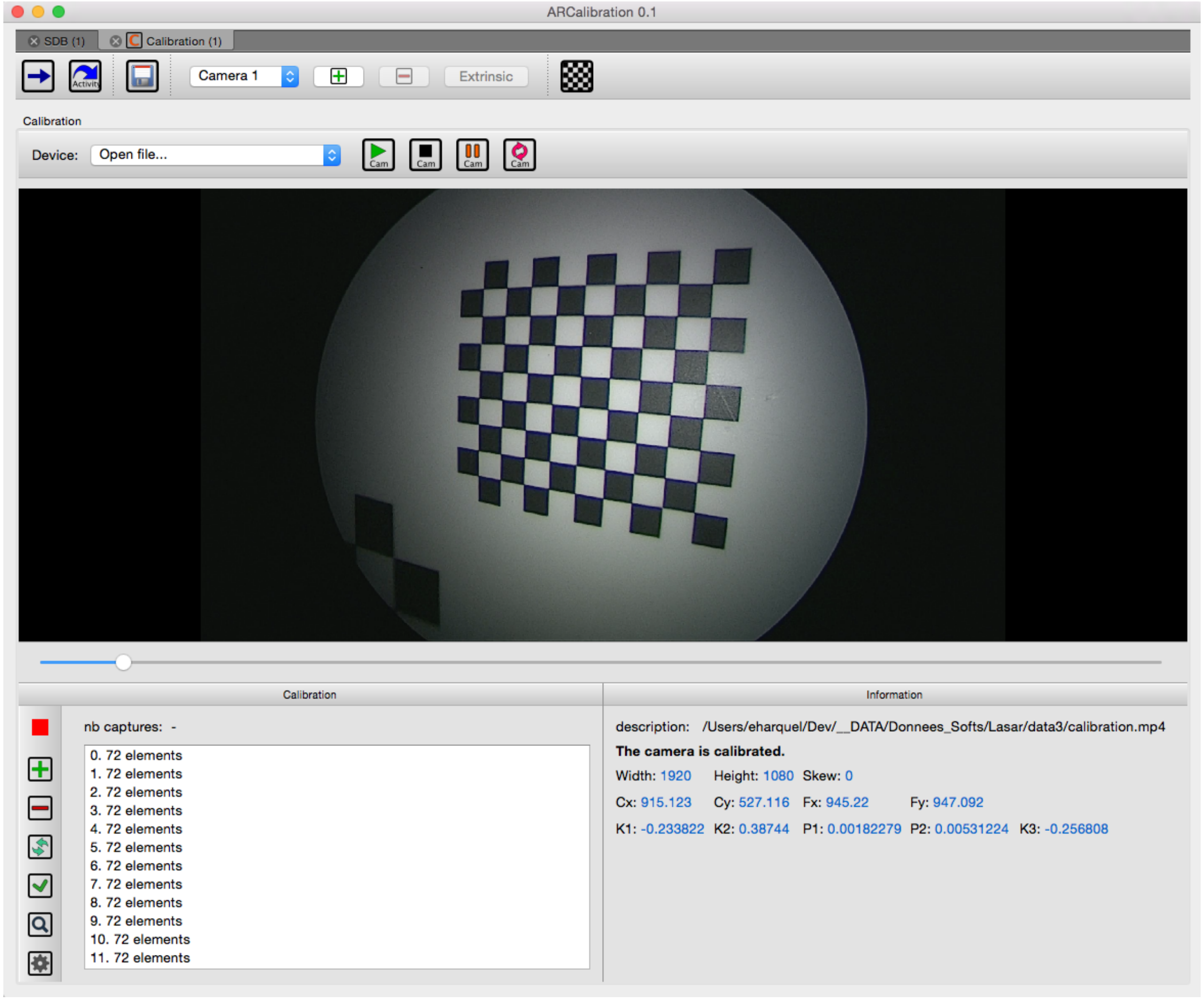
1.4.2 Applications

ARCalibration

ARCalibration is an application dedicated to the camera calibration.

VideoTracking

VideoTracking is a basic application for optical tag tracking.



Camera 1

Device: Open file...

nb captures: -

0. 72 elements
1. 72 elements
2. 72 elements
3. 72 elements
4. 72 elements
5. 72 elements
6. 72 elements
7. 72 elements
8. 72 elements
9. 72 elements
10. 72 elements
11. 72 elements

description: /Users/eharquel/Dev/__DATA/Donnees_Softs/Lasar/data3/calibration.mp4

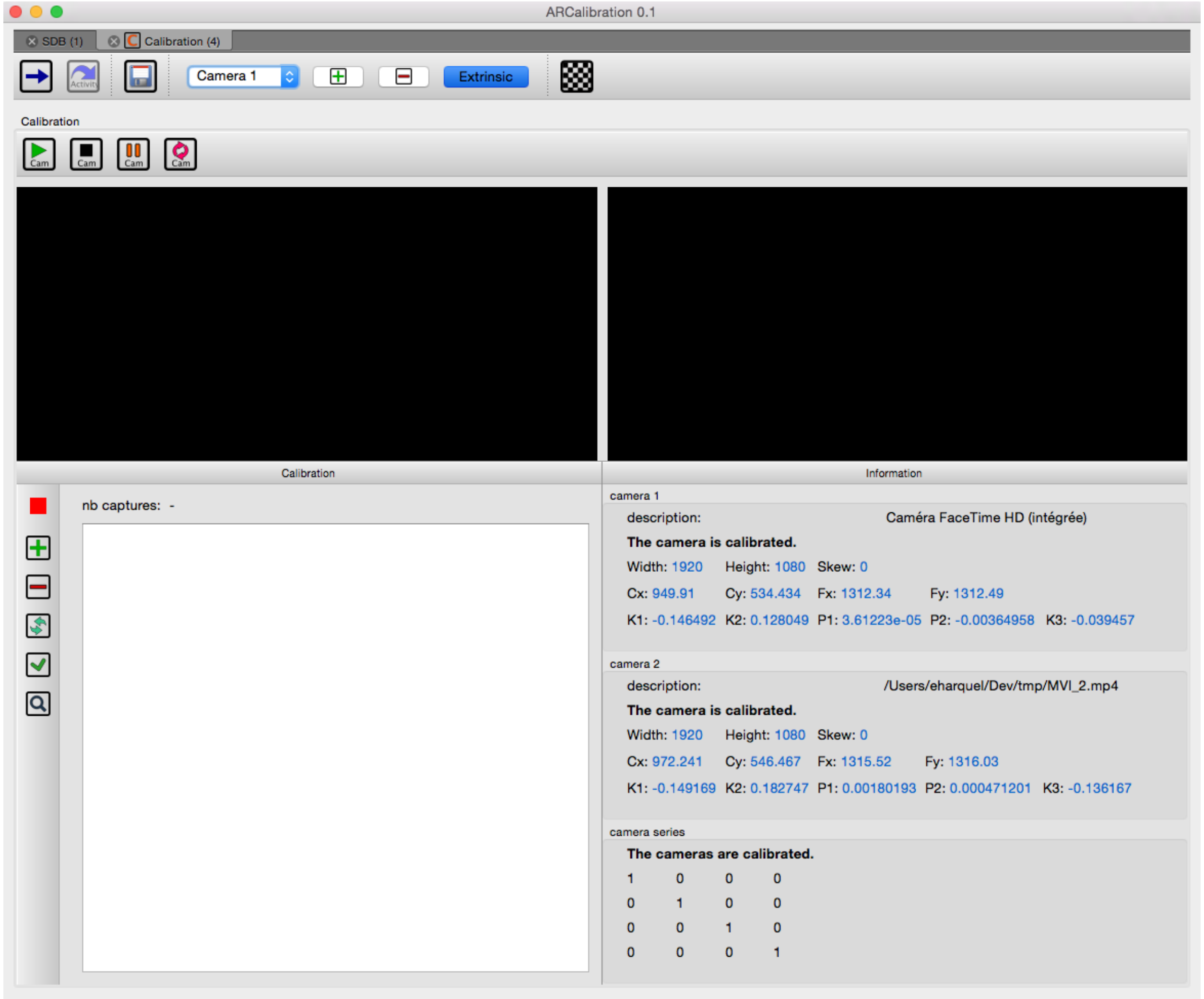
The camera is calibrated.

Width: 1920 Height: 1080 Skew: 0

Cx: 915.123 Cy: 527.116 Fx: 945.22 Fy: 947.092

K1: -0.233822 K2: 0.38744 P1: 0.00182279 P2: 0.00531224 K3: -0.256808

Fig. 1.6: Mono camera intrinsic calibration.



nb captures: -

camera 1

description: Caméra FaceTime HD (intégrée)

The camera is calibrated.

Width: 1920 Height: 1080 Skew: 0

Cx: 949.91 Cy: 534.434 Fx: 1312.34 Fy: 1312.49

K1: -0.146492 K2: 0.128049 P1: 3.61223e-05 P2: -0.00364958 K3: -0.039457

camera 2

description: /Users/eharquel/Dev/tmp/MVI_2.mp4

The camera is calibrated.

Width: 1920 Height: 1080 Skew: 0

Cx: 972.241 Cy: 546.467 Fx: 1315.52 Fy: 1316.03

K1: -0.149169 K2: 0.182747 P1: 0.00180193 P2: 0.000471201 K3: -0.136167

camera series

The cameras are calibrated.

```
1    0    0    0
0    1    0    0
0    0    1    0
0    0    0    1
```

Fig. 1.7: Stereo camera extrinsic calibration.

CHAPTER 2

Installation

2.1 Latest Release Version

The last release of the project is:

- fw4spl_0.10.2.3

2.2 Installation for Windows

2.2.1 Prerequisites for Windows users

If not already installed:

1. Install Mercurial
2. Optionally you can install TortoiseHg
3. Install Visual Studio 2013 Community
4. Install Python 2.7
5. Install CMake
6. Install jom
7. Install ninja

Qt is an external library used in FW4SPL. For the successful compilation of Qt for FW4SPL, please see the following requirements:

- http://wiki.qt.io/Building_Qt_5_from_Git

2.2.2 FW4SPL installation

Good practice in FW4SPL recommend to separate source files, build and install folders. So to prepare the development environment:

- Create a development folder (Dev)
- Create a build folder (Dev\Build)
  - Add a sub folder for Debug and Release.
- Create a source folder (Dev\Src)

- Create a install folder (Dev\Install)
  - Add a sub folder for Debug and Release.

To prepare the third party environment:

- Create a third party folder (BinPkgs)
- Create a build folder (BinPkgs\Build)
  - Add a sub folder for Debug and Release.
- Create a source folder (BinPkgs\Src)
- Create an install folder (BinPkgs\Install)
  - Add a sub folder for Debug and Release.
- Set environment for a x64 version. For compile BinPkgs and sources, you must use the 'VS2013 x64 Native Tools Command Prompt'

Dependencies

> **Warning:** Be sure to be in the 'VS2013 x64 Native Tools Command Prompt'

- Clone the three following repositories in the (BinPkgs) source folder:
  - fw4spl-deps
  - fw4spl-ar-deps
  - fw4spl-ext-deps
- Update the cloned repositories to the used version.

Note: Make sure that CMake is set as environment variable.

- Call the cmake-gui
- During Configure, choose the generator 'NMake Makefiles JOM'.

Note: make sure the generator JOM are set in your PATH.

- Set the following arguments:
  - *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps
  - *CMAKE_INSTALL_PREFIX*: set the install location.
  - *CMAKE_BUILD_TYPE*: set to Debug or Release
- Generate the code.
- Compile the FW4SPL dependencies with jom in the console (e.g. jom all, jom qt, etc).

Source

> **Warning:** Be sure to be in the 'VS2013 x64 Native Tools Command Prompt'

- Clone the three following repositories in the (Dev) source folder:
  - fw4spl
  - fw4spl-ar
  - fw4spl-ext
- Update the cloned repositories to the used version.

Note: Make sure that CMake is in your PATH.

- Call the cmake-gui.
- During Configure, choose the generator ('NMake Makefiles JOM' for compile BinPkgs or 'Ninja' for compile FW4SPL sources)

Note: make sure the generator Ninja and JOM are set in your PATH.

- Set the following arguments:
  - *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext
  - *CMAKE_INSTALL_PREFIX*: set the install location.
  - *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.
  - *CMAKE_BUILD_TYPE*: set to Debug or Release
  - *PROJECT_TO_BUILD* set the name of the application to build (See DevSrcApps)

  Note: If PROJECT_TO_BUILD is empty, all application will be compile

  - *PROJECT_TO_INSTALL* set the name of the application to install

  Note: If PROJECT_TO_BUILD is empty, all application will be compile

Warning: Make sure the arguments concerning the compiler (advanced arguments) point to Visual Studio.

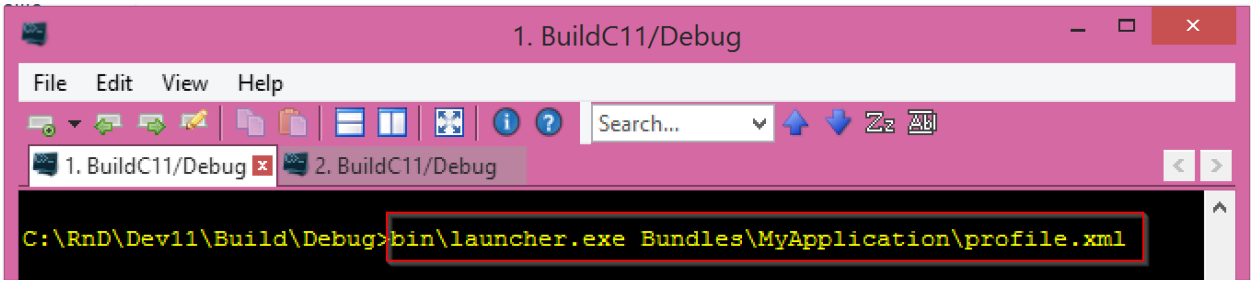
- Generate the code.
- Compile the FW4SPL source code with ninja in the console.

Note: it is possible to generate eclipse project with CMake. You just have to check ECLIPSE_PROJECT.

2.2.3 Launch an application

After an successful compilation the application can be launched with the launcher.exe from FW4SPL. Therefore the profile.xml of the application in the build folder has to be passed as argument.

Note: Make sure that the external libraries directory is set to the path (set PATH=<FW4SPL Binpkgs path>\Debug\bin;<FW4SPL Binpkgs path>\Debug\x64\vc12\bin;%PATH%).



```
C:\RnD\Dev11\Build\Debug>bin\launcher.exe Bundles\MyApplication\profile.xml
```

2.2.4 Recommended software

The following programs may be helpful for your developments:

- Eclipse CDT: Eclipse is a multi-OS Integrated Development Environment (IDE) for computer programming.
- Notepad++: Notepad++ is a free source code editor, which is designed with syntax highlighting functionality.
- ConsoleZ: ConsoleZ is an alternative command prompt for Windows, adding more capabilities to the default Windows command prompt. To compile FW4SPL with the console the windows command prompt has to be set in the tab settings.

2.3 Installation for Linux

2.3.1 Prerequisites for Linux users

If not already installed:

1. Install git
2. Install gcc The minimal version required is 4.8 or clang The minimal version required is 3.5
3. Install Python 2.7
4. Install CMake The minimal version required is 3.0
5. Install Ninja

Depending on which linux distribution you use, for example on Debian you can do:

```
$ apt-get install build-essential ninja-build python2.7 git
```

Qt is an external library used in FW4SPL. For the successful compilation of Qt with FW4SPL, please see the following requirements:

- http://wiki.qt.io/Building_Qt_5_from_Git

2.3.2 FW4SPL installation

FW4SPL works with data separation for source, build and install data. To prepare the development environment:

- Create your “Dev” directory

```
$ mkdir Dev
```

- Create into “Dev” the source, build and install directories

```
$ mkdir Dev/Src Dev/Build Dev/Install
```

- Create sub-folders to separate Debug and Release compilations

```
$ mkdir Dev/Build/Debug Dev/Build/Release Dev/Install/Debug Dev/Install/Release
```

To prepare the third party environment:

- Create a third party folder (Deps)

```
$ mkdir Deps
```

- Create into “Deps” the source, build and install directories

```
$ mkdir Deps/Src Deps/Build Deps/Install
```

- Create sub-folders to separate Debug and Release compilations

```
$ mkdir Deps/Build/Debug Deps/Build/Release Deps/Install/Debug Deps/Install/Release
```

Build tools

FW4SPL is a CMake project. That means, for each build target there is a CMakeLists that provides build parameters. To configure you project you can use the `cmake` command from the build folder with the sources as arguments:

```
$ cd Dev/Build/Debug
$ ccmake ../../Src/fw4spl
```

if you want to use **Ninja** as build to tools, use the option `-G Ninja`, as following:

```
$ ccmake -G Ninja ../../Src/fw4spl
```

It is the same process for Deps and FW4SPL sources. It is recommended to use make to compile the deps.

Dependencies

- Clone the repository into your source directory of Deps

```
$ cd Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-deps.git fw4spl-deps
```

- Get into fw4spl-deps folder and update to the latest stable version

```
$ cd fw4spl-deps
$ git checkout fw4spl_0.10.2.3
```

- Get into your Build directory (Debug or Release) : here an example if you want to compile in DEBUG

```
$ cd Deps/Build/Debug
```

- Call ccmake and point to the sources

```
$ ccmake ../../Src/fw4spl-deps
```

To build the dependencies, you must configure the project with cmake into the Build folder

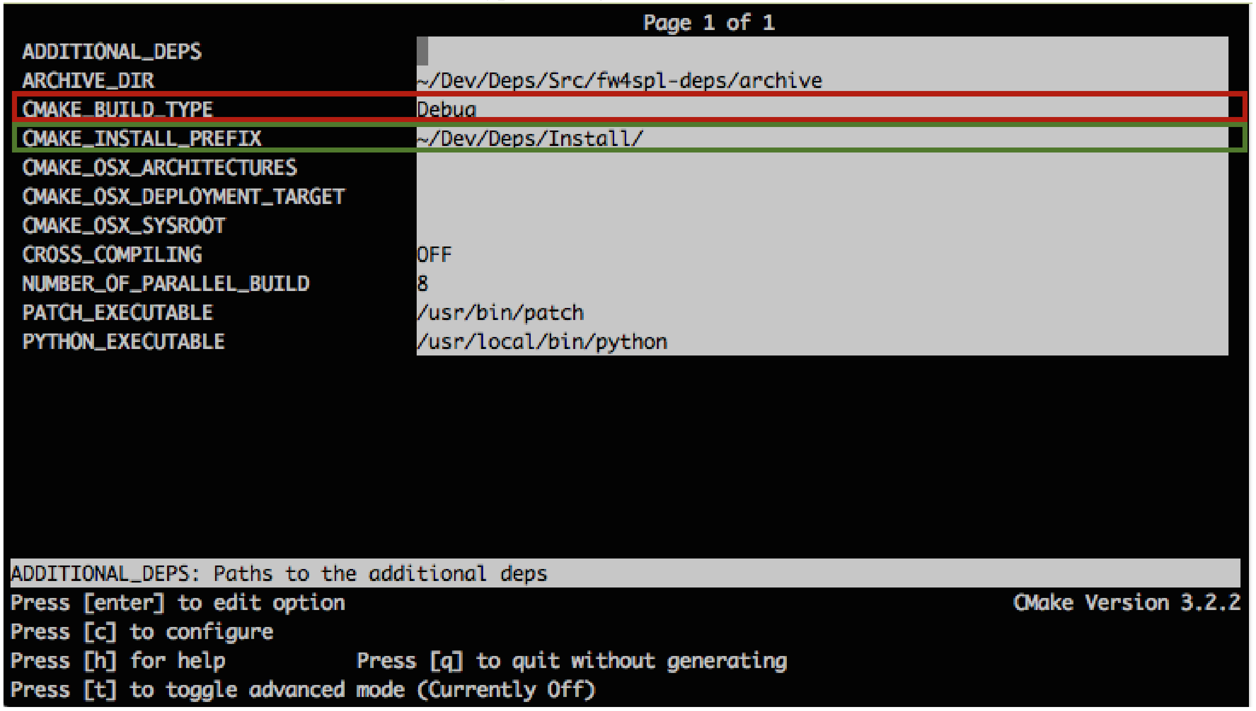
```
$ cd ~/Dev/Deps/Build
$ cmake ../Src/fw4spl-deps -DCMAKE_INSTALL_PREFIX=../Install -DCMAKE_BUILD_TYPE=Debug
```

Or open cmake gui editor, see *Build tools* instructions.

```
$ ccmake ../Src/fw4spl-deps
```

Some CMake variables have to be change:

- *CMAKE_INSTALL_PREFIX*: set the install location.
- *CMAKE_BUILD_TYPE*: set the build type 'Debug' or 'Release'



```
                                                   Page 1 of 1
 ADDITIONAL_DEPS
 ARCHIVE_DIR                      ~/Dev/Deps/Src/fw4spl-deps/archive
 CMAKE_BUILD_TYPE                 Debug
 CMAKE_INSTALL_PREFIX             ~/Dev/Deps/Install/
 CMAKE_OSX_ARCHITECTURES
 CMAKE_OSX_DEPLOYMENT_TARGET
 CMAKE_OSX_SYSROOT
 CROSS_COMPILING                  OFF
 NUMBER_OF_PARALLEL_BUILD         8
 PATCH_EXECUTABLE                 /usr/bin/patch
 PYTHON_EXECUTABLE                /usr/local/bin/python










ADDITIONAL_DEPS: Paths to the additional deps
Press [enter] to edit option                                   CMake Version 3.2.2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Press configure (*[c]*) and generate (*[g]*) makefiles.

- Compile the FW4SPL dependencies with make in the console, it will automaticaly download, build and install each dependencies.

```
$ make all
```

Warning: Do NOT use ninja to compile the dependencies, it cause conflict with qt compilation.

Source

- Clone fw4spl repository into your source directory

```
$ cd Dev/Src
$ git clone https://github.com/fw4spl-org/fw4spl.git fw4spl
```

- Get into fw4spl folder and update to the latest stable version

```
$ cd fw4spl
$ git checkout fw4spl_0.10.2.3
```

- Get into your Build directory (Debug or Release) : here an example if you want to compile in DEBUG

```
$ cd Dev/Build/Debug
```

- Call *ccmake* and point to the sources

To use make :

```
$ ccmake ../../Src/fw4spl
```

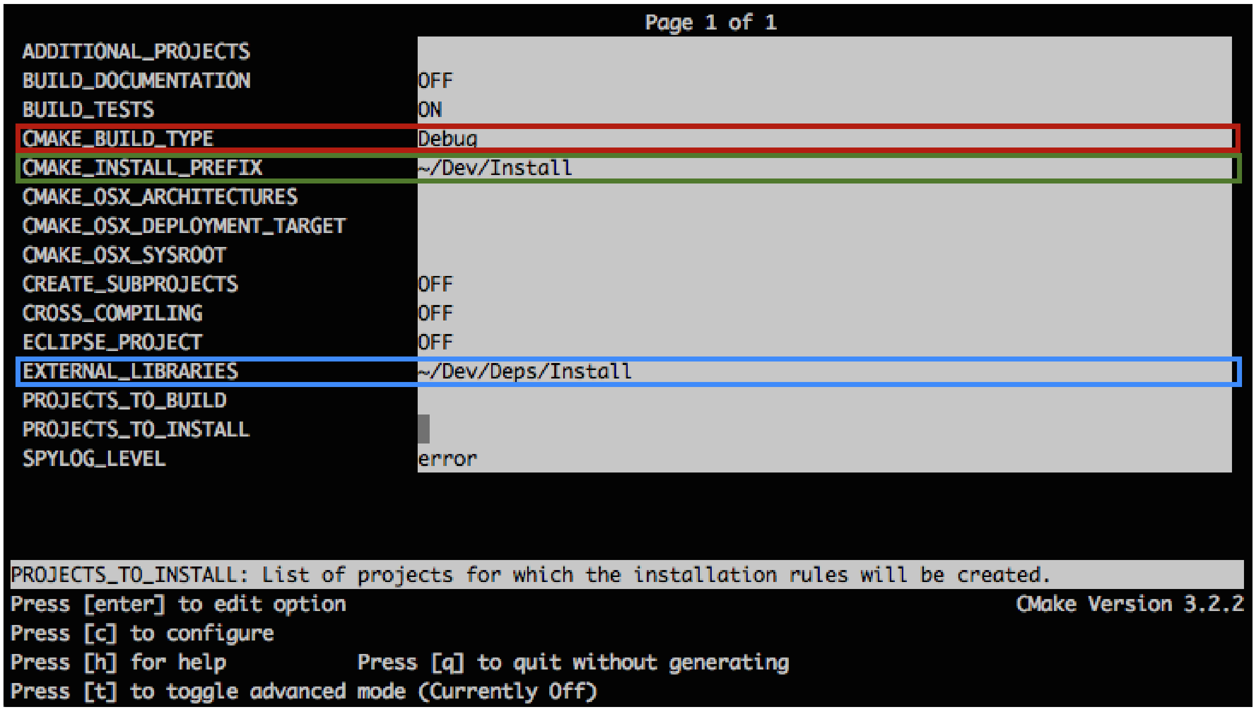
To use ninja :

```
$ ccmake -G Ninja ../../Src/fw4spl
```

- Change the following cmake arguments
  - *CMAKE_INSTALL_PREFIX*: set the install location (Dev/Install/Debug or Release)
  - *CMAKE_BUILD_TYPE*: set to DEBUG or RELEASE.
  - *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.(ex : Deps/Install/Debug)
  - *PROJECT_TO_BUILD*: set the list of the projects you want to build (ex: VRRender, Tuto01Basic ...), each project should be separated by ";".

Note:

- If PROJECT_TO_BUILD is empty, all application will be compiled
- If PROJECT_TO_INSTALL is empty, no application will be installed



```
                                     Page 1 of 1
 ADDITIONAL_PROJECTS
 BUILD_DOCUMENTATION              OFF
 BUILD_TESTS                      ON
 CMAKE_BUILD_TYPE                 Debug
 CMAKE_INSTALL_PREFIX             ~/Dev/Install
 CMAKE_OSX_ARCHITECTURES
 CMAKE_OSX_DEPLOYMENT_TARGET
 CMAKE_OSX_SYSROOT
 CREATE_SUBPROJECTS               OFF
 CROSS_COMPILING                  OFF
 ECLIPSE_PROJECT                  OFF
 EXTERNAL_LIBRARIES               ~/Dev/Deps/Install
 PROJECTS_TO_BUILD
 PROJECTS_TO_INSTALL
 SPYLOG_LEVEL                     error



PROJECTS_TO_INSTALL: List of projects for which the installation rules will be created.
Press [enter] to edit option                                        CMake Version 3.2.2
Press [c] to configure
Press [h] for help            Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Press configure (**[c]**) and generate (**[g]**) makefiles.

Then, build dependencies with ninja.

```
$ ninja all
```

2.3.3 Launch an application

To build a specific or several applications the CMake argument PROJECTS_TO_BUILD can be set. Use ; so separate each application name.

After an successful compilation the application can be launched with the launcher program from FW4SPL. Therefore the profile.xml of the application in the build folder has to be passed as argument to the launcher call in the console.

```
$ bin/launcher Bundles/MyApplication_Version/profile.xml
```

Example:

```
$ cd /Dev/Build
$ bin/launcher Bundles/VRRender_0-9/profile.xml
```

2.3.4 Extensions

fw4spl has two extension repositories:

- fw4spl-ext: contains additional functionalities and proofs of concept
- fw4spl-ar: contains functionalities for augmented reality (video tracking, calibration)

Dependencies

If you want to use this extension, you need to clone the deps repositories:

- fw4spl-ext-deps: contains the scripts to compile the external libraries used by fw4spl-ext

```
$ cd ~/Dev/Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ext-deps.git fw4spl-ext-deps
$ cd fw4spl-ext-deps
$ git checkout fw4spl_0.10.2.3
```

- fw4spl-ar-deps: contains the scripts to compile the external libraries used by fw4spl-ar

```
$ cd ~/Dev/Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ar-deps.git fw4spl-ar-deps
$ cd fw4spl-ar-deps
$ git checkout fw4spl_0.10.2.3
```

You must re-edit cmake configuration to add this repository:

```
$ cd ~/Dev/Deps/Build
$ ccmake .
```

Modify *ADDITIONAL_DEPS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps separated by ';'

```
~/Dev/Deps/Src/fw4spl-ext-deps/;~/Dev/Deps/Src/fw4spl-ar-deps/
```

Source

If you want to use fw4spl extension, you need this repositories:

- fw4spl-ext: extension of fw4spl repository, contains additional functionalities and proofs of concept

```
$ cd Dev/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ext.git fw4spl-ext
$ cd fw4spl-ext
$ git checkout fw4spl_0.10.2.3
```

- fw4spl-ar: another extension of fw4spl, contains functionalities for augmented reality (video tracking)

```
$ cd ../../Build
$ ccmake .
```

Modify *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext separated by ';'

```
~/Dev/Src/fw4spl-ext/;~/Dev/Src/fw4spl-ar/
```

2.3.5 Recommended software

The following programs may be helpful for your developments:

- Eclipse CDT
- QtCreator

2.4 Installation for MacOSX

2.4.1 Prerequisites for MacOSX users

If not already installed:

1. Install Xcode
2. Install git
3. Install Python 2.7
4. Install CMake
5. Install Ninja : to use instead of **make**.

For an easy install, you can use the Hombrew project to install missing packages.

```
$ brew install git
$ brew install python
$ brew install cmake
$ brew install ninja
```

2.4.2 FW4SPL installation

FW4SPL works with data separation for source, build and install data. To prepare the development environment:

- Create a development folder (Dev)

```
$ mkdir Dev
```

- **Create the build, source and install folder**
  - Dev/Build
  - Dev/Src

    - Dev/Install

```
$ mkdir Dev/Build Dev/Src Dev/Install
```

To prepare the third party environment:

- Create a third party folder (Deps)

```
$ mkdir Dev/Deps
```

- **Create the build, source and install folder**
    - Dev/Deps/Build
    - Dev/Deps/Src
    - Dev/Deps/Install

```
$ mkdir Dev/Deps/Build Dev/Deps/Src Dev/Deps/Install
```

Build tools

FW4SPL is a CMake project. That means, for each build target there is a CMakeLists that provides build parameters. To configure you project you can use the `cmake` command from the build folder with the sources as arguments:

```
$ ccmake /PATH/TO/fw4spl
```

if you want to use **Ninja** as build to tools, use the option `-G Ninja`, as following:

```
$ ccmake -G Ninja /PATH/TO/fw4spl
```

It is the same process for BinPkgs and FW4SPL sources. It is recommended to use make to compile the deps.

Dependencies

For the third party libraries the following repository have to be cloned in the (Deps) source folder:

- fw4spl-deps: contains the scripts to compile the external libraries used by fw4spl (Boost, VTK, ITK, Qt, . . .)

```
$ cd ~/Dev/Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-deps.git fw4spl-deps
$ cd fw4spl-deps
$ git checkout fw4spl_0.10.2.3
```

To build the dependencies, you must configure the project with cmake into the Build folder

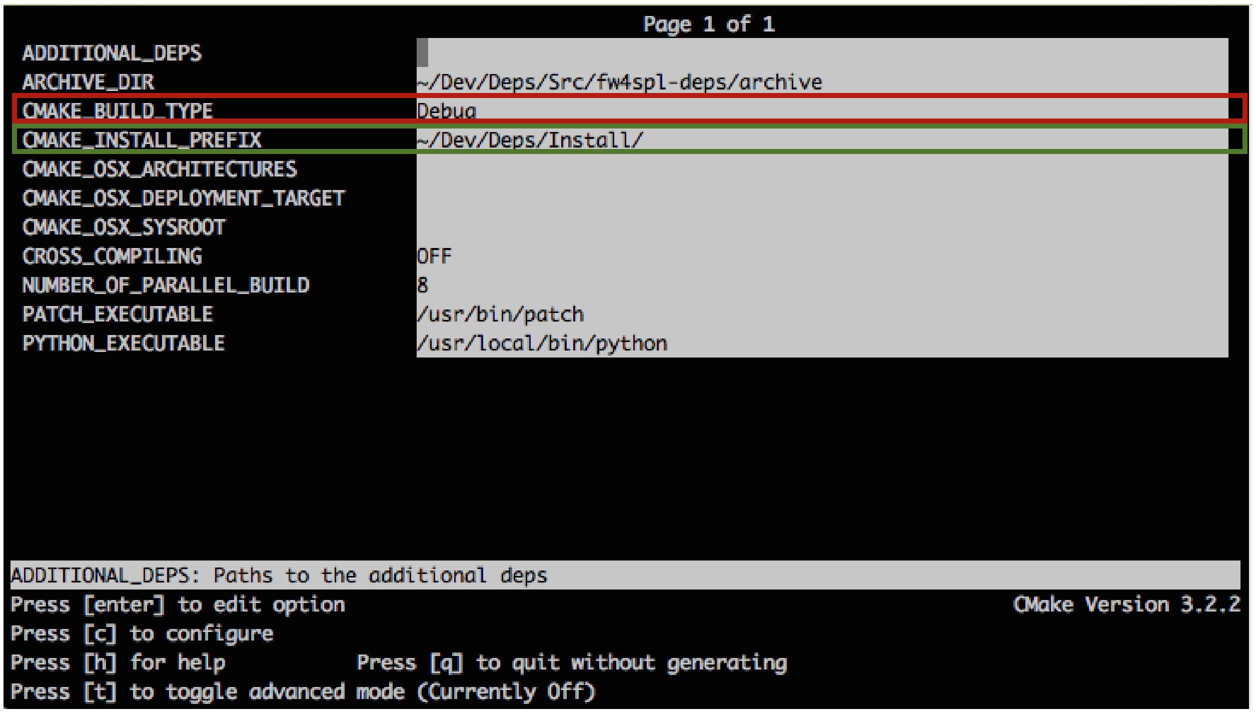
```
$ cd ~/Dev/Deps/Build
$ cmake ../Src/fw4spl-deps -DCMAKE_INSTALL_PREFIX=../Install -DCMAKE_BUILD_TYPE=Debug
```

Or open cmake gui editor, see *Build tools* instructions.

```
$ ccmake ../Src/fw4spl-deps
```

Some CMake variables have to be change:

- *CMAKE_INSTALL_PREFIX*: set the install location.
- *CMAKE_BUILD_TYPE*: set the build type 'Debug' or 'Release'



```
                                 Page 1 of 1
 ADDITIONAL_DEPS
 ARCHIVE_DIR                    ~/Dev/Deps/Src/fw4spl-deps/archive
 CMAKE_BUILD_TYPE               Debug
 CMAKE_INSTALL_PREFIX           ~/Dev/Deps/Install/
 CMAKE_OSX_ARCHITECTURES
 CMAKE_OSX_DEPLOYMENT_TARGET
 CMAKE_OSX_SYSROOT
 CROSS_COMPILING                OFF
 NUMBER_OF_PARALLEL_BUILD       8
 PATCH_EXECUTABLE               /usr/bin/patch
 PYTHON_EXECUTABLE              /usr/local/bin/python







ADDITIONAL_DEPS: Paths to the additional deps
Press [enter] to edit option                                 CMake Version 3.2.2
Press [c] to configure
Press [h] for help           Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Press configure (*[c]*) and generate (*[g]*) makefiles.

Then, compile the FW4SPL dependencies with make

```
$ make all
$ make install_tool
```

Warning: Do NOT use ninja to compile the dependencies, it cause conflict with qt compilation.

Source

For the FW4SPL source code the following repository have to be cloned in the (Dev) source folder:

- fw4spl: main repository, contains the core libraries and bundles.

```
$ cd ~/Dev/Src
$ git clone https://github.com/fw4spl-org/fw4spl.git fw4spl
$ cd fw4spl
$ git checkout fw4spl_0.10.2.3
```

Note: For the source compilation we use `ninja` instead of `make`. But if you prefer to use make, replace all the `ninja` command with `make` and remove `-G Ninja` in the cmake command.

To build fw4spl, you must configure the project with cmake into the Build folder

```
$ cd ~/Dev/Build
$ cmake ../Src/fw4spl -DCMAKE_INSTALL_PREFIX=../Install -DCMAKE_BUILD_TYPE=Debug -DEXTERNAL_LIBRARIES
```

Or open cmake gui editor, see *Build tools* instructions.

```
$ ccmake ../Src/fw4spl -G Ninja
```

Some CMake variables have to be change:

- *CMAKE_INSTALL_PREFIX*: set the install location.
- *EXTERNAL_LIBRARIES*: set the install path of the third part libraries.
- *CMAKE_BUILD_TYPE*: set to Debug or Release

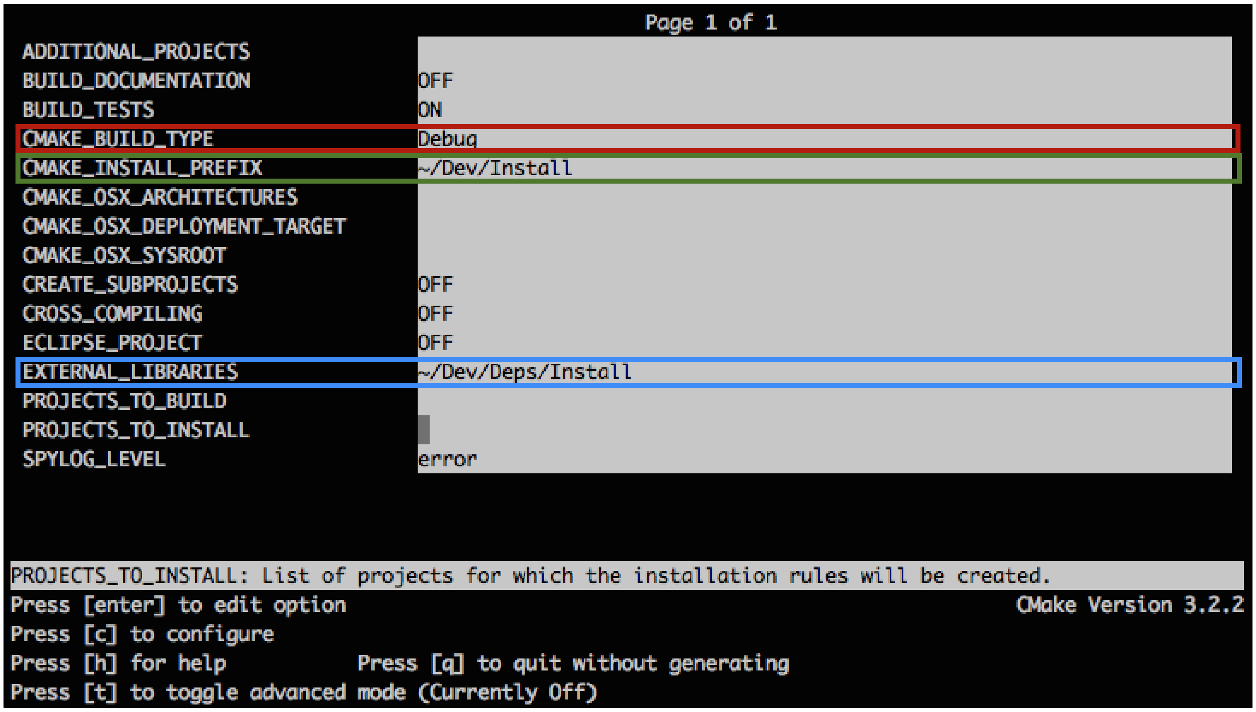
You can re-edit cmake configuration :

```
$ ccmake .
```

- *PROJECT_TO_BUILD* set the name of the application to build (See DevSrcApps)
- *PROJECT_TO_INSTALL* set the name of the application to install

Note:

- If PROJECT_TO_BUILD is empty, all application will be compiled
- If PROJECT_TO_INSTALL is empty, no aplication will be installed



```
                                                  Page 1 of 1
 ADDITIONAL_PROJECTS
 BUILD_DOCUMENTATION              OFF
 BUILD_TESTS                      ON
 CMAKE_BUILD_TYPE                 Debug
 CMAKE_INSTALL_PREFIX             ~/Dev/Install
 CMAKE_OSX_ARCHITECTURES
 CMAKE_OSX_DEPLOYMENT_TARGET
 CMAKE_OSX_SYSROOT
 CREATE_SUBPROJECTS               OFF
 CROSS_COMPILING                  OFF
 ECLIPSE_PROJECT                  OFF
 EXTERNAL_LIBRARIES               ~/Dev/Deps/Install
 PROJECTS_TO_BUILD
 PROJECTS_TO_INSTALL
 SPYLOG_LEVEL                     error



PROJECTS_TO_INSTALL: List of projects for which the installation rules will be created.
Press [enter] to edit option                                                 CMake Version 3.2.2
Press [c] to configure
Press [h] for help             Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Press configure (**[c]**) and generate (**[g]**) makefiles.

Then, build dependencies with ninja.

```
$ ninja all
```

2.4.3 Launch an application

To build a specific or several applications the CMake argument `PROJECTS_TO_BUILD` can be set. Use `;` so separate each application name.

After an successful compilation the application can be launched with the launcher program from a terminal. Therefore the profile.xml of the application in the build folder has to be passed as argument to the launcher:

```
$ bin/launcher Bundles/MyApplication_Version/profile.xml
```

Example:

```
$ cd ~/Dev/Build
$ bin/launcher Bundles/VRRender_0-9/profile.xml
```

Note: To generate the projects in release, change CMake argument `CMAKE_BUILD_TYPE` to `Release` for fw4spl and fw4spl-deps

Warning: Do NOT compile debug and release in the same Build and Install folder

2.4.4 Extension

fw4spl has two extension repositories:

- fw4spl-ext: contains additional functionalities and proofs of concept
- fw4spl-ar: contains functionalities for augmented reality (video tracking, calibration)

Dependencies

If you want to use this extension, you need to clone the deps repositories:

- fw4spl-ext-deps: contains the scripts to compile the external libraries used by fw4spl-ext

```
$ cd ~/Dev/Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ext-deps.git fw4spl-ext-deps
$ cd fw4spl-ext-deps
$ git checkout fw4spl_0.10.2.3
```

- fw4spl-ar-deps: contains the scripts to compile the external libraries used by fw4spl-ar

```
$ cd ~/Dev/Deps/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ar-deps.git fw4spl-ar-deps
$ cd fw4spl-ar-deps
$ git checkout fw4spl_0.10.2.3
```

You must re-edit cmake configuration to add this repository:

```
$ cd ~/Dev/Deps/Build
$ ccmake .
```

Modify *ADDITIONAL_DEPS*: set the source location of fw4spl-ar-deps and fw4spl-ext-deps separated by ';'

```
~/Dev/Deps/Src/fw4spl-ext-deps/;~/Dev/Deps/Src/fw4spl-ar-deps/
```

Source

If you want to use fw4spl extension, you need this repositories:

- fw4spl-ext: extension of fw4spl repository, contains additional functionalities and proofs of concept

```
$ cd Dev/Src
$ git clone https://github.com/fw4spl-org/fw4spl-ext.git fw4spl-ext
$ cd fw4spl-ext
$ git checkout fw4spl_0.10.2.3
```

- fw4spl-ar: another extension of fw4spl, contains functionalities for augmented reality (video tracking)

```
$ cd ../../Build
$ ccmake .
```

Modify *ADDITIONAL_PROJECTS*: set the source location of fw4spl-ar and fw4spl-ext separated by ';'

```
~/Dev/Src/fw4spl-ext/;~/Dev/Src/fw4spl-ar/
```

2.4.5 Recommended softwares

The following programs may be helpful for your developments:

- **IDE:**
  - Qt creator
  - Eclipse CDT.
- **Versioning tools:**
  - SourceTree

CHAPTER 3

Software Architecture Description (SAD)

3.1 General

3.1.1 Introduction

The framework FW4SPL (FrameWork for Software Production) is an open-source framework, developed by IRCAD (research institute against cancer and disease). The principle of FW4SPL is the fast and easy creation of applications, mainly in the medical field. Therefore it provides features like digital image processing in 2D and 3D, visualization or simulation of medical interactions. To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

FW4SPL is built on component-based architecture composed of C++ libraries. The three main concepts of the architecture, explained in the following sections, are:

- object-service concept
- component approach
- signal-slot communication

The framework is multi-platform and runs under Windows, Linux and MacOS. The programming language of the framework is C++. This document will introduce the general architecture of FW4SPL.

3.1.2 Annexes

- *Srclib list:* this document lists all libraries with a brief description.
- *Object list:* this document lists all data with a brief description.
- *Service list:* this document lists all services and bundles with a brief description.
- *Third party:* this document contains a description of libraries used to support this architecture and its functions.
- *OSR diagram:* this document introduces how to represent an application configuration as a diagram.

3.2 Object-Service concept

3.2.1 Introduction

Inside the Object Oriented Programming (OOP) paradigm, an object is an instance of a class which contains all its data and methods (such as reading, writing, visualizations, image analysis, etc.). This philosophy works well, provided that the classes do not change with time. However, in this situation, the maintenance of source code is difficult.

In order to make this maintenance easier, FW4SPL architecture relies on an Object-Service paradigm where data and their methods are separated into different code units.

3.2.2 Object

Objects represent data used in the framework. They can be simple (boolean, integer, string, etc.) or advanced structures (image, mesh, video, patient, etc.) without depending on the input data format. For example, an input image could have one of several formats such as Jpeg or Dicom but the FW4SPl object will be the same.

Moreover, these object classes contain only data features and their corresponding getter/setter methods.

For instance, the `Image` object:

- contains a buffer pointer, a buffer size, the image's dimension and origin,
- has public setter/getter methods to access these members,
- does not have methods such as reading or writing a buffer

The `fwData` library contains the standard simple and advanced data. It is the FW4SPL's main data library. There is also the `fwMedData` library which contains several structures to store medical data. A data list with a brief description is available in the appendixes.

Creating data

New data must be created as described below.

In the header file (MyData.hpp):

```
class MyData : public ::fwData::Object
{

public :
    fwCoreClassDefinitionsWithFactoryMacro( (MyData)(::fwData::Object),
        (()), ::fwData::factory::New< MyData >) ;


    // Private constructor, required for data factory
    MyData(::fwData::Object::Key key);

    /// Destructor, required for all data
    virtual ~MyData();

    /// Defines shallow copy, required for all data
    void shallowCopy( const Object::csptr& _source );

    /// Defines deep copy, required for all data
    void cachedDeepCopy(const Object::csptr& _source,
                            DeepCopyCacheType &cache);

};
```

In the source file (MyData.cpp), this line must also be added to declare `MyClass` as data of the framework architecture :

```
fwDataRegisterMacro( MyData );
```

3.2.3 Service

A service represents a functionality which uses or modifies data. A service is always associated with a data. For example, image data can have a reader service, a writer service, a visualization service or a processing operator.

Service type

Some service categories exist in FW4SPL. These categories are called *service types* and are represented by an abstract class. The basic service types are:

- `io::IReader`: base interface for reader services.
- `io::IWriter`: base interface for writer services.
- `fwGui::IActionSrv`: base interface to manage action from a button or a menu in the GUI.
- `gui::editor::IEditor`: base interface to create new widget in the GUI.
- `fwRender::IRender`: base interface to create new visualization widgets in the GUI.
- `fwServices::IController`: Does nothing in particular but can be considered as a default service type to be implemented by unclassified services.

All services require a type association and must inherit from an abstract type service.

Service methods

Several methods exist to manipulate a service. The main methods are: `configure`, `start`, `stop`, and `update`.

- `configure`: parses the service parameters and analyzes its configuration. For example, this method is used to configure an image file path on the file system for an image reader service.
- `start`: initializes and launches the service (be careful, starting and instantiating a service is not the same thing. For example, for a visualization service, the `start` method instantiates all GUI widgets necessary to visualize the data but the service itself is instantiated before.).
- `stop`: stops the service. For example, for a visualization service, this method detaches and destroys all GUI widgets previously instantiated earlier in the `start` method.
- `update` method is called to perform an action on the data associated with the service. For example, for an image reader service, the service reads the image, converts it and loads it into the associated data.

These methods are mandatory, but can be empty. This is because some services do not need a start/stop process, an update process or to listen to object modifications.

Service states

These methods must follow a calling sequence. For example, it is not possible to stop a service before starting it. To secure the process, a state machine has been implemented to control the calling sequence.

The calling sequence to manage a service is:

```
MyData::sptr myData = MyData::New();
MyService::sptr mySrv = MyService::New();
mySrv->setObject(myData);

mySrv->setConfiguration( ... ); // set parameters
mySrv->configure(); // check parameters
mySrv->start(); // start the service
```

```
mySrv->update(); // update the service
mySrv->stop(); // stop the service
```

Create a service

A new service must be created as described below.

In the header file (MyService.hpp):

```
class MyService : public AbstractServiceType
{
public:

    // Macro to define few important parameters/functions
    // used by the architecture
    fwCoreServiceClassDefinitionsMacro((MyService)(AbstractServiceType));

    // Service constructor
    MyService() throw() ;

    // Service destructor.
    virtual ~MyService() throw() ;

protected:

    // To configure the service
    void configuring() throw(fwTools::Failed);

    // To start the service
    void starting() throw(::fwTools::Failed);

    // To stop the service
    void stopping() throw(::fwTools::Failed);

    // To update the service
    void updating() throw(::fwTools::Failed);
};
```

In the source file (MyService.cpp), this line must be also added to declare `MyService` as a service of the framework architecture:

```
fwServicesRegisterMacro( AbstractServiceType, MyService, MyData );
```

Note: When a new service is created, the following functions must be overloaded from IService class : `configuring`, `starting`, `stopping` and `updating`. The top level functions from IService class check the service state before any call to the redefined method.

3.2.4 Object and service factories

To instantiate an object or a service, the architecture requires the use of a factory system. In class-based programming, the factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating classes without specifying the exact class that will be created. This is done by creating classes via a factory method, which is either specified in an interface (abstract class) and implemented in child classes (concrete classes) or implemented in a base class (optionally as a template method), which can be overridden when inherited in derivative classes; rather than by a constructor.[#]_

Object factory

The `fwData` library has a factory to register and create all objects. The registration is managed by two macros:

```
// in .hpp file
fwCoreClassDefinitionsWithFactoryMacro( (MyData)(::fwData::Object),
    (()), ::fwData::factory::New< MyData >);

// in .cpp file
fwDataRegisterMacro( MyData );
```

Then, there are only two ways to build data in the framework:

```
// Direct creation
MyData::sptr obj = MyData::New();

// Factory creation (here obj is an object of type
// MyData, it is possible to cast it)
::fwData::Object::sptr obj = ::fwData::factory::New("MyData");
```

Service factory

The `fwService` library has a factory to register and create all services. The registration is managed by two macros:

```
// in .hpp file
fwCoreServiceClassDefinitionsMacro ((MyService)(AbstractServiceType));

// in .cpp file
fwServicesRegisterMacro( AbstractServiceType, MyService, MyData );
```

Then, there is only one way to build a service in the framework:

```
::fwServices::registry::ServiceFactory::sptr srvFactory
        = ::fwServices::registry::ServiceFactory::getDefault();

// Factory creation (here srv is a service of type MyService,
// it is possible to cast it)
::fwServices::IService::sptr srv = srvFactory->create("MyService");
```

3.2.5 Object-Service registry

The FW4SPL architecture is standardized thanks to:

- Abstract classes `fwData::Object` and `fwService::IService`.
- The two factory systems.

In an application, one of the problems is managing the life cycle of a large number of object instances and their services. This problem is solved by the class `fwServices::registry::ObjectService` which maintains the relationship between objects and services. This class concept is very simple :

```
// OSR is a singleton
class ObjectService
{
  // relation map beetwen an object and his associated services
  map < Object *, vec < IService > > osr;

  // Associates a service to an object
  // manages in the function the association: srv->setObject(obj);
  void registerService ( Object * obj , IService * srv );

  // Dissociates a service to his object
  void unregisterService ( IService * srv );

  // ...
}

// Some helpers exist : below, add method is used to combine
// factory system with service registration
::fwServices::IService::sptr add(::fwData::Object::sptr obj,
        std::string serviceType, std::string _implementationId)
```

This registry manages the object-service relationships and guarantees the non-destruction of an object while some services are still working on it.

3.2.6 Object-Service concept example

To conclude, the generic object-service concept is illustrated with this example:

```
// Create an object
::fwData::Object::sptr obj = ::fwData::factory::New("::fwData::Image");

// Create a reader and a view for this object
::fwServices::IService::sptr reader
    = ::fwServices::add(obj, "::io::IReader", "MyCustomImageReader");
::fwServices::IService::sptr view
    = ::fwServices::add(obj, "::fwRender::IRender", "MyCustomImageView");

// Configure and start services
reader->setConfiguration ( /* ... */ );
reader->configure();
reader->start();

view->setConfiguration ( /* ... */ );
view->configure();
view->start();

// Execute services
reader->update(); // Read image on filesystem
view->update(); // Refresh visualization with the new image buffer

// Stop services
reader->stop();
view->stop();

// Destroy services
::fwServices::registry::ObjectService::unregisterService(reader);
::fwServices::registry::ObjectService::unregisterService(view);
```

This example shows the code to create a small application to read an image and visualize it. You can easily transform this code to build an application which reads and displays a 3D mesh by changing object and services implementation strings only.

3.3 Signal-slot communication

3.3.1 Overview

“Signals and slots” is a language construct introduced in Qt [1] for communication between objects.

The concept is that objects and services(explained in 2.3) can send signals containing event information which can be received by other services using special functions known as slots.

3.3.2 FW4SPL implementation

In the FW4SPL architecture, the library `fwCom` provides a set of tools dedicated to communication. These communications are based on the signal and slot concept.

`fwCom` provides the following features :

- function and method wrapping
- direct slot calling
- asynchronous slot calling
- ability to work with multiple threads
- auto-disconnection of slot and signals
- arguments loss between slots and signals

3.3.3 Slots

Slots are wrappers for functions and class methods that can be attached to a `fwThread::Worker`. The purpose of this class is to provide synchronous and asynchronous mechanisms for method and function calling.

Slots have a common base class : SlotBase. This allows the storage of them in the same container. Slots are designed such that they can be called, even where only the argument type is known.

Examples :

A slot wrapping the function sum, which is a function with the signature int (int, int) :

```
::fwCom::Slot< int (int, int) >::sptr slotSum = ::fwCom::newSlot( &sum );
```

A slot wrapping the function start with signature void() of the object `a` which class type is `A` :

```
::fwCom::Slot< void() >::sptr slotStart = ::fwCom::newSlot(&A::start, &a);
```

Execution of the slots using the run method :

```
slotSum->run(40,2);
slotStart->run();
```

Execution of the slots using the method call, which returns the result of the execution :

[1] http://wiki.qt.io/Qt_signal-slot_quick_start

```
int result = slotSum->call(40,2);
slotStart->call();
```

A slot declaration and execution, through a SlotBase :

```
::fwCom::Slot< size_t (std::string) > slotLen
        = ::fwCom::Slot< size_t (std::string) >::New( &len );
::fwCom::SlotBase::sptr slotBaseLen = slotLen;
::fwCom::SlotBase::sptr slotBaseSum = slotSum;
slotBaseSum->run(40,2);
slotBaseSum->run<int, int>(40,2);

// This one needs the  explicit argument type
slotBaseLen->run<std::string>("R2D2");
result = slotBaseSum->call<int>(40,2);
result = slotBaseSum->call<int, int, int>(40,2);
result = slotBaseLen->call<size_t, std::string>("R2D2");
```

3.3.4 Signals

Signals allow to perform grouped calls on slots. For this purpose, a signal class provides a mechanism to connect slots.

Examples:

The following instruction declares a signal with a void signature.

```
::fwCom::Signal< void() >::sptr sig = ::fwCom::Signal< void() >::New();
```

The connection between a signal and a slot of the same information type:

```
sig->connect(slotStart);
```

The following instruction will trigger the execution of all slots connected to this signal:

```
sig->emit();
```

It is possible to connect multiple slots with the same information type to the same signal and trigger their simultaneous execution.

Signals can take several arguments as a signature which will trigger their connected slots by passing the right arguments.

In the following example a signal is declared of type void(int, int). The signal is connected to two different types of slot, void (int) and int (int, int).

```
using namespace fwCom;
Signal< void(int, int) >::sptr sig2 = Signal< void(int, int) >::New();
Slot< int(int, int) >::sptr    slot1 = Slot< int(int, int) >::New(...);
Slot< void(int) >::sptr        slot2 = Slot< void(int) >::New(...);

sig2->connect(slot1);
sig2->connect(slot2);

sig2->emit(21, 42);
```

Here 2 points need to be highlighted :

- A signal cannot return a value. Consequently their return type is void. Thus, the return value of a slot, triggered by a signal, equally cannot be retrieved.

- To successfully trigger a slot using a signal, the minimum requirement as to the number of arguments or fitting argument types has to be given by the signal. In the last example the slot slot2 only requires one argument of type int, but the signal is emitting two arguments of type int. Because the signal signature fulfills the slot's argument number and argument type, the signal can successfully trigger the slot slot2. The slot slot2 takes the first emitted argument which fits its parameter (here 21, the second argument is ignored).

Disconnection

The disconnect method is called between one signal and one slot, to stop their existing connection. A disconnection assumes a signal slot connection. Once a signal slot connection is disconnected, it cannot be triggered by this signal. Both connection and disconnection of a signal slot connection can be done at any time.

```
sig2->disconnect(slot1);
sig2->emit(21, 42); // do not trigger slot1 anymore
```

The instructions above will cause the execution of slot2. Due to the disconnection between sig2 and slot1, the slot slot1 is not triggered by sig2.

Connection handling

The connection between a slot and a signal returns a connection handler:

```
::fwCom::Connection connection = signal->connect(slot);
```

Each connection handler provides a mechanism which allows a signal slot connection to be disabled temporarily. The slot stays connected to the signal, but it will not be triggered while the connection is blocked :

```
::fwCom::Connection::Blocker lock(connection);
signal->emit();
// 'slot' will not be executed while 'lock' is alive or until lock is
// reset
```

Connection handlers can also be used to disconnect a slot and a signal :

```
connection.disconnect();
// slot is not connected anymore
```

Auto-disconnection

Slots and signals can handle an automatic disconnection :

- on slot destruction : every signal slot connection to this slot will be destroyed
- on signal destruction : every slot connection to the signal will be destroyed

All related connection handlers will be invalidated when an automatic disconnection occurs.

3.3.5 Manage slots or signals in a class

The library `fwCom` provides two helper classes to manage signals or slots in a structure.

HasSlots

The class `HasSlots` offers mapping between a key (string defining the slot name) and a slot. `HasSlots` allows the management of many slots using a map. To use this helper in a class, the class must inherit from `HasSlots` and must register the slots in the constructor:

```
struct ThisClassHasSlots : public HasSlots
{
  typedef Slot< int()> GetValueSlotType;

  ThisClassHasSlots()
  {
      newSlot("sum", &SlotsTestHasSlots::getValue, this);
  }

  int sum(int a, int b)
  {
      return a+b;
  }

  int getValue()
  {
      return 4;
  }
};
```

Then, slots can be used as below :

```
ThisClassHasSlots obj;
obj.slot("sum")->call<int>(5,9);
obj.slot< ThisClassHasSlots::GetValueSlotType >("getValue")->call();
```

HasSignals

The class `HasSignals` provides mapping between a key (string defining the signal name) and a signal. `HasSignals` allows the management of many signals using a map, similar to `HasSlots`. To use this helper in a class, the class must inherit from `HasSignals` as seen below and must register signals in the constructor:

```
struct ThisClassHasSignals : public HasSignals
{
  typedef ::fwCom::Signal< void()> SignalType;

  ThisClassHasSignals()
  {
      newSignal< SignalType >("sig");
  }
};
```

Then, signals can be used as below:

```
ThisClassHasSignals obj;
Slot< void()>::sptr slot = ::fwCom::newSlot(&anyFunction)
obj.signal("sig")->connect( slot );
obj.signal< SignalsTestHasSignals::SignalType >("sig")->emit();
obj.signal("sig")->disconnect( slot );
```

3.3.6 Signals and slots used in objects and services

Slots are used in both objects and services, whereas signals are only used in services. The abstract class `fwData::Object` inherits from the `HasSignals` class as a basis to use signals :

```
class Object : public ::fwCom::HasSignals
{
  /// Key in m_signals map of signal m_sigObjectModified
  static const ::fwCom::Signals::SignalKeyType s_MODIFIED_SIG;
  //...

  /// Type of signal m_sigObjectModified
  typedef ::fwCom::Signal< void ( CSPTR( ::fwServices::ObjectMsg ) ) >
                ObjectModifiedSignalType;

  /// Signal that emits an ObjectMsg when an object is modified
  ObjectModifiedSignalType::sptr m_sigObjectModified;

  Object()
  {
      m_sigObjectModified = newSignal< ObjectModifiedSignalType >(s_MODIFIED_SIG);
      //...
  }
}
```

Moreover the abstract class `fwService::IService` inherits from the `HasSlots` class and the `HasSignals` class, as a basis to communicate through signals and slots. Actually, the methods `start()`, `stop()`, `swap()` and `update()` are all slots. Here is an extract with `update()`:

```
class IService : public ::fwCom::HasSlots, public ::fwCom::HasSignals
{
  typedef ::boost::shared_future< void > SharedFutureType;

  /// Key in m_slots map of slot m_slotUpdate
  static const ::fwCom::Slots::SlotKeyType s_UPDATE_SLOT;

  /// Type of signal m_slotUpdate
  typedef ::fwCom::Slot<SharedFutureType()> UpdateSlotType;

  /// Slot to call update method
  UpdateSlotType::sptr m_slotUpdate;

  IService()
  {
      //...
      m_slotUpdate = newSlot( s_UPDATE_SLOT, &IService::update, this ) ;
      //...
  }

  //...
}
```

To automatically connect object signals and service slots, it is possible to override the method `IService::getObjSrvConnections()`. Please note that to be effective the attribute "autoconnect" of the service must be set to "yes" in the xml configuration (see *App-config*). The default implementation of this method connect the `s_MODIFIED_SIG` object signal to the `s_UPDATE_SLOT` slot.

```
IService::KeyConnectionsType IService::getObjSrvConnections() const
{
    KeyConnectionsType connections;
    connections.push_back( std::make_pair( ::fwData::Object::s_MODIFIED_SIG, s_UPDATE_SLOT ) );
    return connections;
}
```

3.3.7 Object signals

Objects have signals that can be used to inform of modifications. The base class `::fwData::Object` has the following signals available.

Objects	Available messages
Object	{`modified`, `addedFields`, `changedFields`, `removedFields`}

Thus all objects in FW4SPL can use the previous signals. Some object classes define extra signals.

Objects	Available messages
Composite	{`addedObjects`, `changedObjects`, `removedObjects`}
Graph	{`updated`}
Image	{`bufferModified`, `landmarkAdded`, `landmarkRemoved`, `landmarkDisplayed`, `distanceAdded`, `distanceRemoved`, `distanceDisplayed`, `sliceIndexModified`, `sliceTypeModified`, `visibilityModified`, `transparencyModified`}
Mesh	{`vertexModified`, `pointColorsModified`, `cellColorsModified`, `pointNormalsModified`, `cellNormalsModified`, `pointTexCoordsModified`, `cellTexCoordsModified`}
ModelSeries	{`reconstructionsAdded`, `reconstructionsRemoved`}
PlaneList	{`planeAdded`, `planeRemoved`, `visibilityModified`}
Plane	{`selected`}
PointList	{`pointAdded`, `pointRemoved`}
Reconstruction	{`meshModified`, `visibilityModified`}
ResectionDB	{`resectionAdded`, `safePartAdded`}
Resection	{`reconstructionAdded`, `pointTexCoordsModified`}
Vector	{`addedObjects`, `removedObjects`}
...	...

3.3.8 Proxy

The class `::fwServices::registry::Proxy` is a communication element and singleton in the architecture. It defines a proxy for signal/slot connections. The proxy concept is used to declare communication channels: all signals registered in a proxy's channel are connected to all slots registered in the same channel. This concept is useful to create multiple connections or when the slots/signals have not yet been created (possible in dynamic programs).

The following shows an example where one signal is connected to several slots:

```
const std::string CHANNEL = "myChannel";

::fwServices::registry::Proxy::sptr proxy
    = ::fwServices::registry::Proxy::getDefault();

::fwCom::Signal< void() >::sptr sig = ::fwCom::Signal< void() >::New();
```

```
::fwCom::Slot< void() >::sptr slot1 = ::fwCom::newSlot( &myFunc1 );
::fwCom::Slot< void() >::sptr slot2 = ::fwCom::newSlot( &myFunc2 );
::fwCom::Slot< void() >::sptr slot3 = ::fwCom::newSlot( &myFunc3 );

proxy->connect(CHANNEL, sig);

proxy->connect(CHANNEL, slot1);
proxy->connect(CHANNEL, slot2);
proxy->connect(CHANNEL, slot3);

sig->emit(); // All slots are called
```

3.4 App-config

3.4.1 Dynamic program with factories

As shown in the *Object-Service concept example*, it is easy to change an application's behaviour by simply changing the appropriate data and services. For example changing an image visualisation application to a 3D model visualisation application. Unfourtunely, this is limited to applications based on one service and one data, and thus would impossible to apply on applications containing multiple services and object.

To overcome this, the FW4SPL architecture provides a dynamic management of configurations to allow the use of multiple objects and services.

The xml configuration for an application is defined with the extension `::fwServices::registry::AppConfig`.

3.4.2 Dynamic program with application configuration

In the `fwServices` library, an application configuration parser allows to parse XML files and creates and manages objects, services and communications.

```
// The parser
void main (int argc , char * argv [])
{
    string xmlAppConfigPath = argv [1];

    ::fwServices::AppConfigManager::sptr acm
            = ::fwServices::AppConfigManager::New();

    acm->setConfig(xmlAppConfigPath);
    acm->create(); // Creates objects and services from config.
    acm->start(); // Starts services specified in config.
    acm->update(); // Updates services specified in config.

    acm->stop(); // Stops services specified in config.
    acm->destroy(); // Destroy all services and then data.
}
```

The following part corresponds to the configuration XML file of the previous *Object-Service concept example*.

```
<object uid="image" type ="::fwData::MyData">

    <service uid="frame" impl="DefaultFrame" type="IFrame" >
```

```
        <!-- service configuration -->
    </service>

    <service uid="view" impl="MyCustomImageView"
             type="::fwRender::IRender" >
        <!-- service configuration -->
    </service>

    <service uid="reader" impl="MyCustomImageReader"
             type="::io::IReader" >
        <!-- service configuration -->
    </service>

    <!-- view listen now image modification -->
    <connect>
        <signal>image/objectModified</signal>
        <slot>view/receive</slot>
    </connect>

    <start uid="frame" />
    <start uid="view"/>
    <start uid="reader"/>

    <!-- Read the image on filesystem and notify
         the view to refresh is content -->
    <update uid ="reader"/>

</ object >
```

This simple example shows how it is possible to build an application with several objects and services using only a program and its configurations files.

3.4.3 Example

```
<extension implements="::fwServices::registry::AppConfig">
    <id>myAppConfigId</id>
    <parameters>
        <param name="appName" default="my Application" />
        <param name="appIconPath" />
    </parameters>
    <desc>Image Viewer</desc>
    <config>

        <object type="::fwData::Composite">

            <!--
                Description service of the GUI:
                The ::gui::frame::SDefaultFrame service automatically positions the various
                containers in the application main window.
                Here, it declares a container for the 3D rendering service.
            -->
            <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                <gui>
                    <frame>
                        <name>${appName}</name>
                        <icon>${appIconPath}</icon>
                        <minSize width="800" height="600" />
                    </frame>
                </gui>
                <registry>
                    <!-- Associate the container for the rendering service. -->
                    <view sid="myRendering" />
                </registry>
            </service>

            <item key="myImage">
                <object uid="myImageUid" type="::fwData::Image">
                    <!--
                        Reading service:
                        The <file> tag defines the path of the image to load. Here, it is a relative
                        path from the repository in which you launch the application.
                    -->
                    <service uid="myReaderPathFile" impl="::ioVTK::SImageReader">
                        <file>./TutoData/patient1.vtk</file>
                    </service>

                    <!--
                        Visualization service of a 3D medical image:
                        This service will render the 3D image.
                    -->
                    <service uid="myRendering" impl="::vtkSimpleNegato::SRenderer" />
                </object>
            </item>

            <!--
                Definition of the starting order of the different services:
                The frame defines the 3D scene container, so it must be started first.
                The services will be stopped the reverse order compared to the starting one.
            -->
            <start uid="myFrame" />
            <start uid="myReaderPathFile" />
            <start uid="myRendering" />

            <!--
                Definition of the service to update:
                The reading service load the data on the update.
                The render update must be called after the reading of the image.
            -->
            <update uid="myReaderPathFile" />
            <update uid="myRendering" />

        </object>

    </config>
</extension>
```

id

The id is the configuration identifier, and is thus unique to each configuration.

parameters (optional)

The parameters is a list of the parameters used by the configuration.

param: defines the parameter

> **name:** parameter name, used as `${paramName}` in the configuration. It will be replaced by the string defined by the service, activity or application that launchs the configuration.
>
> **default (optional):** default value for the parameter, it is used if the value is not given by the config launcher.

desc (optional)

The description of the application.

config

The config tag includes the services and objects to launch.

object

It defines an object of the AppConfig. We usually use a ::fwData::Composite in order to add sub-objects. An object can contain a list of services. Some object objects can have a specific configuration : ::fwData::TransformationMatrix3D, ::fwData::Float, ::fwData::List, ...

> **uid (optional):** Unique identifier of the object (::fwTools::fwID). If it is not defined, it will be automatically generated.
>
> **type:** Object type (ex: `::fwData::Image`, `::fwData::Composite`)
>
> **src (optional, "new" by default)** Defines if the object should be created (`new`) or if it already exists in the application (`ref`). In the last case, the uid must be the same as the first declaration of this object (with `new`).

service: It represents a service working on the object

> **uid (optional):** Unique identifier of the service. If it is not defined, it will be automatically generated.
>
> **impl:** Service implementation type (ex: `::ioVTK::SImageReader`)
>
> **type (optional):** Service type (ex: `::io::IReader`)
>
> **autoConnect (optional, "no" by default):** Defines if the service receives the signals of the working object
>
> **worker (optional):** Allows to run the service in another worker (see *Multithreading*)

Some services needs a specific configuration, it is usually described in the doxygen of the method `configuring()`.

matrix (optional): It works only for `::fwData::TransformationMatrix3D` objects. It defines the value of the matrix.

```xml
<object uid="matrix" type="::fwData::TransformationMatrix3D">
    <matrix>
    <![CDATA[
        1   0   0   0
        0   1   0   0
        0   0   1   0
        0   0   0   1
    ]]>
    </matrix>
</object>
```

value (optional): Only these objects contain this tag : `::fwData::Boolean`, `::fwData::Integer`, `::fwData::Float` and `::fwData::String`. It allows to define the value of the object.

```
<object type="::fwData::Integer">
    <value>42</value>
</object>
```

item (optional): It defines a sub-object of a composite. It can only be used if the parent object is a `::fwData::Composite`.

key: key of the object in the composite

object: The 'item' tag can only contain 'object' tags that represents the composite sub-object

```
<item key="myImage">
    <object uid="myImageUid" type="::fwData::Image" />
</item>
```

colors (optional): Only `::fwData::TransferFunction` contains this tag. It allows to fill the transfer function values.

```
<object type="::fwData::TransferFunction">
    <colors>
        <step color="#ff0000ff" value="1" />
        <step color="#ffff00ff" value="500" />
        <step color="#00ff00ff" value="1000" />
        <step color="#00ffffff" value="1500" />
        <step color="#0000ffff" value="2000" />
        <step color="#000000ff" value="4000" />
    </colors>
</object>
```

connect (optional): allows to connect a signal to one or more slot(s). The signal and slots must be compatible.

```
<connect>
    <signal>object_uid/signal_name</signal>
    <slot>service_uid/slot_name</slot>
</connect>
```

proxy (optional): Allows to connect one or more signal(s) to one or more slot(s). The signals and slots must be compatible.

channel: Name of the channel use for the proxy.

```
<proxy channel="myChannel">
    <signal>object_uid/signal_name</signal>
    <slot>service_uid/slot_name</slot>
</proxy>
```

start: defines the service to start when the AppConfig is launched. The services will be automatically stopped in the reverse order when the AppConfig is stopped.

```
<start uid="service_uid" />
```

update: defines the service to update when the AppConfig is launched.

```
<update uid="service_uid" />
```

3.5 Activities

An activity is defined by the extension `::fwActivities::registry::Activities`. It is used to launch an *AppConfig* with the selected data, it will create a new data `::fwMedData::ActivitySeries` that inherits from a fwMedData::Series.

The service ::activities::action::SActivityLauncher allows to launch an activity. Its role is to create the specific Activity associated with the selected data.

This action should be followed by the service guiQt::editor::DynamicView : this service listens the action signals and launchs the activity in a new tab.

- **::activities::action::SActivityLauncher** uses the selected data to generate the activity.
- **::guiQt::editor::DynamicView** displays the activity in the application.
- **::fwData::Vector** contains the set of selected data .

3.5.1 Activity series

The ::fwMedData::ActivitySeries has a ::fwData::Composite that contains all the data required by the activity.

```
class FWMEDDATA_CLASS_API ActivitySeries : public ::fwMedData::Series
{
public:

    /// Constructor
    FWMEDDATA_API ActivitySeries();

    /// Destructor
    FWMEDDATA_API virtual ~ActivitySeries();

    /// Defines shallow copy
    FWMEDDATA_API void shallowCopy( const ::fwData::Object::csptr &_source );

    /// Defines deep copy
    FWMEDDATA_API void cachedDeepCopy( const ::fwData::Object::csptr &_source, DeepCopyCacheType &cac

    /**
     * @brief Data container
     * @{ */
    ::fwData::Composite::sptr getData () const;
    void setData(const ::fwData::Composite::sptr & val);
    /**  @} */

    /**
     * @brief Activity configuration identifier
     * @{ */
    const std::string &getActivityConfigId () const;
    void setActivityConfigId (const std::string &val);
    /**  @} */

protected:

    /// Activity configuration identifier
    ConfigIdType m_activityConfigId;

    /// Data container
    ::fwData::Composite::sptr m_data;
};
```

3.5.2 Example

```
<extension implements="::fwActivities::registry::Activities">
   <id>myActivityId</id>
   <title>3D Visu</title>
   <desc>Activity description ...</desc>
   <icon>Bundles/media_0-1/icons/icon-3D.png</icon>
   <requirements>
       <requirement name="param1" type="::fwData::Image" /> <!-- defaults : minOccurs = 1, maxOccurs
       <requirement name="param2" type="::fwData::Mesh" maxOccurs="3" container="composite">
           <key>Item1</key>
           <key>Item2</key>
           <key>Item3</key>
       </requirement>
       <requirement name="imageSeries" type="::fwMedData::ImageSeries" minOccurs="0" maxOccurs="2" />
       <requirement name="modelSeries" type="::fwMedData::ModelSeries" minOccurs="1" maxOccurs="1" />
       <!-- ...-->
   </requirements>
   <builder>::fwActivities::builder::ActivitySeries</builder>
   <validator>::fwActivities::validator::ImageProperties</validator><!-- pour fw4spl_0.9.2 -->
   <appConfig id="myAppConfigId">
       <parameters>
           <parameter replace="registeredImageUid" by="@values.param1" />
           <parameter replace="orientation" by="frontal" />
           <!-- ...-->
       </parameters>
   </appConfig>
</extension>
```

The activity parameters are (in the following order):

id

The activity unique identifier.

title

The activity title that will be displayed on the tab.

desc

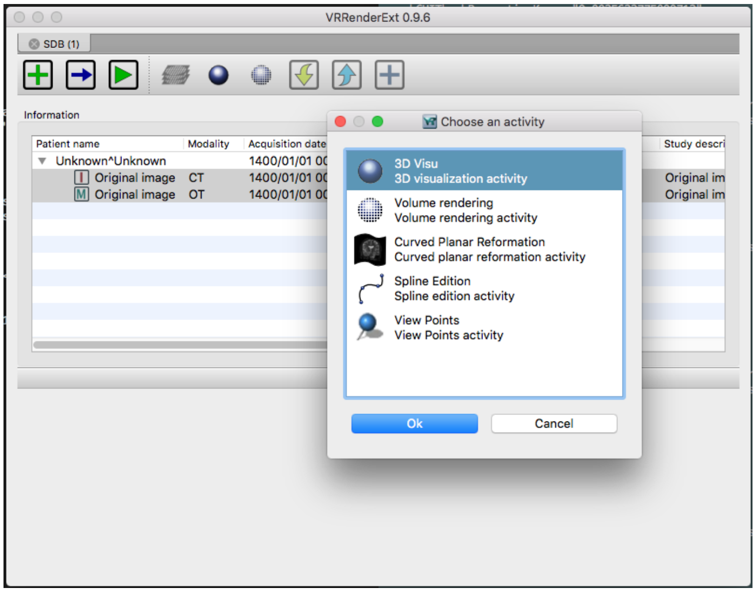
The description of the activity. It is displayed by the SActivityLauncher when several activity can be launched with the selected data.

icon

The path to the activity icon. It is displayed by the SActivityLauncher when several activity can be launched with the selected data.

requirements

The list of the data required to launch the activity. This data must be selected in the vector (`::fwData::Vector`).



Patient name	Modality	Acquisition date	Study descri
Unknown^Unknown		1400/01/01 00	
Original image	CT	1400/01/01 00	Original im
Original image	OT	1400/01/01 00	Original im

3D Visu
3D visualization activity

Volume rendering
Volume rendering activity

Curved Planar Reformation
Curved planar reformation activity

Spline Edition
Spline edition activity

View Points
View Points activity

requirement: A required data.

name: Key used to add the data in the activity Composite.

type: The data type (ex: `::fwMedData::ImageSeries`).

minOccurs (optional, "1" by default): The minimum number of occurrences of this type of object in the vector.

maxOccurs (optional, "1" by default): The maximum number of occurrences of this type of object in the vector.

container (optional, "vector" or "composite", default: composite): Container used to contain the data if minOccurs or maxOccurs are not "1". If the container is "composite", you need to specify the "key" of each object in the composite.

builder

Implementation of the activity builder. The default builder is `::fwActivities::builder::ActivitySeries` : it creates the `::fwMedData::ActivitySeries` and adds the required data in its composite with de defined key.

The builder `::fwActivities::builder::ActivitySeriesInitData` allows, in addition to what the default builder does, to create data when minOccurs == 0 and maxOccurs == 0.

validators (optional)

It defines the list of validators. If you need only one validator, you don't need the "validators" tag (only "validator").

validator (optional): It allows to validate if the selected required objects are correct for the activity.

For example, the validator `::fwActivities::validator::ImageProperties` checks that all the selected images have the same size, spacing and origin.

appConfig

It defines the AppConfig to launch and its parameters

id: Identifier of the AppConfig

parameters: List of the parameters required by the AppConfig

parameter: Defines a parameter

> **replace:** Name of the parameter as defined in the AppConfig
>
> **by:** Defines the string that will replace the parameter name. It should be a simple string (ex. frontal) or define a sesh@ path (ex. @values.myImage). The root object of the sesh@ path is the composite contained in the ActivitySeries.

3.6 Multithreading

3.6.1 Overview

The multithreading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge to prominence from the more specialized field of transaction processing:

- Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs.
- Techniques that would allow speedup of the overall system throughput of all tasks would be a meaningful performance gain.

Some advantages include:

- If a thread gets a lot of cache misses, the other thread(s) can continue, taking advantage of the unused computing resources, which thus can lead to faster overall execution, as these resources would have been idle if only a single thread was executed.
- If a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread can avoid leaving these idle.
- If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization of its values.

Some criticisms of multithreading include:

- Multiple threads can interfere with each other when sharing hardware resources such as caches or translation look aside buffers (TLBs).
- Execution times of a single thread are not improved but can be degraded, even when only one thread is executing. This is due to slower frequencies and/or additional pipeline stages that are necessary to accommodate thread-switching hardware.
- Hardware support for multithreading is more visible to software, thus requiring more changes to both application programs and operating systems than multiprocessing.
- Thread scheduling is also a major problem in multithreading.

Michael K. Gschwind, et al. [1]

[1] Michael K. Gschwind, Valentina Salapura. 2011. Using Register Last Use Infomation to Perform Decode-Time Computer Instruction Optimization US 20130086368 A1 [Patent]. http://www.google.com/patents/US20130086368

3.6.2 Worker and Timer

In the FW4SPL architecture, the library `fwThread` provides few tools to execute asynchronous tasks on different threads.

In this library, the class `Worker` creates and manages a task loop. The default implementation creates a loop in a new thread. Some tasks can be posted on the worker and will be executed on the managed thread. When the worker is stopped, it waits for the last task to be processed and stops the loop.

```
::fwThread::Worker::sptr worker = ::fwThread::Worker::New();

::boost::packaged_task<void> task( ::boost::bind( &myFunction ) );
::boost::future< void > future = task.get_future();
::boost::function< void () > f = moveTaskIntoFunction(task);

worker->post(f);

future.wait();
worker->stop();
```

The Timer class provides single-shot or repetitive timers. A Timer triggers a function once after a delay, or periodically, inside the worker loop. The delay or the period is defined by the duration attribute.

```
::fwThread::Worker::sptr worker = ::fwThread::Worker::New();

::fwThread::Timer::sptr timer = worker->createTimer();

timer->setFunction(  ::boost::bind( &myFunction)  );

::boost::chrono::milliseconds duration
        = ::boost::chrono::milliseconds(100) ;
timer->setDuration(duration);

timer->start();
//...
timer->stop();

worker->stop();
```

3.6.3 Mutex

The namespace `fwCore::mt` provides common foundations for multithreading in FW4SPL, especially tools to manage mutual exclusions. In computer science, mutual exclusion refers to the requirement of ensuring that two concurrent threads are not in a critical section at the same time, it is a basic requirement in concurrency control, to prevent race conditions. Here, a critical section refers to a period when the process accesses a shared resource, such as shared memory. A lock system is designed to enforce a mutual exclusion concurrency control policy.

Currently, FW4SPL uses Boost Thread library which allows the use of multiple execution threads with shared data, keeping the C++ code portable. `fwCore::mt` defines a few typedef over Boost:

```
namespace fwCore
{
namespace mt
{

typedef ::boost::mutex Mutex;
typedef ::boost::unique_lock< Mutex > ScopedLock;
typedef ::boost::recursive_mutex RecursiveMutex;
typedef ::boost::unique_lock< RecursiveMutex > RecursiveScopedLock;

/// Defines a single writer, multiple readers mutex.
typedef ::boost::shared_mutex ReadWriteMutex;
/**
* @brief Defines a lock of read type for read/write mutex.
* @note Multiple read lock can be done.
*/
typedef ::boost::shared_lock< ReadWriteMutex > ReadLock;
/**
* @brief Defines a lock of write type for read/write mutex.
* @note Only one write lock can be done at once.
*/
typedef ::boost::unique_lock< ReadWriteMutex > WriteLock;
/**
* @brief Defines an upgradable lock type for read/write mutex.
* @note Only one upgradable lock can be done at once but there
        may be multiple read lock.
*/
typedef ::boost::upgrade_lock< ReadWriteMutex > ReadToWriteLock;
/**
* @brief Defines a write lock upgraded from ReadToWriteLock.
* @note Only one upgradable lock can be done at once but there
        may be multiple read lock.
*/
typedef ::boost::upgrade_to_unique_lock< ReadWriteMutex >
            UpgradeToWriteLock;

} //namespace mt
} //namespace fwCore
```

3.6.4 Multithreading and communication

Asynchronous call

Slots are able to work with `fwThread::Worker`. If a Slot has a Worker, each asynchronous execution request will be run in its worker, otherwise asynchronous requests can not be satisfied without specifying a worker.

Setting worker example:

```
::fwCom::Slot< int (int, int) >::sptr slotSum
        = ::fwCom::newSlot( &sum );
::fwCom::Slot< void () >::sptr slotStart
        = ::fwCom::newSlot( &A::start, &a );

::fwThread::Worker::sptr w = ::fwThread::Worker::New();
slotSum->setWorker(w);
slotStart->setWorker(w);
```

`asyncRun` method returns a boost::shared_future< void >, that makes it possible to wait for end-of-execution.

```
::boost::future< void > future = slotStart->asyncRun();
// do something else ...
future.wait(); //ensures slotStart is finished before continuing
```

asyncCall method returns a boost::shared_future< R > where R is the return type. This allows facilitates waiting for end-of-execution and retrieval of the computed value.

```
::boost::future< int > future = slotSum->asyncCall();
// do something else ...
future.wait(); //ensures slotStart is finished before continuing
int result = future.get();
```

In this case, the slots asynchronous execution has been *weakened*. For an async call/run pending in a worker queue, it means that :

- if the slot is detroyed before the execution of this call, it will be canceled.
- if slot's worker is changed before the execution of this call, it will also be canceled.

Asynchronous emit

As slots can work asynchronously, triggering a Signal with asyncEmit results in the execution of connected slots in their worker :

```
sig2->asyncEmit(21, 42);
```

The instruction above has the consequence of running each connected slot in its own worker.

Note: Each connected slot must have a worker set to use asyncEmit.

3.6.5 Object-Service and Multithreading

Object

The architecture allows the writing of thread safe functions which manipulate objects easily. Objects have their own mutex (inherited from fwData::Object) to control concurrent access from different threads. This mutex is available using the following method:

```
::fwCore::mt::ReadWriteMutex & getMutex();
```

The namespace fwData::mt contains several helpers to lock objects for multithreading:

- ObjectReadLock: locks an object mutex on read mode.
- ObjectReadToWriteLock: locks an object mutex on upgradable mode.
- ObjectWriteLock: locks an object mutex on exclusive mode.

The following example illustrates how to use these helpers:

```
::fwData::String::sptr m_data = ::fwData::String::New();
{
    // lock data to write
    ::fwData::mt::ObjectReadLock readLock(m_data);
} // helper destruction, data is no longer locked


{
    // lock data to write
    ::fwData::mt::ObjectWriteLock writeLock(m_data);

    // unlock data
    writeLock.unlock();
    // lock data to read
    ::fwData::mt::ObjectReadToWriteLock updrageLock(m_data);

    // unlock data
    updrageLock.unlock();

    // lock again data to read
    updrageLock.lock();

    // lock data to write
    updrageLock.upgrade();

    // lock data to read
    updrageLock.downgrade();

} // helper destruction, data is no longer locked
```

Services

The service architecture allows the writing of a thread-safe service by avoiding the requirement of explicit synchronization. Each service has an associated worker in which service methods are intended to be executed.

Specifically, all inherited `IService` methods (`start`, `stop`, `update`, `receive`, `swap`) are slots. Thus, the whole service life cycle can be managed in a separate thread.

Since services are designed to be managed in an associated worker, the worker can be set/updated by using the inherited method :

```
// Initializes m_associatedWorker and associates
// this worker to all service slots
void setWorker( ::fwThread::Worker::sptr worker );

// Returns associate worker
::fwThread::Worker::sptr getWorker() const;
```

Since the signal-slot communication is thread-safe and `IService::receive(msg)` method is a slot, it is possible to attach a service to a thread and send notifications to execute parallel tasks.

Note: Some services use or require GUI backend elements. Thus, they can't be used in a separate thread. All GUI elements must be created and managed in the application main thread/worker.

3.7 Serialization

3.7.1 Overview

Serialization is the process to save plain C++ structures from memory to hard drive. In fw4spl, `fwAtoms` library provides tools to serialize all data (and especially Object that extend `::fwData::Object`) to a JSON format [1]. Of course, this process is also available for loading data from JSON format to plain C++ structures.

To achieve this serialization, `fwAtoms` provides basic structures (which extend `::fwAtoms::Base`) to manage better plain C++ structure evolution. Thus, there are two main steps in the serialization process:

[1] Introducing JSON. http://json.org/

- Converting a `::fwData::Object` into a `::fwAtoms::Object`
- Serializing a `::fwAtoms::Base` in a JSON format

3.7.2 Atom objects

The basic structures provided by `fwAtoms` library are a set of restricted C++ type. All these structures extend `::fwAtoms::Base` and cover all basic types and containers:

type	brief
::fwAtoms::Base	Base class of all atoms
::fwAtoms::String	Atom to represent string types
::fwAtoms::Numeric	Atom to represent numeric types (floating number or integer)
::fwAtoms::Boolean	Atom to represent a boolean value
::fwAtoms::Map	Atom to represent an associative container (std::string to ::fwAtoms::Base)
::fwAtoms::Sequence	Atom to represent a sequence of object like vector or list
::fwAtoms::Object	Atom to represent a C++ object with attributes
::fwAtoms::Blob	Atom to represent binary information like buffers

For instance, consider the following C++ class:

```
class SimpleClass
{
    bool m_myBoolean;
};

class ComplexClass
{
    std::string m_myString;
    float m_myFloat;
    SimpleClass* m_mySimpleClass;
};
```

It's Atom equivalent is (simplified code):

```
fwAtoms::Object
{
    metaInfos
    {
        "CLASSNAME_METAINFO" : "SimpleClass"
        "ID_METAINFO" : "<ID of the object>"
    }

    attributes
    {
        "myBoolean" : ::fwAtoms::Boolean
    }
}


fwAtoms::Object
{
    metaInfos
    {
        "CLASSNAME_METAINFO" : "ComplexClass"
        "ID_METAINFO" ; "<ID of the object>"
    }
    attributes
    {
        "myString" : ::fwAtoms::String
        "myFloat" : ::fwAtoms::Numeric
        "mySimpleClass" : ::fwAtoms::Object("SimpleClass")
    }
}
```

The main advantage of this representation is the ability to change easily the form of a class. In fact, all plain C++ objects are represented as `Atoms::Object` with a map of attributes. Thus, adding, removing or changing the content of an attribute is easy. Moreover, because of these restricted types, atom parsing is also made easier. The main difficulty is how to convert plain C++ object using this set of restricted types.

3.7.3 Convert a `fwData::Object`

As explained earlier, all objects in fw4spl inherit from the `::fwData::Object` class. To convert a C++ object in Atom, it must inherit from this class. To allow this conversion, some work must be done.

The first thing is to update the header file of the structure and add these lines :

```
// Before all namespace
fwCampAutoDeclareDataMacro((<namespace elem>)
        (<namespace elem>)(<class name>), <method export macro>);

// In the public class part
fwCampMakeFriendDataMacro((<namespace elem>)
        (<namespace elem>)(<class name>));
```

These two functions allow the declaration of the class to the conversion process.

Next, the conversion systems must know the class information including attributes, base class, library location and data version. This is achieved by creating a class which defines these properties.

Example

This can be illustrated by taking the previous class and creating these two files:

Header file of the newly created class: ComplexClass.hpp

```
// Reference class

fwCampAutoDeclareDataMacro((fwData)(ComplexClass), FWDATA_API);

namespace fwData
{
class ComplexClass : public ::fwData::Object
{
    fwCampMakeFriendDataMacro((fwData)(ComplexClass));

    std::string m_myString;
    float m_myFloat;
    ::fwData::SimpleClass* m_mySimpleClass;
};
}
```

Header file of serialization class :

```
// hpp binding file
#include <fwCamp/macros.hpp>
#include <fwData/ComplexClass.hpp>
#include "fwDataCamp/config.hpp"

fwCampDeclareAccessor((fwData)(ComplexClass), (fwData)(SimpleClass));
```

Source file of serialization class :

```
// cpp binding file
// include previous cpp file

#include <fwCamp/UserObject.hpp>

fwCampImplementDataMacro((fwData)(ComplexClass))
{
    builder
        .tag("object_version", "1")
        .tag("lib_name", "fwData")
        .base< ::fwData::Object>()
        .property("myString" , &::fwData::ComplexClass::m_myString)
        .property("myFloat" , &::fwData::ComplexClass::m_myFloat)
        .property("mySimpleClass" , &::fwData::ComplexClass::m_mySimpleClass)
        ;
}
```

In a header file, the method fwCampDeclareAccessor is necessary when an object has a pointer or a smart pointer to another object.

In a source file, fwCampImplementDataMacro declares the properties of the bound object with an object called a builder: it provides several methods to describe the object to bind.

method	brief
tag(key, value)	Register a tag in the atom meta information.
base<BaseClass>()	Identify the base class of the bound object
property(arg1, arg2)	Set property of the object and how to access it

Most of the work is completed when the header file of the relevant class has been updated and a binding class created. The last step is to register the binding class in the conversion system using the following line in the library containing binding classes:

```
localDeclarefwDataComplexClass();
```

In fw4spl, data are located in `fwData` library whereas data binding classes are located in `fwDataCamp` library. The above line registering a binding class can be found in `fwDataCamp` autoload.hpp files.

Serialization file example

For more information about serialization see:

location	brief
Srclib/core/fwData/include/	fwData header files with serialization macros
Srclib/core/fwDataCamp	Serialization description of all fw4spl data
Srclib/core/fwDataCamp/include/fwDataCamp/autoload.hpp	Auto loading data bindings in the system

fwData::Object to fwAtoms::Object conversion

The requirements to convert an fwData::Object into an fwAtoms::Object are in the fwAtomConversion library.

Two functions are necessary to achieve this conversion:

```
//Convert a fwData::Object into fwAtoms::Object
SPTR(::fwAtoms::Object) convert( const SPTR(::fwData::Object) &data );

//Convert a fwAtoms::Object into fwData::Object
SPTR(::fwData::Object) convert( const SPTR(::fwAtoms::Object) &atom );
```

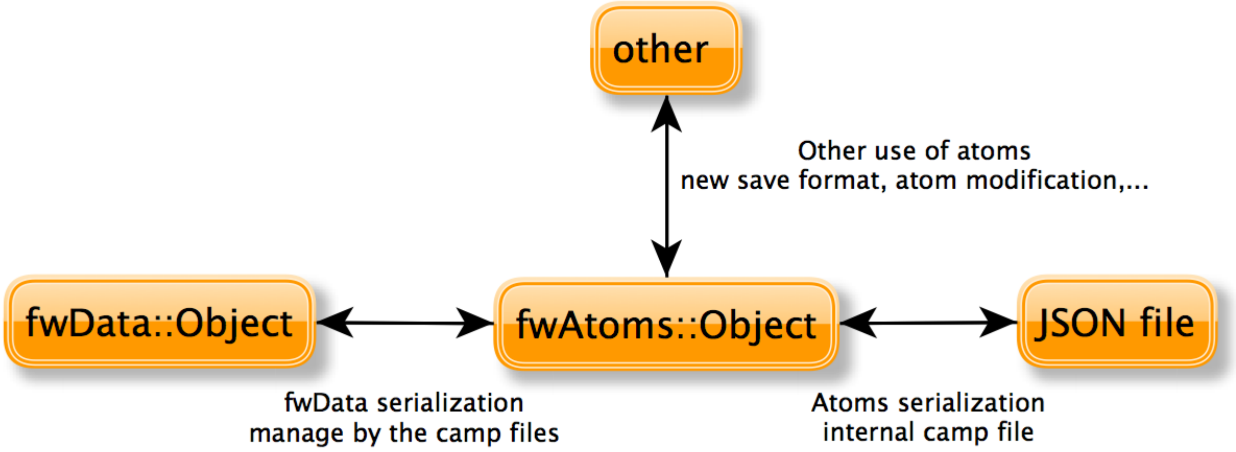
3.7.4 Serialize an Atoms object to JSON format

When a fw4spl data is converted into Atoms, it can be saved in JSON format. Both an Atom reader and Atom writer are available in the fwAtomsBoostIO fw4spl library: simply instantiate one of these classes with an Atom object and call the read or write method.

To serialize atoms into JSON, a visitor pattern is used. An example can be found in the fwAtomsBoostIO/Reader.cpp file.

3.7.5 Conclusion

Accordingly, you have now the requirements to serialize data in the framework and a basic knowledge about the mechanism behind it. To conclude, this is a diagram of the serialization mechanism:



3.8 Medical patient folder

DICOM is a software integration standard that is used in Medical Imaging. All modern medical imaging systems (aka Imaging Modalities) equipment like X-Rays, Ultrasounds, CT (Computed Tomography), and MRI (Magnetic Resonance Imaging) support DICOM and use it extensively. The core of DICOM is a file format and a networking protocol.

All Medical Images are saved in DICOM format. Medical Imaging Equipment creates DICOM files. Doctors use DICOM Viewers, computer software applications that can display DICOM images.

> DICOM files contain more than just images. Every DICOM file holds patient information (name, ID, sex and birth date), important acquisition data (e.g., type of equipment used and its settings), and the context of the imaging study that is used to link the image to the medical treatment it was part of.

Roni Z. 2011. Introduction to DICOM [1]:

The objects representing the medical patient data In FW4SPL are aligned with the DICOM standard. In the library `fwMedData` several structures and values have been retrieved:

- `Patient`: name, primary hospital identification number, birth date and sex.
- `Study`: unique identifier of the study, study date and time, referring physician, institution-generated description, age of the patient.
- `Equipment`: institution where the equipment that produced the composite instances is located.
- `Series`: unique identifier of the series, type of equipment that originally acquired the data used to create this series, series date and time, series description, name of the physician(s) administering the series.

In FW4SPL, the class `Series` is the main structure and contains pointers to Patient, Study and Equipment structure. The class `SeriesDB` is a container holding several instances of the `Series` class.

To specify an object of type `Series`, the library `fwMedData` holds the following classes inherited from `Series`:

- `ImageSeries` which corresponds to the image series of DICOM (CT images, MRI images, etc).
- `ModelSeries` which corresponds to the meshes series of DICOM and also represents 3D patient models.

The `fwMedData` library also provides a custom series called `ActivitySeries`. An `ActivitySeries` is a `Series` linked to an activity (sub part an application). Hence it is possible to save the state of all the objects used in the activity. Further application specific parameters which are not referred to an object can also be saved in an `ActivitySeries`. Application parameters in relation to the patient can be the view point on an organ, landmarks, calculated distances between organ points, etc.

3.9 Component-based software

The FW4SPL is also a component-based architecture.

Component-based software engineering (CBSE) (also known as component-based development (CBD)) is a branch of software engineering that emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. It is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software. Excerpt from “Component-based software engineering” [1] on Wikipedia

3.9.1 Definitions and characteristics

An individual software component is a software package that encapsulates a set of related code: resources, objects, services, XML configuration, etc.

[1] Roni Z. 2011. Introduction to DICOM. Introduction. http://dicomiseasy.blogspot.fr/2011/10/introduction-to-dicom-chapter-1.html

[1] Component-based software engineering http://en.wikipedia.org/wiki/Component-based_software_engineering

All the architecture is placed into separate components so that all of the data and functions inside each component are semantically related. Because of this principle, it is often said that components are modular and cohesive.

Components communicate with each other via interfaces. When a component offers services to the rest of the system, it adopts a provided interface which specifies services that other components can use. The generic architecture provided by classes Object/IService and the factory system make this interfacing easier.

Re-usability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them.

3.9.2 Component-based implementation

Implementation requires a dynamic structure which represents the component and a software launcher which loads and manages these components. A component, called a bundle, is just a simple folder that contains :

- the component description file (plugin.xml) to describe the content of the dynamic library
- the dynamic library, the type of which (.so, .dll, .dylib) differs between operating systems
- other shared resources (icons, XSD file, media files, ...)

The software launcher uses the library `fwRuntime` to parse the software description file (profile.xml) and load required dynamic libraries:

```
./launcher.exe mySoftware/profile.xml
```

The component description file (plugin.xml) is used to describe the content of the dynamic library. This file reveals which concepts and concept implementations are proposed by the component. These terms are identified in the file by keywords:

- Extension point: the concept
- Extension: a concept implementation (there can be many implementations one of a single concept)

In some cases, the Extension point is represented by an abstract class in a component, and the Extension by the class that it inherits from the abstract class of another component.

One example is the service concept. The component description file of servicesReg introduces the concept of service and incorporates the class IService into the dynamic library:

```
<plugin id="serviceReg">

  <library name="servicesReg" />

  <extension-point id="::fwServices::registry::ServiceFactory" />

</plugin>
```

And in another component, a new service is proposed in the dynamic library and the information is shared in the description file.

```
<plugin id="myBundle">

   <library name ="myBundle" />

   <! -- myBundle requires the bundle servicesReg to run -->
   <requirement id="servicesReg" />

   <! -- Need code related to ::io::IReader -->
   <requirement id="io" />

```

```
    <extension implements =" ::fwServices::registry::ServiceFactory ">
        <! -- service type -->
        <type>::io::IReader</type>
        <! -- the service name available in this component library -->
        <service>::myBundle::myReader</service>
        <! -- the object type associated to the service -->
        <object>::fwData::myData</object>
        <desc>Description of my reader</desc>
    </extension>

</plugin>
```

Even if it is often the case, concepts are not limited to class level. A lot a concepts can be defined : service configurations, operator parameters, etc.

3.10 Manager and updater services

3.10.1 Concepts

In the FW4SPL architecture, there is an object container which is often used: `::fwData::Composite`. This container is also an Object and represents a map which associates a string with an Object. The architecture provides two main services to manage a Composite: a composite updater and a service manager.

Updater

The updater service extends service type `::ctrlSelection::IUpdaterSrv` and the work on a selection composite. This kind of service listens specific events from objects identified by their UID. When it receives an event, it performs an operation on an object in the selection composite and notifies composite listeners.

Available operations on composite are:

- Adding an object
- Swapping an object
- Removing an object
- Removing an object if present
- Adding or swapping an object
- Doing nothing

There are few generic updater services which listen all events sent by Objects, and few other which work with particular Object events.

Manager

The manager services extend service type `::ctrlSelection::IManagerSrv` and react to updater messages. This kind of service manages services on identified data if they are present in a composite. There are few manager services, but the most common is `::ctrlSelection::manager::SwapperSrv`. This service manages other services on objects stored in the composite. When this manager gets notified, it can perform an action defined in the manager configuration on the concerned object such as :

- starting the services of the concerned object

- stopping the services of the concerned object
- create communication connection between new objects and/or new services

3.10.2 Implementation

Updater

An updater implementation must inherit from the `::ctrlSelection::IUpdaterSrv` service.

In the example below, an updater is used to manage a `::fwData::Reconstruction` object identified with the `reconstruction` key in a selection composite. This `::fwData::Reconstruction` is stored in a `::fwMedData::ModelSeries` and we used a specific updater to listen signals and manage the structure.

The updater provites slots to react on object/service signals.

Example

For example, the updater `::ctrlSelection::SObjFromSlots` provides the following slots :

- `add(object)`: add the given object in the composite with the configured key
- `swapObj(object)`: swap the given object in the composite with the configured key
- `addOrSwap(object)`: if the configured key exists in the composote, the object is swapped, else it is added
- `remove()`: remove the object with the configured key from the composite
- `removeIfPresent()`: remove the object if the configured key exists in the composite

Updater configuration example:

```xml
<object id="model_uid" type="::fwMedData::ModelSeries">
    <service uid="${GENERIC_UID}_listOrganEditor" impl="::uiMedData::editor::SModelSeriesList"
        type="::gui::editor::IEditor" autoConnect="yes" />
</object>

<object type="::fwData::Composite">
    <service uid="myUpdater" impl="::ctrlSelection::updater::SObjFromSlot" type="::ctrlSelection::IUp
        <compositeKey>reconstruction</compositeKey><!-- key of the updated object -->
    </service>
</object>

<!-- connect updater to listen the reconstruction selection -->
<connect>
    <signal>listOrganEditor/reconstructionSelected</signal>
    <slot>myUpdater/addOrSwap</slot>
</connect>
```

Manager

Managers inherit from `::ctrlSelection::IManagerSrv`. As explained earlier, they manage tasks or services on objects which appear or disappear from the composite on which they are working.

Example

For instance, the XML configuration below manages a GUI to configure rendering options of a reconstruction from a reconstruction list thanks to the `::ctrlSelection::manager::SwapperSrv` service. In this configuration, the manager updates the services attached to the `rec` object each time it is added, removed or swapped.

Manager configuration example

```
<object type="::fwData::Composite">
  <service uid="manager_uid" impl="::ctrlSelection::manager::SwapperSrv"
        type="::ctrlSelection::IManagerSrv" autoConnect="yes" >
        <mode type="dummy" />
        <config>
            <object id="rec" type="::fwData::Reconstruction">
                <service uid="organMaterialEditor" impl="::uiReconstruction::OrganMaterialEditor"
                    type="::gui::editor::IEditor" />
                <service uid="representationEditor" impl="::uiReconstruction::RepresentationEditor"
                    type="::gui::editor::IEditor" />
            </object>
    </config>
  </service>
</object>
```

mode The mode must be “stop”, “dummy” or “startAndUpdate”. The mode “stop”, used by default, starts the services when their attached object is added in the compsite and stop and unregister the services when the object is deleted. The mode “dummy” doesn’t stop the services when its attached object is deleted but swap it on a dummy object. The mode “startAndUpdate” start and update the services when its attached object is added in the composite.

object It defines the objects and their services to manage.

- **id**: the key of the object in the composite
- **type**: the type of the object

The services are declared as same as in the AppConfig.

connect (optional): It allows to connect a signal to one or more slot(s). The signal and slots must be compatible. The signal uid is optional, if it is not defines, the signal is from the current managed object.

```
<object type="::fwData::Composite">
    <service uid="manager_uid" impl="::ctrlSelection::manager::SwapperSrv"
        type="::ctrlSelection::IManagerSrv" autoConnect="yes" >
        <mode type="dummy" />
        <config>
            <object id="rec" type="::fwData::Reconstruction">

                <!--  ....  services ....    -->

                <connect>
                    <signal>object_uid/signal_name</signal>
                    <slot>service_uid/slot_name</slot>
                </connect>

                <connect>
                    <signal>signal_name</signal><!-- signal from recontruction "rec" -->
                    <slot>service_uid/slot_name</slot>
                </connect>
            </object>
        </config>
    </service>
<object>
```

proxy (optional): It allows to connect one or more signal(s) to one or more slot(s). The signals and slots must be compatible. The signal uid is optional, if it is not defines, the signal is from the current managed object.

channel: Name of the channel use for the proxy.

```
<object type="::fwData::Composite">
    <service uid="manager_uid" impl="::ctrlSelection::manager::SwapperSrv"
        type="::ctrlSelection::IManagerSrv" autoConnect="yes" >
        <mode type="dummy" />
        <config>
            <object id="rec" type="::fwData::Reconstruction">

                <!--  ....  services ....    -->

                <proxy channel="myChannel">
                    <signal>object_uid/signal_name</signal>
                    <slot>service_uid/slot_name</slot>
                </proxy>

                <proxy channel="myOtherChannel">
                    <signal>signal_name</signal><!-- signal from recontruction "rec" -->
                    <slot>service_uid/slot_name</slot>
                </proxy>
            </object>
        </config>
    </service>
<object>
```

3.11 Graphical User Interface

3.11.1 Overview

Graphical User Interface (GUI) is the process of displaying the graphical components of an application. In fw4spl, the `fwGui` library provides abstract tools to display components like windows, buttons, textfield, aso.

The software architecture provides a way of selecting different backends in order to manage the GUI components. As a result, the `fwGuiQt` library has been created to display components created using the Qt soup. Presently, this backend is the only one supported by the applications.

3.11.2 Backend

When creating an application, we need to specify which gui backend we want to use. To do so, the chosen gui bundle must be activated and started in the profile.xml of the application. The main gui bundle for any application is `guiQt`. The `gui` bundle must be activated regardless of the chosen backend.

```
<activate id="gui" version="0-1" />
<activate id="guiQt" version="0-1" />

<!-- ... -->

<start id="guiQt" />
```

Warning : The gui backend bundle must be started before any other bundle in the profile.xml.

3.11.3 Configuration

Frames

The frame is the main component of a GUI. The main service used to represent a general frame is `::fwGui::IFrameSrv`. The service `::gui::frame::DefaultFrame` is the default implementation for the main application frame. Every backend must provide its own implementation of this service.

The DefaultFrame service is configurable with different parameters :

- Application name
- Application icon
- Minimum window size
- GUI elements (toolbar, menubar, aso.)

```
<service uid="mainFrame" type="::fwGui::IFrameSrv"
    impl="::gui::frame::DefaultFrame" autoConnect="no" >
    <gui>
        <frame>
            <name>Application name</name>
            <icon>path_to_application_icon</icon>
            <minSize width="800" height="600"/>
        </frame>
        <menuBar />
        <toolBar >
            <toolBitmapSize height= "32" width="32" />
        </toolBar>
    </gui>
    <registry>
        <menuBar sid="menuBar" start="yes" />
        <toolBar sid="toolBar" start="yes" />
        <view sid="view" start="yes" />
    </registry>
</service>
```

Menus and actions

The menu bar is used to organize application action groups. The main service used to display that kind of bar is `::fwGui::IMenuBarSrv`. The service `::gui::aspect::DefaultMenuBarSrv` is the default implementation. Every backend must provide its own implementation of this service.

The configuration is used to associate a menu label with the service representing the menu.

```
<service uid="menuBar" type="::fwGui::IMenuBarSrv"
    impl="::gui::aspect::DefaultMenuBarSrv" autoConnect="no" >
    <gui>
        <layout>
            <menu name="First Menu"/>
            <menu name="Second Menu"/>
        </layout>
    </gui>
    <registry>
        <menu sid="firstMenu" start="yes" />
        <menu sid="secondMenu" start="yes" />
    </registry>
</service>
```

The main service used to display a menu is ::fwGui::IMenuSrv. The service ::gui::aspect::DefaultMenuSrv is the default implementation. Every backend must provide its own implementation of this service.

The configuration is used to associate an action name and the service performing the action. An action can be configured with a shortcut, a style (default, check, radio) and/or an icon. Several special actions can also be specified (QUIT, ABOUT, aso.).

```
<service uid="myMenu" type="::fwGui::IMenuSrv"
    impl="::gui::aspect::DefaultMenuSrv" autoConnect="no" >
    <gui>
        <layout>
            <menuItem name="First Item" icon="icon_path" />
            <menuItem name="Checked Item" style="check" />
            <separator />
            <menuItem name="Quit" shortcut="Ctrl+Q" specialAction="QUIT" />
        </layout>
    </gui>
    <registry>
        <menuItem sid="actionFirstItem" start="no" />
        <menuItem sid="actionCheckedItem" start="no" />
        <menuItem sid="actionQuit" start="no" />
    </registry>
</service>
```

A menu can also be displayed using a tool bar. The main service used to display a tool bar is ::fwGui::IToolBarSrv. The service ::gui::aspect::DefaultToolBarSrv is the default implementation. Every backend must provide its own implementation of this service.

The configuration of a tool bar is the same as the one used to describe a menu.

Layouts

The layouts are used to organize the different parts of a GUI. The main service used to manage layouts is ::fwGui::IGuiContainerSrv. The service ::gui::view::DefaultView is the default implementation. Every backend must provide its own implementation of this service.

Several types of layout can be used :

- Line layout
- Cardinal layout
- Tab layout

Every layout can be configured with a set of parameters (orientation, alignment, aso.).

```
<service uid="subView" type="::gui::view::IView"
    impl="::gui::view::DefaultView" autoConnect="no" >
    <gui>
        <layout type="::fwGui::LineLayoutManager" >
            <orientation value="horizontal" />
            <view caption="view1" />
            <view caption="view2" />
        </layout>
    </gui>
    <registry>
        <view sid="subView1" start="yes" />
        <view sid="subView2" start="yes" />
    </registry>
</service>
```

3.11.4 Multi-threading

The `fwGui` library has been designed to support multi-thread application. When a GUI component needs to be accessed, the function call must be encapsulated in a lambda declaration as shown in this example:

```
::fwGui::registry::Worker::get()->postTask<void>(
[&] {
        //TODO Write function calls
}
).wait();
```

This encapsulation is required because all access to GUI components must be performed in the thread containing the GUI. It moves the function calls from the current thread, to the GUI thread.

3.12 Generic Scene

3.12.1 Overview

A generic scene in FW4SPL is visualization feature to visualize several elements like meshes or images in a scene. The scene is based on VTK. The main task of the generic scene is to manage all visualization services of the different elements contained in the scene. The generic scene is universal and therefore applicable for divers visualization tasks.

As used in FW4SPL, the scene generic configures a VTK scene with a simple xml configuration. Hence, FW4SPL is mainly used for medical assignments, the generic scene can be seen as fusion of a negatoscope and the 3D model visualization.

3.12.2 Components

Manager

The SRender is the manager service of the VTK scene. This service works on an object of type *fwData::Composite* that contains all objects to display.

The manager retrieves its specified container. The VTK context (vtkRender and vtkRenderWindow) is installed in the container of the manager.

The manager listens to the object signals of the associated *fwData::Composite* object. The transferred signals inform the manager if objects in the *fwData::Composite* have been added, removed or changed. In response to the modifications within the *fwData::Composite* object the manager supervises the starting and stopping of the visualization services (adaptors explained below), which are specified in its configuration. Thus, an object is added or removed to the *fwData::Composite* object, the corresponding adaptor which works on this object is started or stopped.

Adaptor

An adaptor (inherited from `::fwRenderVTK::IVtkAdaptorService`) is a service to manipulate or display a FW4SPL data. Services representing an adaptor are managed by a generic scene (SRender). The adaptors are the gateway between FW4SPL objects and VTK objects. To respect the principles of the framework, adaptors are kept as generic as possible. Therefore they are reusable in further applications or even adaptors.

An adaptor is a specific service that need to implements the methods `doStart`, `doStop`, `doUpdate`, *doConfigure*‘ and `doSwap` instead of the usual `starting`, `updating`, ...

```cpp
class MyAdaptor : public ::fwRenderVTK::IVtkAdaptorService
{

public:

    fwCoreServiceClassDefinitionsMacro ( (MyAdaptor)(::fwRenderVTK::IVtkAdaptorService) );

protected:

    /// Parse the adaptor "config" tag
    void configuring() throw(fwTools::Failed);

    /// Initialize the vtk pipeline (actor, mapper, ...)
    void doStart();

    /// Clear the vtk pipeline
    void doStop();

    /// Update the pipeline from the current object
    void doUpdate();

    /// Update the pipeline with the new object (eventually call doStop();doStart();)
    void doSwap();
};
```

3.12.3 Configuration

```xml
<service uid="generiSceneUID" impl="::fwRenderVTK::SRender" type="::fwRender::IRender">
    <scene renderMode="auto|timer|none" offScreen="imageKey" width="1920" height="1080">
        <renderer id="myRenderer" layer="0" background="0.0" />
        <vtkObject id="transform" class="vtkTransform" />
        <picker id="negatodefault" vtkclass="fwVtkCellPicker" />

        <adaptor id="tmAdaptor" class="::visuVTKAdaptor::Transform" uid="adaptorUID" objectId="tm3dKe
            <config transform="transform" />
        </adaptor>
        <adaptor id="snapshot" class="::visuVTKAdaptor::Snapshot" objectId="self">
            <config renderer="myRenderer" />
        </adaptor>

        <!-- ...... -->

        <connect>
            <signal>adaptorUID/modified</signal>
            <slot>serviceUid/updateTM</slot>
        </connect>

        <connect waitForKey="tm3dKey">
            <signal>modified</signal><!-- signal for object "tm3dKey" -->
            <slot>serviceUid/updateTM</slot>
        </connect>
        <proxy channel="myChannel">
            <signal>adaptor2UID/modified</signal>
            <slot>service2Uid/updateTM</slot>
        </proxy>
    </scene>
    <fps>30</fps><!-- used if renderMode=="timer" -->
</service>
```

renderMode (optional, “auto” by default) This attribute is forwarded to all adaptors. For each adaptor, if renderMode=”auto”, the scene is automatically rendered after doStart, doUpdate, doSwap, doStop and m_vtkPipelineModified=true. If renderMode=”timer” the scene is rendered at N frame per seconds (N is defined by **fps** tag). If renderMode=”none” you should call ‘render’ slot to call reder the scene.

offScreen (optional): Key of the image used for off screen render

width (optional, “1280” by default): Width for off screen render

height (optional, “720” by default): Height for off screen render

renderer Defines a renderer. At least one renderer is mandatory, but there can be multiple renderer on different layers.

- **id** (mandatory): the identifier of the renderer
- **layer** (optional): defines the layer of the vtkRenderer. This is only used if there are layered renderers.
- **background** (optional): the background color of the rendering screen.

The color value can be defines as a grey level value (ex . 1.0 for white) or as a hexadecimal value (ex : #ffffff for white).

vtkObject

Represents a vtk object. It is usually used for vtkTransform or vtkImageBlend.

- **id** (mandatory): the identifier of the vtkObject
- **class** (mandatory): the classname of the vtkObject to create. For example vtkTransform, vtkImageBlend, ...

picker Represents a picker.

- **id** (mandatory): the identifier of the picker
- **vtkclass** (optional, by default vtkCellPicker): the classname of the picker to create.

adaptor

Defines the adaptors to display in the scene.

- **id** (mandatory): the identifier of the adaptor
- **class** (mandatory): the classname of the adaptor service
- **uid** (optional): the fwID to specify for the adaptor service
- **objectId** (mandatory): the key of the adaptor’s object in the scene’s composite.
- **autoConnect** (optional, “yes” by default): if “yes” the service slot are automatically connected to the object signals.
- **config**: adaptor’s configuration. It is parsed in the adaptor’s configuring() method.

Note: The “self” key is used when the adaptor works on the scene’s composite.

connect/proxy (optional)

Connects signal to slot

- **waitForKey** (optional): defines that the connection is made only if the key is present in the scene composite.
- **signal** (mandatory): must be signal holder UID, followed by '/', followed by signal name.
- **slot** (mandatory): must be slot holder UID, followed by '/', followed by slot name.

Note: To use the signal of the object (defined by waitForKey), you don't have to write object uid, only the signal name.

3.13 Data file migration

Contents

3.13.1 Overview

The data migration system consists on converting the data to another version. It allows us to adapt any version of data to any version of software, and thus ensuring compatibility between data and software independently of their version.

Migration process is applied on two independent steps :

- In `::ioAtoms::SReader` while reading data files, previously serialized with `fwAtoms`, right before converting said data to `::fwData::Objects`.
- In `::ioAtoms::SWriter` after data is converted to `fwAtoms::Base`.

3.13.2 Definitions

Context It represents a complex chunk of data. For example, the medical patient folder, the software preference file, etc. Hereafter we will consider a medical patient folder which is called **MedicalData**.

Structural patch This sort of patch affects only one object of the serialized data regardless of the context (ex: add or remove attribute, type, ...), see *Structural patch*.

Semantic patch This sort of patch is applied on a context to migrate to a given version without changing the data structure. (These patches are sometimes called contextual patches), see *Semantic patch*.

Patcher A patcher defines the methods to parse the data and applies the structural and semantic patches, see *Patcher*.

3.13.3 Data Version

After the conversion from `::fwData::Object` to `::fwAtoms::Object`, each data is assigned a version number. Said number is defined in the camp serialization source files (see Serialization). Each data structure modification causes an incrementation of the data version.

Example of data declaration for introspection (used to convert to `fwAtoms`):

```
#include <fwCamp/UserObject.hpp>

fwCampImplementDataMacro((fwData)(ComplexClass))
{
    builder
        .tag("object_version", "1") // data version
        .tag("lib_name", "fwData")
        .base< ::fwData::Object>()
        .property("myString" , &::fwData::ComplexClass::m_myString)
        .property("myFloat" , &::fwData::ComplexClass::m_myFloat)
        .property("mySimpleClass" , &::fwData::ComplexClass::m_mySimpleClass)
        ;
}
```

3.13.4 Context version

The context version must be incremented after a data version modification.

Note:

- If several data versions are modified simultaneously, only one incrementation of the context version is necessary.
- A single context version can contain data with different versions (see the example below).

The `.versions` file contains a detailed description of the context version, and the version of each data.

Example of `V1.versions`:

```
{
    "context": "MedicalData",
    "version_name": "V1",
    "versions":
    {
        "::fwData::Array": "1",
        "::fwData::Boolean": "1",
        "::fwData::Image": "1",
        "::fwData::Integer": "1",
        "::fwData::Material": "1",
        "::fwData::Mesh": "1",
        "::fwData::Patient": "1",
    }
}
```

Example of `V2.versions`:

```
{
    "context": "MedicalData",
    "version_name": "V2",
    "versions":
```

```
    {
        "::fwData::Array": "1",
        "::fwData::Boolean": "1",
        "::fwData::Image": "2",
        "::fwData::Integer": "1",
        "::fwData::Material": "1",
        "::fwData::Mesh": "1",
        "::fwMedData::Patient": "1",
    }
}
```

3.13.5 Migration

The migration is applied on a given context. It is described in the `.graphlink` file. It defines how to migrate from a context version to another.

Example of `V1ToV2.graphlink`:

```
{
    "context" : "MedicalData",
    "origin_version" : "V1",
    "target_version" : "V2",
    "patcher" : "DefaultPatcher",
    "links" : [
        {
            "::fwData::Patient" : "1",
            "::fwMedData::Patient" : "1"
        },
        {
            "::fwData::Image" : "1",
            "::fwData::Image" : "2"
        }
    ]
}
```

The `links` tag represents the data version modifications, by doing so, associated patches can be applied.

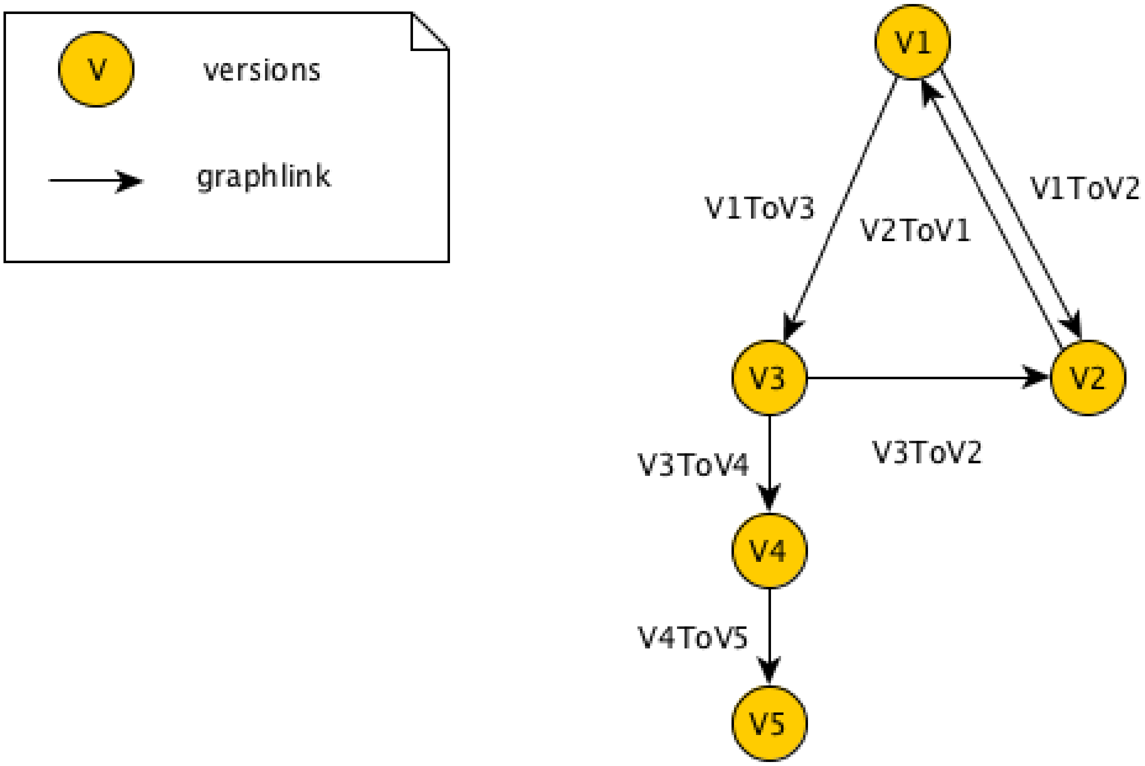
Warning: Two `.versions` files must be defined, one for each version (V1.versions and V2.versions).

Note: It is not necessary to specify a simple data version incrementation on the `links` tag, the patching system establishes this information from the data version defined in the `.versions` files.

3.13.6 Graph

The `.graphlink` and `.versions` files are parsed and the information is stored in the `::fwAtoms::VersionsManager`. Each context defines a graph.

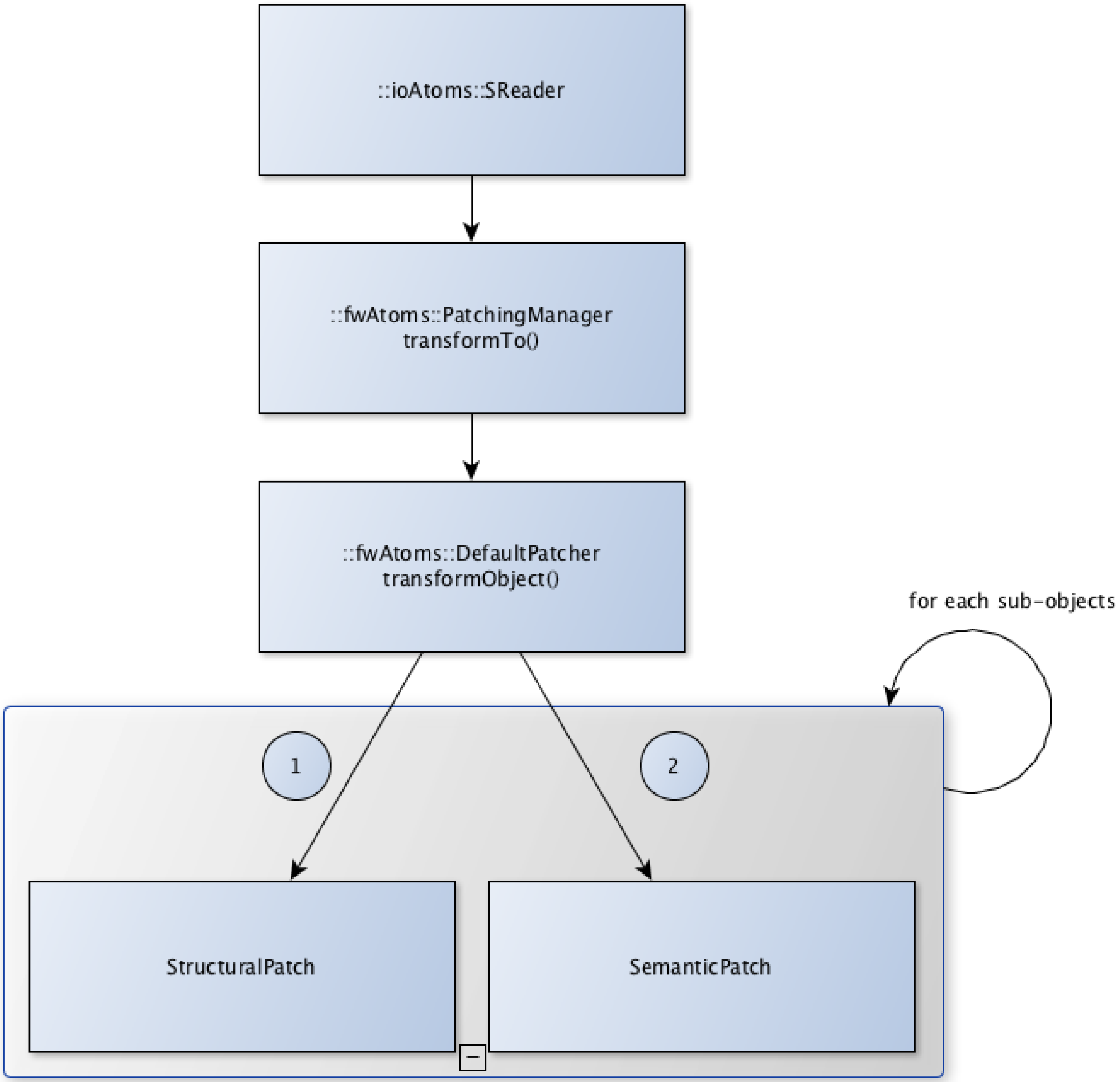
Example of graph:



The graph is used to find the migration path from an initial version to a target version. In our example, it is possible to migrate from V1 to V5, the data is converted to V3, V4 then V5. If several paths are possible, the shortest path is used.

3.13.7 Structure

The `fwAtomsPatch` library contains the base classes to perform the migration.



PatchingManager This class provides the `transformTo()` method used to migrate the data. It uses the graph to apply the patcher on each version.

patcher::IPatcher Base class for patchers.

patcher::DefaultPatcher Patcher used by default. It performs the data migration in two steps: first it applies the structural patches recursivly on each sub-object and then applies the semantic patches recursivly on each sub-object .

IPatch Base class for structural and semantic patches. It provides an `apply()` method that must be implemented in sub-classes.

ISemanticPatch Base class for semantic patches.

IStructuralPatch Base class for structural patches.

IStructuralCreator Base class for creators. It provides a `create()` method that must be implemented in sub-classes.

SemanticPatchDB Singleton used to register all the semantic patches.

StructuralPatchDB Singleton used to register all the structural patches.

CreatorPatchDB Singleton used to register all the creator patches.

VersionsGraph Registers the migration graphs.

VersionsManager Singleton used to register all the version graph.

The `fwStructuralPatch` library contains the structural patches for `fwData` and `fwMedData` conversion.

The `fwMDSemanticPatch` library contains the semantic patches for `fwData` and `fwMedData` conversion in the `MedicalData` context.

The `patchMedicalData` bundle must be activated in your application to allow migration in `MedicalData` context.

Structural patch

The structural patches are registered in the `::fwAtomsPatch::StructuralPatchDB` singleton. A structural patch provides a method `apply` that performs the structure conversion. The constructor defines the classname and versions of the origin and target objects as described in the `.graphlink` links section.

Example of structural patch to convert the `fwData::Image` from version 1 to 2. We add three attributes related to medical imaging: the number of components `nb_components`, the window center `window_center` and the window width `window_width`.

```
#include "fwStructuralPatch/fwData/Image/V1ToV2.hpp"

#include <fwAtoms/Numeric.hpp>
#include <fwAtoms/Numeric.hxx>

namespace fwStructuralPatch
{

namespace fwData
{

namespace Image
{

V1ToV2::V1ToV2() : ::fwAtomsPatch::IStructuralPatch()
{
    m_originClassname = "::fwData::Image";
    m_targetClassname = "::fwData::Image";
    m_originVersion   = "1";
    m_targetVersion   = "2";

}

// ----------------------------------------------------------------------------

void V1ToV2::apply(
    const ::fwAtoms::Object::sptr& previous, // object in the origin version
    const ::fwAtoms::Object::sptr& current, // clone of the previous object to convert in the targer
    ::fwAtomsPatch::IPatch::NewVersionsType& newVersions) // map < previous object, new object > asso
{
    // Check if the previous and current object version and classname correspond
    IStructuralPatch::apply(previous, current, newVersions);
    // Update object version
    this->updateVersion(current);

    // Create helper
    ::fwAtomsPatch::helper::Object helper(current);

    helper.addAttribute("nb_components", ::fwAtoms::Numeric::New(1));
    helper.addAttribute("window_center", ::fwAtoms::Numeric::New(50));
    helper.addAttribute("window_width", ::fwAtoms::Numeric::New(500));
}

} // namespace Image

} // namespace fwData

} // namespace fwStructuralPatch
```

To register the structural patch:

```
// fwStructuralPatch/autoload.cpp

::fwAtomsPatch::StructuralPatchDB::sptr structuralPatches = ::fwAtomsPatch::StructuralPatchDB::getDe
structuralPatches->registerPatch(::fwStructuralPatch::fwData::Image::V1ToV2::New());
```

Creator

The creator provides a method `create` that allows to create a new object with the default attribute initialization. The creator is used in structural patches to create new sub-objects. Creators are registered in the `::fwAtomsPatch::StructuralCreatorDB` singleton.

Creators are useful for adding an attribute that is a non-null object.

Example of creator for the `::fwMedData::Patient` :

```
#include "fwStructuralPatch/creator/fwMedData/Patient1.hpp"

#include <fwAtoms/String.hpp>

#include <fwAtomsPatch/helper/Object.hpp>

namespace fwStructuralPatch
{
namespace creator
{
namespace fwMedData
{

Patient1::Patient1()
{
    m_classname = "::fwMedData::Patient";
    m_version   = "1";
}

// ----------------------------------------------------------------------------

::fwAtoms::Object::sptr Patient1::create()
{
    // Create an empty ::fwAtoms::Object with the classname, version and ID informtation
    ::fwAtoms::Object::sptr patient = this->createObjBase();

    ::fwAtomsPatch::helper::Object helper(patient);

    helper.addAttribute("name", ::fwAtoms::String::New(""));
    helper.addAttribute("patient_id", ::fwAtoms::String::New(""));
    helper.addAttribute("birth_date", ::fwAtoms::String::New(""));
    helper.addAttribute("sex", ::fwAtoms::String::New(""));

    return patient;
}

} // namespace fwMedData
} // namespace creator
} // namespace fwStructuralPatch
```

To register the creator:

```
// fwStructuralPatch/creator/autoload.cpp

::fwAtomsPatch::StructuralCreatorDB::sptr creators = ::fwAtomsPatch::StructuralCreatorDB::getDefault
creators->registerCreator(::fwStructuralPatch::creator::fwMedData::Equipment1::New());
```

Semantic patch

The semantic patches are registered in the `::fwAtomsPatch::SemanticPatchDB` singleton. The structural patch provides a method `apply` that performs the structure conversion. The constructor defines the origin classname, the origin version of the object, and the origin and the target context version as described in the `.graphlink`.

The semantic patch is used when we need several objects to perform the object migration.

Example of semantic patch :

```
#include "fwMDSemanticPatch/V2/V3/fwData/Image.hpp"

#include <fwAtoms/Object.hpp>
#include <fwAtoms/Object.hxx>
#include <fwAtoms/Numeric.hpp>
#include <fwAtoms/Numeric.hxx>

#include <fwAtomsPatch/helper/functions.hpp>


namespace fwMDSemanticPatch
{
namespace V2
{
namespace V3
{
namespace fwData
{

Image::Image() : ::fwAtomsPatch::ISemanticPatch()
{
    m_originClassname = "::fwData::Image";
    m_originVersion   = "1";
    this->addContext("MedicalData", "V2", "V3"); // Context version
}

// -----------------------------------------------------------------------------

void Image::apply(
    const ::fwAtoms::Object::sptr& previous, // object in the origin version
    const ::fwAtoms::Object::sptr& current, // clone of the previous object to convert in the targer
    ::fwAtomsPatch::IPatch::NewVersionsType& newVersions) // map < previous object, new object > asso
{
    // Check if the previous and current object version and classname correspond
    ISemanticPatch::apply(previous, current, newVersions);

    // Cleans object fields (also creates them if they are missing)
    ::fwAtomsPatch::helper::cleanFields( current );

    ::fwAtomsPatch::helper::Object helper( current );

    ::fwAtoms::Object::sptr array        = ::fwAtoms::Object::dynamicCast(previous->getAttribute("ar
    ::fwAtoms::Numeric::sptr nbComponent =
             ::fwAtoms::Numeric::dynamicCast(array->getAttribute("nb_of_components"));

    helper.replaceAttribute("nb_components", nbComponent->clone());
}

// -----------------------------------------------------------------------------

} // namespace fwData
} // namespace V3
} // namespace V2
} // namespace fwMDSemanticPatch
```

This patch changed the attribute `nb_components` in the image copied from array `nb_of_components`.

To register the semantic patch:

```
// fwMDSemanticPatch/V1/V2/fwData/autoload.cpp
::fwAtomsPatch::SemanticPatchDB::sptr contextPatchDB = ::fwAtomsPatch::SemanticPatchDB::getDefault();
contextPatchDB->registerPatch(::fwMDSemanticPatch::V1::V2::fwData::Composite::New());
```

Patcher

The patcher defines the methods to parse the data and applies the structural and semantic patches. It must inherit from `fwAtomsPatch::patcher::IPatcher` and implements the `transformObject()` method.

We usually use the `DefaultPatcher`. The conversion is processed in two steps: first it applies the structural patches recursivly on each sub-objects, then it applies the semantic patches recursively on each sub-objects.

Rules

Rule 1 A change in data (fwData, fwMedData, ...) involves the incrementation of the data version and the context version and thus, the creation of structural and/or semantic patch.

Rule 2 The creator patch creates the `fwAtoms::Object` representing the data object. The `::fwAtoms::Object` created must be the same as the data created with a `New()` and converted to `fwAtoms`.

Rule 3 The *buffer object* (converted as BLOB in fwAtoms) is just reused (without copy) during the migration. If its structure is modified, you should clone the buffer before applying the patch.

Rule 4 If an object is contained in the `fwAtoms::Object` to migrate but is not present in the current context version (in the `.versions` file), this object will be erased from the `fwAtoms::Object`.

3.13.8 Usage

If you have to modify data, you don't have to re-implement all the migration system, but there are steps to perform :

step 1 Increment the data version in camp declaration (and update the declaration of the attribute if needed). See *Data Version*.

step 2 Increment the context version: create new `.versions` files (with the associated data version). See *Context version*.

step 3 Create the `.graphlink` file. See *graphlink*.

step 4 (optional) Create the creator if you need to add a new non-null objet. See *Creator*.

step 5 Create the structural patch. See *Structural patch*.

step 6 (optional) Create the semantic patch if you need other objects to update the current one. See *Semantic patch*.

Note: You can create migration patches from V1 to V3 without using the V1 to V2 and V2 to V3.

CHAPTER 4

Coding style

4.1 Terminology

- Rules are mandatory. Any rule can be (exceptionally) exceeded, but if so, it has to be rigorously justified.
- Recommendations are optional.
- **Camel case** is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter. In programming languages, **camel case** is assumed to start with a lowercase letter. We will use the term **upper camel case** when it starts with a capital.

```
camelCaseLabel
UpperCamelCaseLabel
```

4.2 Generalities

Rule 44 [Preferred language] English is the preferred language for types, variables, functions naming, and code comments.

Rule 45 [Maximum size of a line] A source code line must not exceed 120 characters.

Rule 46 [Indentation] Use only spaces, and an indent level has four spaces.

4.3 C++ coding

4.3.1 Source and files

Rule 4 [Files tree] Source files must be placed in a folder `src/`. Public header files must be placed in a folder `include/`. Private headers may be placed in a different location.

Rule 5 [Files hierarchy] The file hierarchy should follow the namespace hierarchy. For instance, the implementation of a class `::ns1::ns2::SService` should be put in `src/ns1/ns2/SService.cpp`.

Rule 6 [Files extensions] Header files use the extension `.hpp`.

Implementation files use the extension `.cpp`.

Files containing implementation of "template" classes use the extension `.hxx`.

Recommendation 2 [Only one class per file] It is recommended to declare (or to implement) only one class per file. However tiny classes may be declared inside the same file.

Rule 7 [Includes] Use the right include directive depending on the context. `#include "..."` must be used to import headers from the same module, whereas `#include <...>` must be used to import headers from other modules.

Rule 8 [Include path] The include path is not an absolute path depending on a local file system. A correct include path does respect the letter case of the filenames and folders (since some platforms require it) and uses the character '/' as a separator.

Rule 9 [Protection against multiple inclusions] You must protect your files against multiple inclusions. To this end, use the standard directives of the precompiler `#ifndef` and `#define` (since `#pragma` once is only supported by Microsoft compilers).

Use the name of the file and the namespace hierarchy inside the define name in order to prevent any conflict with a file which has the same name but located in a different namespace. Namespaces and file name must be separated by a single underscore _. The define name must be prefixed and suffixed by two underscores __. Last, a comment must be placed after `#endif` to quote the define.

```
#ifndef __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__ // Preamble protecting against
#define __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__ // multiple inclusions.

#endif // __NAMESPACEA_NAMESPACEB_SAMPLE_HPP__
```

Recommendation 3 [Independent headers] A header should compile alone. All necessary includes should be contained inside the header itself. In the following sample :

```
// Header.hpp

class Foo
{
public:
    std::string m_string;
}
```

you will be forced to include the file in this way to get a successful build :

```
// Source.hpp

#include <string>
#include "Header.hpp"
```

This is a bad practice, the header should rather be written :

```
// Header.hpp

#include <string>

// Header.hpp
class Foo
{
public:
    std::string m_string;
}
```

So that people can simply include the header :

```
// Source.hpp

#include "Header.hpp"
```

Recommendation 4 [Minimize inclusions] Try to minimize as much as possible inclusions inside a header file. Include only what you use. Use *forward declarations* when you can (i.e. a type or class structure is not referenced inside the header). This will limit dependency between files and reduce compile time. Hiding the implementation can also help to minimize inclusions (see *Hide implementation*)

Rule 10 [Sort headers inclusions] You must sort headers in the following order : same module, framework libraries, bundles, external libraries, standard library. This way, this helps to make each header independent. The rule can be broken if a different include order is necessary to get a successful build.

```
#include "currentModule.hpp"

#include <libSampleB/second.hpp>
#include <libSampleA/first.hpp>
#include <libSampleB/subModule/first.hpp>

#include <Qt/QtGui>
#include <vector>
#include <map>
```

Recommendation 5 [Sort inclusions alphanumerically] In addition to the previous sort, you may sort includes in alphanumerical order, according to the whole path. Thus they will be grouped by module. For a better readability, an empty line can be added between each module.

```
#include "currentModule.hpp"

#include <libSampleA/first.hpp>
#include <libSampleB/second.hpp>

#include <libSampleB/subModule/first.hpp>
#include <libSampleB/subModule/second.hpp>

#include <Qt/QtGui>

#include <map>
#include <vector>
```

4.3.2 Naming conventions

Rule 11 [Class] Class names must be written in upper camel case. It should not repeat a namespace name. For instance `::editor::SCustomEditor` should be rather called `::editor::SCustom`.

Rule 12 [File] The name of the file should be based on the class name defined in it. It must follow the same letter case.

Rule 13 [Namespace] Namespaces must be written in camel case. A comment quoting the namespace must be placed next to the ending '}'.

```
namespace namespaceA
{
namespace namespaceB
{
    class Sample
    {
    ...
    };
} // namespace namespaceB
} // namespace namespaceA
```

When referring a namespace, you must put `::` if this is a root namespace, with an exception for `std` namespace. Ex: `::boost::filesystem`.

Rule 14 [Function and method names] Functions and methods names must be written in camel case.

Recommendation 6 [Correct naming of functions] Try as much as possible to help the users of your code by using comprehensive names. You may for instance help them to indicate the cost of a function. A function that executes a search to retrieve an object must not be called like a getter. In this case, it is better to call it `findObjet()` instead of `getObject()`.

Rule 15 [Variable] Variable names must be written in camel case. Members of a class are prefixed with a `m_`.

```
class SampleClass
{
private:
   int m_identifier;
   float m_value;
};
```

Static variables are prefixed with a `s_`.

```
static int s_staticVar;
```

Rule 16 [Constant] Constant variables must be written in snake_case but in capitals, and follow the previous rule.

```
class SampleClass
{
    static const int s_AAA_BBB_CCC_VALUE = 1;
};

void fooFunction()
{
    const int AAA_BBB_VAR = 1;
    ...
}
```

Rule 17 [Type] Type names, like classes, must be written in upper camel case.

```
typedef int CustomType;
typedef vector<int> CustomContainer;
```

Rule 18 [Template parameter] Template parameters must be written in capitals. In addition, they must be short and explicit.

```
template< class KEY, class VALUE > class SampleClass
{
    ...
};
```

Rule 19 [Macro] Macros without parameters must be written in capitals. On the contrary, there is no specific rule on macros with parameters.

```
#define CUSTOM_FLAG_A 1
#define CUSTOM_FLAG_B 1

#define CUSTOM_MACRO_A( x ) x
#define Custom_Macro_B( x ) x
#define custom_Macro_C( x ) x
#define custom_macro_d( x ) x
```

Rule 20 [Enumerated type] An enumerated type name must be written in upper camel case. Labels must be written in capitals. If a `typedef` is defined, it follows the upper camel case standard.

```
typedef enum SampleEnum
{
    LABEL_1,
    LABEL_2
    ...
} SampleEnumType;
```

Rule 21 [Service] A service implementation is identified by a `S` at the beginning of the class name. Example : `SCustomEditor`. A service interface is identified by a `I` at the beginning of the class name. Example : `IEditor`.

Rule 22 [Signal] A signal name must be prefixed with `sig`. It should be suffixed by a past action (ex: Updated, Triggered, Cancelled, CakeCookedAndBaked). It follows other common variable naming rules (member of a class, etc...).

```
class Sample
{
    SigType::sptr m_sigImageDisplayed;
};
```

Rule 23 [Slot] A slot name must be prefixed with `slot`. It should be suffixed by an imperative order (Ex: Update, Run, Detach, Deliver, OpenWebBrowser, GoToFail). It follows other common variable naming rules (member of a class, etc...).

```
class Sample
{
    SlotType::sptr m_slotDisplayImage;
}
```

4.3.3 Coding rules

Blocks

Rule 24 [Indentation] Code block indentation and bracket positioning follow the Allman style.

```
void function(void)
{
    if(x == y)
    {
        something1();
        something2();
    }
    else
    {
        somethingElse1();
        somethingElse2();
    }
    finalThing();
}
```

Rule 25 [Indentation of namespaces] Namespaces are an exception of the previous rule. They should not be indented.

```
namespace namespaceA
{
namespace namespaceB
{
    ...
} // namespace namespaceB
} // namespace namespaceA
```

Rule 26 [Blocks are mandatory] After a control statement (if, else, for, while/do...while, try/catch, switch, foreach, etc...), it is mandatory to open a block, whatever is the number of instructions inside the block.

Rule 27 [Scope] The keywords `public`, `protected` and `private` are not indented, they should be aligned with the keyword `class`.

```
class Sample
{
public:
    ...
private:
    ...
};
```

Class declaration

Recommendation 7 [Only three scope sections] When possible, use only one section of each scope type `public`, `protected` and `private`. They must be declared in this order.

Recommendation 8 [Group class members by type] You may group class members in each scope according to their type: type definitions, constructors, destructor, operators, variables, functions.

Rule 28 [Hide implementation] Avoid non-const public member variables except in very small classes (i.e. a 3D point). The Pimpl idiom may also be helpful to separate the implementation from the declaration.

Recommendation 9 [Hide implementation] Try to put variables as much as possible in the `private` section.

Rule 29 [Accessors] Since you protect your member variables from the outside, you will have to write accessors, named `getXXX()` and `setXXX()`. Getters are always `const`.

Rule 30 [Template class function definition] The function definition of a template class must be defined after the declaration of the class.

```
template < typename TYPE >
class Sample
{
public:
    void function(int i);
};

template < typename TYPE >
inline Sample<TYPE>::function(int i)
{
    ...
}
```

Recommendation 10 [Separate template class function definition] In addition of the previous rule, you may put the definition of the function in a `.hxx` file. This file will be included in the implementation file right after the header file (the compile time will be reduced comparing with an inclusion of the `.hxx` in the header file itself).

```
#include <namespaceA/file.hpp>
#include <namespaceA/file.hxx>
```

Initializer list

Rule 31 [One initializer per line] In a class constructor, use the initialization list as much as possible. Place one initializer per line. Constructors of base classes should be placed first, followed by member variables. Do not specify an initializer if it is the default one (empty std::string for instance).

```
SampleClass::SampleClass( const std::string& name, const int value ) :
    BaseClassOne( name ),
    BaseClassTwo( name ),
    m_value( value ),
    m_misc( 10 )
{}
```

Recommendation 11 [Align everything that improves readability] To improve readability, you may align members on one hand and argument lists on the other hand.

```
SampleClass::SampleClass( const std::string& name, const int value ) :
    BaseClassOne  ( name ),
    BaseClassTwo  ( name ),
    m_value       ( value ),
    m_misc        ( 10 )
{}
```

Functions

Rule 32 [Constant reference] Whenever possible, use constant references to pass arguments of non-primitive types. This avoids useless and expensive copies.

```
void badFunction( std::vector<int> array )
{
    ...
}

void goodFunction( const std::vector<int>& array )
{
    ...
}
```

Recommendation 12 [Constant reference for shared pointers] For performance sake, it is preferable to use `const&` to pass arguments of type `::boost::shared_ptr`. It is only useful to pass the pointer by copy if the pointer can be invalidated by an another thread during the function call. If you have any doubt, it is safer to pass the argument by copy.

Rule 33 [Constant functions] Whenever a member function should not modify an attribute of a class, it must be declared as `const`.

```
void readOnlyFunction( const std::vector<int>& array ) const
{
    ...
}
```

Recommendation 13 [Limit use of expression in arguments] When passing arguments, try to limit the use of expressions to the minimum.

```
// This is bad
function( fn1(val1 + val2 / 4 ), fn2( fn3( val3 ), val4) );

// This is better
```

```
    const float res0 = val1 + val2 / 4;

    const float res1 = fn1(res0);
    const float res3 = fn3(val3);
    const float res2 = fn2(res3, val4);

    function( res1 , res2 );
```

Miscellaneous

Rule 34 [Enumerator labels] Each label must be placed on a single line, followed by a comma. If you assign values to labels, align values on the same column.

```
    enum OpenFlag
    {
        OPEN_SHARE_READ       = 1,
        OPEN_SHARE_WRITE      = 2,
        OPEN_EXISTING         = 4,
    };
```

Rule 35 [Use of namespaces] You have to organize your code inside namespaces. By default, you will have at least one namespace for your module (application or bundle). Inside this namespace, it is recommended to split your code into sub-namespaces. This helps notably to prevent naming conflicts.

It is forbidden to use the expression‘‘using namespace‘‘ in header files but it is allowed in implementation files. It is however recommended to use aliases in this latter case.

```
    namespace bf = ::boost::filesystem;
```

Rule 36 [Keyword const] Use this keyword as much as possible for variables, parameters and functions.

Recommendation 14 [Keyword auto] Use this keyword as much as possible to improve maintainability and robustness of the code.

Rule 37 [Prefer constants instead of #define] Use a static constant object or an enumeration instead of a `#define`. This will help the compiler to make type checking. You will also be able to check the content of the constants while debugging. You can also define a scope for them, inside the namespace, inside a class, private to a class, etc...

Rule 38 [Prefer references over pointers] When possible, use references instead of pointers, especially for function parameters. Pointer as parameter should only be used if it is considered to have a NULL pointer or when passing a C-like array. If you use a pointer, always check it if is null in the current scope before dereferencing it.

Rule 39 [Type conversion] For type conversion, use the C++ operators which are `static_cast`, `dynamic_cast`, `const_cast` and `reinterpret_cast`. Use them wisely in the appropriate case. You may read this documentation.

Recommendation 15 [Strings to numbers/numbers to string conversion] When converting strings to numbers or numbers to string, prefer the use of boost::lexical_cast.

Recommendation 16 [Exceptions] Exceptions are the preferred mechanism to handle error notifications.

Rule 40 [Explicit integer types] When you do need a specific integer size, use type definitions declared in <cstdint>, for example :

Bits	Signed	Unsigned
8	int8_t	uint8_t
16	int16_t	uint16_t
32	int32_t	uint32_t
64	int64_t	uint64_t

4.4 Documentation

Rule 41 [Document the code] The code must be documented with **Doxygen**, an automated tool to generate documentation.

Rule 42 [Location of the documentation] Every documentation that can be useful to a user must be placed inside the header files. Thus a user of a module can find the declaration of a class and its documentation at the same place. Inside the implementation file, the documentation will give more details about algorithms. Moreover, every documentation must be placed next to the entity it is refering to, in order to help searching inside the code.

Recommendation 17 [Lightweight documentation] Inside a documentation block, only use necessary tags. This will avoid to overload the documentation and makes it readable. By the way, empty tags will be presented inside the generated documentation and will be useless. Just use an empty line to make a separation inside a documentation block. Don't indicate parameter types when using `@param` directive. This is useless since it will duplicate information of the function prototype. Also, prefer the use of `///` whenever possible.

Example 1 : Bad documentation block

```
/**
* @brief        A very short description.
*
* A longer description, giving more details about the documented piece
* of code.
*********************************************
* @param
*********************************************
* @return
*********************************************
* @exception
*********************************************
* @todo
*********************************************
```

Example 2 : Good documentation block

```
/**
* @brief        A very short description.
*
* A longer description, giving more details about the documented piece
* of code.
*/
```

Example 3 : Function documentation

```
class Sample
{
public:
    /**
    * Retrieve the thing.
    *
    * @return       The thing value.
    */
    const std::string& getThing( void ) const;
    /**
    * @brief        Set the thing.
    *
    * @param        thing   :  The new thing.
    */
    void setThing( const std::string& thing );

private:
    /// stored thing
    std::string     m_thing;
};
```

Recommendation 18 [Structured documentation] Doxygen provides a default structure when you generate the documentation. However, when dealing with a big documented entity, it is often recommended to use the group feature (`@name`). With this feature you will build a logical view of the class interfaces.

Rule 43 [Document service configuration] The method `configuring` of a service must be properly documented. It should indicate every parameter that can be passed, no matter if it is optional or not. Example :

```
/**
 * @verbatim
<adaptor id="points" class="::namespace::SService">
    <config option1="default" option2="false"/>
</adaptor>
 @endverbatim
 * - \b option1 : first option.
 * - \b option2(optional) : second option.
*/
NAMESPACE_API void configuring() throw(fwTools::Failed);
```

4.5 XML coding

Rule 48 [Id name] Id should have a semantic name. Avoid id like myXXXXX or customXXXXX. Moreover, id must be written in lower case with an underscore as separator.

```
<service id="generic_scene" />
```

4.6 CMakeLists coding

Rule 1 [Function name] Standard CMake functions and macros should be written in lower case. Each word is generally separated by an underscore (this is a rule of CMake anyway).

```
add_subdirectory("library/")
include_directories(SYSTEM "/usr/local")
```

Rule 2 [Macro name] Custom macros should be written in camel case.

```
fwLoadProperties()
fwLink("boost")
```

Rule 3 [Variable name] Variables should be written in upper case letters separated if needed by underscores.

```
set(VARIABLE_NAME "")
```

Recommendation 1 [Expression in block ending] In the past, CMake enforced to specify the label or expression in block ending, for instance :

```
function(name arg1 arg2)
    ...
    if(expr1)
        ...
    else(expr1)
        ...
    endif(expr1)
    ...
endfunction(name)
```

This is no longer needed in latest CMake versions, and we recommend to use this possibility for the sake of simplicity.

```
function(name arg1 arg2)
    ...
    if(expr1)
        ...
    else()
        ...
    endif()
    ...
endfunction()
```

4.7 Licence

Rule 47 [LGPL] Do not forget to put the LGPL licence block on fw4spl.

```
/* ***** BEGIN LICENSE BLOCK *****
 * FW4SPL - Copyright (C) IRCAD, 2009-2015.
 * Distributed under the terms of the GNU Lesser General Public License (LGPL) as
 * published by the Free Software Foundation.
 * ***** END LICENSE BLOCK ***** */
```

CHAPTER 5

Frequently Asked Questions (FAQ)

5.1 What is fw4spl?

The framework FW4SPL (FrameWork for Software Production) is an open-source framework, developed by IRCAD (research institute against cancer and disease). The principle of FW4SPL is the fast and easy creation of applications, mainly in the medical field. Therefore it provides features like digital image processing in 2D and 3D, visualization or simulation of medical interactions. To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

5.2 What does fw4spl mean?

FW4SPL means FrameWork for Software Production Line. It is also called F4S ("forces").

5.3 What are the features of fw4spl?

The framework is built around the notion of component (bundle). To build an application with FW4SPL there are no programming skills required. By writing a simple XML the users can design their own application.

FW4SPL has a component-based architecture composed of C++ libraries. There are three main concepts in the architecture: - object-service concept - component approach - signal-slot communication

5.4 Which platforms does fw4spl run on?

This framework can run under Windows, Linux and MacOS and we are working on the Android part.

5.5 Where can I find applications developed with fw4spl ?

Some tutorials are provided with the framework and you can also build VR-Render, a free visualization software.

5.6 Which prerequiste do I need to develop bundle?

You must have a good knowledge in C++. Concerning the configuration files, they are written in XML.

5.7 What are the BinPkgs?

The BinPkgs (binary packages) contain all the extern libraries needed by fw4spl. For each BinPkg, a CMakeLists provides the OS specific instructions to build it . They can be downloaded on https://github.com/fw4spl-org/fw4spl-deps

5.8 Is it difficult to compile an application with fw4spl?

No, it isn't. You just have to compile all the bundles and libraries used by the application.

5.9 Why does fw4spl provide a launcher?

The launcher is used to create the entry point of the application. It parses the profile and configuration xml file to build it.

5.10 How can I debug my program ?

First, you can change the log level of a sub-project in the CMake configuration.

The allowed values are : ['trace', 'debug', 'info', 'error', 'fatal', 'warning', 'disable']. the value 'trace' gives me the maximun of log, 'disable' disables log.

> **note a** : Printing many log messages (by activating trace on all sub-projects for ex.) can be very time consuming for the application.
>
> > **note b** : Under windows system, log messages are saved on filesystem in SLM.log file, in the working directory.

Secondly, you can debug your application using gdb (Linux/Mac) or Visual Studio (Windows) and compiling your application in Debug mode

> **note a** : you can use gdb like this "LD_LIBRARY_PATH=./lib gdb -arg bin/launcher Bundles/myApp/myProfile.xml", and press "r" for run the program
>
> **note b** : you can use under gdb the command "catch throw" hence gdb will stop near the error **note c** : Documentation to learn using gdb : http://www.cs.tau.ac.il/lin-club/lecture-notes/GDB_Linux_telux.pdf

Thirdly, you can manage the program complexity by reducing the number of activated components (in profile.xml) and the number of created services (in config.xml) to better localize errors.

Fourthly, verify that your profile.xml / plugin.xml and each bundle plugin.xml are well-formed, by using xmllint (command line tool provided by libxml2).

5.11 I have an assertion/fatal message when I launch my program, any idea to correct the problem ?

First, you can read the output message :) and try to solve the problem. In many cases, there are two kind of problems. The program fails to :

- **create the service given in configuration In this case, four reasons are possibles :**

  - the name of service implementation in config.xml contains mistakes
  - the bundle that contains this service is not activated in the profile
  - the bundle plugin.xml, that contains this service, not declares the service or the declaration contains mistakes.
  - the service is not register in the Service Factory (forget of macro REGISTER_SERVICE(...) in file .cpp)

- manage the configuration of service. In this case, the implementation code in .cpp file (generally configuring() method of service) does not correspond to description code in config.xml (Missing arguments, or not well-formed, or mistakes string parameters).

5.12 If I use fw4spl, do I need wrap all my data ?

The first question is to know if the data is on center of application:

- Need you to shared data between few bundles ?
- Need you to attach services on this data ?
  - If the answer is no, you don't need to wrap your data.
  - Otherwise, you need to have an object that inherits of ::fwData::Object.

In this last case, do you need shared this object between different services which use different libraries, ex for Object Image : itk::Image vs vtkImage ?

- If the answer is yes, you need create a new object like fwData::Image and a wrapping with fwData::Image<=>itk::Image and fwData::Image<=>vtkImage.
- Otherwise, you can just encapsulated an itk::Image in fwData::Image and create an accessor on it. (however, this kind of choice implies that all applications that use fwData::Image need itk library for running.)

5.13 What is a sesh@ path ?

A sesh@ path is a path used to browse an object (and sub-object) using the introspection (see fwDataCamp and *Serialization*). The path begins with a '@' or a '!'. - `@` : the returned string is the fwID of the sub-object defined by the path. - `!` : the returned string is the value of the sub-object, it works only on String, Integer, Float and Boolean object.

5.13.1 Example:

To get the fwID of an image contained in a Composite with the key "myImage"

```
@values.myImage
```

To get the fwID of the first reconstruction of a ModelSeries contained in a Composite with the key "myModel"

```
@values.myModel.reconstruction_db.0
```

CHAPTER 6

How to use CMake with Fw4spl

6.1 CMake for fw4spl

6.1.1 Introduction

Fw4spl and it's dependencies are based on CMake . Note that the minimal version of cmake to have is 2.8.12.

6.1.2 CMake files for dependencies

fw4spl dependencies are based on the ExternalProject concept from lastest versions of cmake.

The concept is to create custom targets to build projects in external trees. Each project has custom steps for download, update/patch, configure, build and install.

Here is a simple example from camp :

```
cmake_minimum_required(VERSION 2.8)

project(campBuilder)

include(ExternalProject)

set(CAMP_CMAKE_ARGS ${COMMON_CMAKE_ARGS}
                    -DBUILD_DOXYGEN:BOOL=OFF
                    -DBOOST_INCLUDEDIR:PATH=${CMAKE_INSTALL_PREFIX}/include/boost-1_57
)

getCachedUrl(https://github.com/greenjava/camp/archive/0.7.1.1.tar.gz CACHED_URL)

ExternalProject_Add(
    camp
    URL ${CACHED_URL}
    DOWNLOAD_DIR ${ARCHIVE_DIR}
    DEPENDS boost
    INSTALL_DIR ${CMAKE_INSTALL_PREFIX}
    CMAKE_ARGS ${CAMP_CMAKE_ARGS}
    STEP_TARGETS CopyConfigFileToInstall
)

ExternalProject_Add_Step(camp CopyConfigFileToInstall
    COMMAND ${CMAKE_COMMAND} -E copy ${CMAKE_SOURCE_DIR}/cmake/findBinpkgs/FindCAMP.cmake ${CMAKE_INS
    COMMENT "Install configuration file"
```

The important parts are in the *ExternalProject_Add* fonction:

- URL: is the download link of the sources
- DOWNLOAD_DIR: The folder where the sources will be stored (set globaly for all deps)
- DEPENDS: The dependencies of the current library (will be compiled before)
- INSTALL_DIR: The folder in which the library will be installed (set globaly for all deps)
- CMAKE_ARGS: CMake options for library which have a cmake build system
- STEP_TARGETS: Custom command (in this example it will copy a script in the install folder)

Note that in other script you can have much more options like:

- PATCH_COMMAND
- CONFIGURE_COMMAND
- BUILD_COMMAND
- INSTALL_COMMAND

Refer you to the documentation of ExternalProject for more informations.

6.1.3 CMake files for fw4spl

Each project (apps, bundles, libs) have two "CMake" files:

- CMakeLists.txt
- Properties.cmake

The CMakeLists.txt file

The CMakeLists.txt should contain at least the function *fwLoadProperties()* to load the Properties.cmake. But it can also contain others functions usefull to link with external libraries.

Here is an example of CMakeLists.txt from guiQt Bundle :

```
fwLoadProperties()
fwUseForwardInclude(
    fwActivities
    fwGuiQt
    fwRuntime
    fwServices
    fwTools

    gui
)

find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)


fwForwardInclude(
    ${Qt5Core_INCLUDE_DIRS}
    ${Qt5Gui_INCLUDE_DIRS}
    ${Qt5Widgets_INCLUDE_DIRS}
)
```

```
fwLink(
        ${Qt5Core_LIBRARIES}
        ${Qt5Gui_LIBRARIES}
        ${Qt5Widgets_LIBRARIES}
)

set_target_properties(${FWPROJECT_NAME} PROPERTIES AUTOMOC TRUE)
```

The first line *fwLoadProperties()* will load the properties.cmake (see explanation in the next section). The *fwUseForwardInclude(...)* function will add the include directories of each argument to the target.

The next lines are for the link with an external libraries (fw4spl-deps), in this example it is Qt.

The first thing to do is to call *find_package(The_lib COMPONENTS The_component)*.

The use *fwForwardInclude* to add includes directories to the target, and *fwLink* to link the libraries with your target.

You can also add custom properties to your target with *set_target_properties*.

The Properties.cmake file

Properties.cmake should contain informations like name, version, dependencies and requirements of the current target.

Here is an example of Properties.cmake from fwData library:

```
set( NAME fwData )
set( VERSION 0.1 )
set( TYPE LIBRARY )
set( DEPENDENCIES fwCamp fwCom fwCore fwMath fwMemory fwTools )
set( REQUIREMENTS  )
```

- NAME: Name of the target
- VERSION: Version of the target
- TYPE: Type of the target (can be library, bundle or executable)
- DEPENDENCIES: Link the target with the given libraries (see target_link_libraries)
- REQUIREMENTS: Ensure that the depends are build before target (see add_dependencies)

6.2 Tutorials

6.2.1 How can I add a new dependency

You may want to add a new dependency into fw4spl-deps or you may want to add your own folder of dependencies.

Tip: You need to know that the main CMakeLists.txt is in fw4spl-deps, and you can add as many additional folders as you want. Use the *ADDITONNAL_DEPS* option in cmake to set the path of your custom deps.

Add a new deps in fw4spl-deps

Adding a new deps is quite easy, the only things to do is to add a new folder *myNewDeps* and put a CMakeLists.txt file into it. The CMakeLists.txt should contain at least:

- cmake_minimum_required()
- project()
- include(ExternalProject)
- ExternalProject_Add(...)

For example:

```
cmake_minimum_required(VERSION 2.8)

project(myDepsBuilder)

include(ExternalProject)

getCachedUrl(http://myDeps.com/myDeps.zip CACHED_URL)

ExternalProject_Add(
    myDeps
    URL ${CACHED_URL}
    DOWNLOAD_DIR Path/To/Your/Download/dir
    PATCH_COMMAND your_patch_command (optional)
    CONFIGURE_COMMAND your_configure_command (optional)
    BUILD_COMMAND  your_build_command (optional)
    INSTALL_COMMAND your_install_command (optional)
    INSTALL_DIR your_install_dir
    CMAKE_ARGS cmake_arguments
)
```

Add a custom deps repository

You may want to add your own folder of dependencies (as fw4spl-ext-deps or fw4spl-ar-deps). In this case your main need to create a CMakeLists.txt in the root of your folder (myDepsFolder/CMakeLists.txt) in order to list the subdirectories of your deps.

```
cmake_minimum_required(VERSION 2.8)

project(CustomDeps)

list(APPEND SUBDIRECTORIES myDeps1)
list(APPEND SUBDIRECTORIES myDeps2)
...
```

Then when you do a *ccmake* or *cmake-gui* in the build of your deps, you need to add the path to your custom repository in the *ADDITONNAL_DEPS* option. Then cmake will automaticaly parsed your folder.

6.2.2 How can I add a custom bundle in fw4spl

You may want to add a new bundle/lib/app in an existing repository or you may want to add your custom repository to fw4spl.

Tip: You need to know that the main CMakeLists.txt is in fw4spl repository, and you can add as many additional repository as you want. Use the *ADDITIONAL_PROJECTS* option in cmake to add path of your custom folders.

Add a new bundle/lib/app in fw4spl

The only thing to do is to write a CMakeLists.txt and a Properties.cmake (see section Cmake for Fw4spl for more informations).

Add a custom repository to fw4spl

As the main CMakeLists.txt is in fw4spl repository,you need to add the path of your folder in *ADDITIONAL_PROJECTS* option when you launch *ccmake* of *cmake-gui* on the build folder of fw4spl. Then your folder will automaticaly be parsed by cmake.

Note: All your bundle/lib/application need to respect the fw4spl-cmake conventions and have a CMakeLists.txt and a Properties.cmake.

CHAPTER 7

Contributors

7.1 Contributors

From 2004 to 2006, an advanced modular software for patient modeling (see publication page) has been designed and implemented by Guillaume Brocker, Johan Moreau, Jean-Baptiste Fasquel, Vincent Agnus and Nicolas Papier. This represented the basis of the component management system of FW4SPL, essentially conceived by Guillaume Brocker and Johan Moreau. This framework version (v0.1) was used to create 3 software tools in visualization and medical image processing in the Eureka project Odysseus (3DVPM, 3DDVP and MARNS software).

Throughout 2007, Vincent Agnus and Jean-Baptiste Fasquel conceived and implemented the main core mechanisms of this new version of FW4SPL. Jean-Baptiste Fasquel focused on the notion of roles coupled with the component management system, the inter-role communication system, as well as an appropriate XML formalism for the description of both roles embedded into components and description of software. Many basic software tools have been built to validate the architecture (see publication page). Vincent Agnus also focused on role design, and more specifically on data structures, a generic serialization mechanism and a powerful template dispatching technique. During his internship in 2007, Benjamin Gaillard has improved the communication system in FW4SPL. In parallel with the work on the pure FW4SPL system, Johan Moreau got involved in the construction/compilation system and, together with Arnaud Charnoz, in the management of external dependencies and some specific medical data structures. Their work also led to advanced visualization of medical images (free download). Early 2008, the framework was available in version 0.2.

During the period from mid-2008 to mid-2009, some advanced data structures and functionalities have been developed on the basis of the architecture to further evaluate it and make it more robust. A larger development team has been involved, including Emilie Harquel, Julien Waechter and Nicolas Philipps additionally to Vincent Agnus, Jean-Baptiste Fasquel, Johan Moreau and Arnaud Charnoz. Additional efforts have been made by Johan Moreau and Arnaud Charnoz on the management of external dependencies. Nicolas Philipps, Julien Waechter and Johan Moreau also improved the construction environment Sconspiracy, initially opened as an opensource project YAMS++ in 2007. The version 0.3 of the framework had been achieved by early summer 2009.

From mid-2009 to mid-2010, the main work on FW4SPL included: performing generic scenes for visualization (mainly developed by Nicolas Philipps, Julien Waechter and Vincent Agnus), a new communication system (mainly developed by Nicolas Philipps and Arnaud Charnoz), new UI components (mainly developed by Emilie Harquel and Julien Waechter), better log and assert system (by Arnaud Charnoz), new documentation (mainly done by Pascal Monnier, Alexandre Hostettler).

FW4SPL (version 0.4) has been opened late 2009 and was used to create several software in the European project Passport (VR-Render, VR-Render WLE, AR-Surg, VR-Planning and VR-Probe software). In December, we had switched to version 0.5 (with generic scene). The latest stable version is 0.6 (new communication system) and the current branch development is the 0.6.1 branch.

Version 0.7 adds a limited Qt support during summer 2010 (Hocine Chekatt's internship) and limited support for Python, OpenNI and SOFA (these two last parts had been developed by Altran). During 2011, FW4SPL 0.8 adds a Qt based 2D scene (Ivan MATHIEU's internship), new buffer for meshes and images, new memory dump mechanisms, a new set of applications (Apps/Examples), a new Dicom reader (Jordi ROMERA's internship), new registration functionalities (Marc Schweitzer's internship) an improved image origin management, etc. A new scenegraph design has been developed but not yet integrated (Loïc Velut's internship).

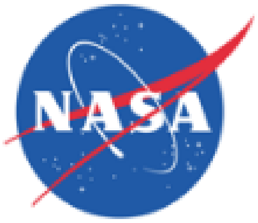
Multithreading (fwThread), signal/slot (fwCom), dump management and data introspection (fwAtoms) mechanisms have been added during 2012 in version 0.9 (co-working between IRCAD and IHU). A new design to manage data and store data (Julien Weinzorn's internship) has been prototyped.

This version supports msvc2010 and has also been used to evaluate the transition to Android and iOS (Adrien Bensaibi's internship). Altran has added a connector towards the management tool of the MIDAS content developed by Kitware. Finally, a version management mechanism has been developed (fwAtomsPatch) (Clément Troesch's internship) and new data has been created (fwMedData). This version has been used by the Visible Patient company within the framework of their ISO 13485 certification. A new repository has also been created (fw4spl-ext) with the aim of welcoming not yet stabilized functionalities or to host PoC. The CMake construction system is also supported.

Version 0.10.0 provides the notion of timeline to manage temporal data (IHU). The SConspiracy construction system has been removed.

ircad France	Core, visualization, image processing, applications and tutorials **Team** [Johan Moreau, Marc Schweitzer, Frédéric CHAMP,] Flavien Bridault-Louchez, Pascal Monnier
ihu STRASBOURG	Core, visualization, image processing, applications and tutorials Team : Julien Waechter, Emilie Harquel, Jessica GROMER
altran	**Proof of concept on Kinect and Sofa integration** • Altran_200609_MAG10_FR.pdf French document p12 • Altitude_17_20100407_FR.pdf French document p26 Proof of concept on MIDAS integration
	Team : Nicolas Philipps, Valentin Martinet, Arnaud Charnoz, Julien Weinzorn
	This project has partly funded by the European Commission via PASS • http://www.passport-liver.eu/ • http://www.vph-noe.eu/vph-repository/doc_download/154-passportppt • newsletter july 2010

7.2 Libraries

7.3 FLOSS projects using FW4SPL

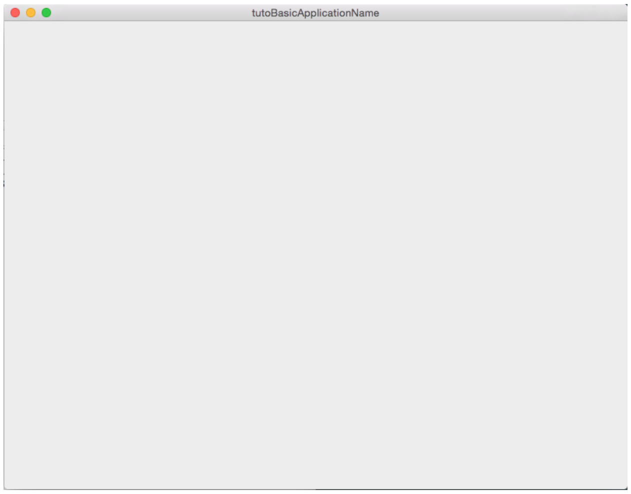
skuld-project	Skuld project work on mobile port (iphone, android, meego/maemo, ...) of FW4SPL

CHAPTER 8

Tutorials

8.1 [*Tuto01Basic*] Create an application

The first tutorial represents a basic application that launch a simple empty frame.



8.1.1 Prerequisites

Before to read this tutorial, you should have seen :

- *Object-service concept*
- *App-config*
- *Component-based software*

8.1.2 Structure

An application is organized around three main files :

- CMakeLists.txt
- Properties.cmake
- plugin.xml

CMakeLists.txt

The CMakeLists is parsed by CMake. For the aplication it should contain the line :

```
fwLoadProperties()
```

This line allows to load Properties.cmake file.

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto01Basic ) # Name of the application
set( VERSION 0.1 ) # Version of the application
set( TYPE APP ) # Type APP represent "Application"
set( DEPENDENCIES  ) # For an application we have no dependencies (libraries to link)
set( REQUIREMENTS # List of the bundles used by this application
    dataReg # to load the data registry
    servicesReg # to load the service registry
    gui # to load gui
    guiQt # to load the Qt implementation of gui
    launcher # executable of the application
    appXml # to parse the application configuration
)

# Set the configuration to use : 'tutoBasicConfig'
bundleParam(appXml PARAM_LIST config PARAM_VALUES tutoBasicConfig)
```

This file contains the minimal requirements to launch an application with a Qt user interface.

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<!-- Application name and version (the version is automatically replaced by CMake
     using the version defined in the Properties.cmake) -->
<plugin id="Tuto01Basic" version="@DASH_VERSION@">

    <!-- Defines the App-config -->
    <extension implements="::fwServices::registry::AppConfig">
        <id>tutoBasicConfig</id><!-- identifier of the configuration -->
        <config>
            <object type="::fwData::Image"><!-- Main object -->

                <!-- Frame service -->
                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>tutoBasicApplicationName</name>
                            <icon>Bundles/Tuto01Basic_0-1/tuto.ico</icon>
                            <minSize width="800" height="600" />
                        </frame>
                    </gui>
                </service>

                <start uid="myFrame" /><!-- start the frame service -->

            </object>
        </config>
    </extension>
</plugin>
```

The `::fwServices::registry::AppConfig` extension defines the configuration of an application.

id: The configuration identifier.

config: Contains the list of objects and services used by the application.

For this tutorial, we have only one object `::fwData::Image` and one service `::gui::frame::DefaultFrame`.

The order of the elements in the configuration is important:

- <service> tags are into <object> tags
- <start> tags are after <service> tags

There are others tags that will be described in the next tutorials.

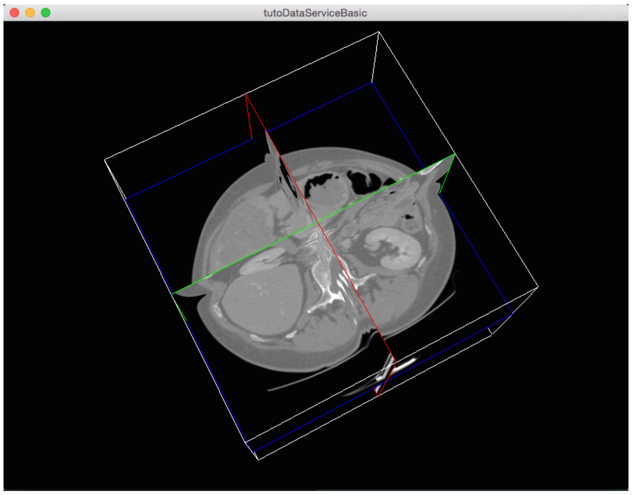
8.1.3 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto01Basic_0-1/profile.xml
```

8.2 [*Tuto02DataServiceBasic*] Display an image

The secons tutorial represents a basic application that display a medical 3D image.



8.2.1 Prerequisites

Before to read this tutorial, you should have seen :

- *[Tuto01Basic] Create an application*

8.2.2 Structure

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto02DataServiceBasic )
set( VERSION 0.1 )
set( TYPE APP )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io # contains the interface for reader and writer.
    ioVTK # contains the reader and writer for VTK files (image and mesh).
    visuVTK # loads VTK rendering library (fwRenderVTK).
    visuVTKQt # containsthe vtk Renderer window interactor manager using Qt.
    vtkSimpleNegato # contains a visualization service of medical image.
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES tutoDataServiceBasicConfig)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<plugin id="Tuto02DataServiceBasic" version="@DASH_VERSION@">

    <extension implements="::fwServices::registry::AppConfig">
        <id>tutoDataServiceBasicConfig</id>
        <config>

            <!-- In tutoDataServiceBasic, the central data object is a ::fwData::Image. -->
            <object type="::fwData::Image">

                <!--
                    Description service of the GUI:
                    The ::gui::frame::SDefaultFrame service automatically positions the various
                    containers in the application main window.
                    Here, it declares a container for the 3D rendering service.
                -->
                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>tutoDataServiceBasic</name>
                            <icon>Bundles/Tuto02DataServiceBasic_0-1/tuto.ico</icon>
                            <minSize width="800" height="600" />
                        </frame>
                    </gui>
                    <registry>
                        <!-- Associate the container for the rendering service. -->
                        <view sid="myRendering" />
                    </registry>
                </service>

                <!--
                    Reading service:
                    The <file> tag defines the path of the image to load. Here, it is a relative
                    path from the repository in which you launch the application.
                -->
                <service uid="myReaderPathFile" impl="::ioVTK::SImageReader">
                    <file>./TutoData/patient1.vtk</file>
                </service>

                <!--
                    Visualization service of a 3D medical image:
                    This service will render the 3D image.
                -->
                <service uid="myRendering" impl="::vtkSimpleNegato::SRenderer" />

                <!--
                    Definition of the starting order of the different services:
                    The frame defines the 3D scene container, so it must be started first.
                    The services will be stopped the reverse order compared to the starting one.
                -->
                <start uid="myFrame" />
                <start uid="myReaderPathFile" />
                <start uid="myRendering" />

                <!--
                    Definition of the service to update:
                    The reading service load the data on the update.
                    The render update must be called after the reading of the image.
                -->
                <update uid="myReaderPathFile" />
                <update uid="myRendering" />

            </object>

        </config>
    </extension>

</plugin>
```

For this tutorial, we have only one object `::fwData::Image` and three service:

- `::gui::frame::DefaultFrame`: frame service
- `::ioVTK::ImageReaderService`: reader for 3D VTK image

- `::vtkSimpleNegato::SRenderer`: render for 3D image

Note: To avoid the `<start uid="myRendering" />`, the frame service can automatically start the rendering service: you just need to add the attribute `start="yes"` in the <view> tag.

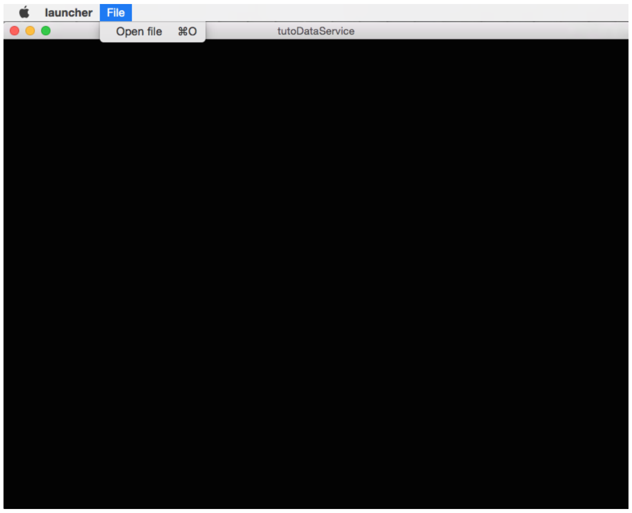
8.2.3 Run

To run the application, you must call the following line in the install or build directory:

```
bin/launcher Bundles/Tuto02DataServiceBasic_0-1/profile.xml
```

8.3 [*Tuto03DataService*] Display an image with menu

The third tutorial is similar to the previous application, but we add gui service like menus.



8.3.1 Prerequisites

Before to read this tutorial, you should have seen :

- *[Tuto02DataServiceBasic] Display an image*
- *Graphical User Interface*

8.3.2 Structure

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto03DataService )
set( VERSION 0.1 )
set( TYPE APP )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io
    ioVTK
    uiIO # contains services to show dialogs for reader/writer selection
    visuVTK
    visuVTKQt
    vtkSimpleNegato
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES tutoDataServiceConfig)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<plugin id="Tuto03DataService" version="@DASH_VERSION@">
    <requirement id="servicesReg" />

    <extension implements="::fwServices::registry::AppConfig">
        <id>tutoDataServiceConfig</id>
        <config>

            <!-- The root data object in tutoDataService is a ::fwData::Image. -->
            <object type="::fwData::Image">

                <!-- Frame service:
                    The frame creates a container fot the rendering service and a menu bar.
                    In this tutorial, the gui services will automatically start the services they reg
                    'start="yes"' attribute.
                -->
                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>tutoDataService</name>
                            <icon>Bundles/Tuto03DataService_0-1/tuto.ico</icon>
                            <minSize width="800" height="600" />
                        </frame>
                        <menuBar />
                    </gui>
                    <registry>
                        <menuBar sid="myMenuBar" start="yes" />
                        <view sid="myRendering" start="yes" />
                    </registry>
                </service>

                <!--
                    Menu bar service:
                    This service defines the list of the menus displayed in the menu bar.
                    Here, we have only one menu: File
                    Each <menu> declared into the <layout> tag, must have its associated <menu> into
                    The <layout> tags defines the displayed information, whereas the <registry> tags
                    services information.
                -->
                <service uid="myMenuBar" impl="::gui::aspect::SDefaultMenuBar">
                    <gui>
                        <layout>
                            <menu name="File" />
                        </layout>
                    </gui>
                    <registry>
                        <menu sid="myMenu" start="yes" />
                    </registry>
                </service>

                <!--
                    Menu service:
                    This service defines the actions displayed in the "File" menu.
                    Here, it registers two actions: "Open file", and "Quit".
                    As in the menu bar service, each <menuItem> declared into the <layout> tag, must
                    associated <menuItem> into the <registry> tag.

                    It's possible to associate specific attributes for <menuItem> to configure their
                    In this tutorial, the attribute 'specialAction' has the value "QUIT". On MS Windo
                    impact, but on Mac OS, this value installs the menuItem in the system menu bar, a
                    value installs the default 'Quit' system icon in the menuItem.
                -->
                <service uid="myMenu" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Open file" shortcut="Ctrl+O" />
                            <separator />
                            <menuItem name="Quit" specialAction="QUIT" shortcut="Ctrl+Q" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionOpenFile" start="yes" />
                        <menuItem sid="actionQuit" start="yes" />
                    </registry>
                </service>

                <!--
                    Quit action:
                    The action service (::gui::action::SQuit) is a generic action that will close the
                    when the user click on the menuItem "Quit".
                -->
                <service uid="actionQuit" impl="::gui::action::SQuit" />

                <!--
                    Open file action:
                    This service (::gui::action::StarterActionService) is a generic action, it starts
```

```
                    services given in the configuration when the user clicks on the action.
                    Here, the reader selector will be called when the actions is clicked.
                -->
                <service uid="actionOpenFile" impl="::gui::action::SStarter">
                    <start uid="myReaderPathFile" />
                </service>

                <!--
                    Reader selector dialog:
                    This is a generic service that show a dialog to display all the reader or writer
                    associated data. By default it is configured to show reader. (Note: if there is o
                    service, it is directly selected without dialog box.)
                    Here, it the only reader available to read a ::fwData::Image is ::ioVTK::ImageRea
                    Tuto02DataServiceBasic), so the selector will not be displayed.
                    When the service was chosen, it is started, updated and stopped, so the data is i
                -->
                <service uid="myReaderPathFile" impl="::uiIO::editor::SIOSelector" />

                <!--
                    3D visualization service of medical images:
                    Here, the service attribute 'autoConnect="yes"' allows the rendering to listen th
                    the data image. So, when the image is loaded, the visualization will be updated.
                -->
                <service uid="myRendering" impl="::vtkSimpleNegato::SRenderer" autoConnect="yes" />

                <!--
                    Here, we only start the frame because all the others services are managed by the
                    - the frame starts the menu bar and the redering service
                    - the menu bar starts the menu services
                    - the menus starts the actions
                -->
                <start uid="myFrame" />

            </object>

        </config>
    </extension>
</plugin>
```

The framework provides some gui services:

Frame (`::gui::frame::DefaultFrame`) This service display a frame and creates menu bar, tool bar and container for views, rendering service, ...

View (`::gui::view::DefaultView`) This service creates sub-container and tool bar.

Menu bar (`::gui::aspect::DefaultMenuSrv`) A menu bar displays menus.

Tool bar (`::gui::aspect::DefaultToolBarSrv`) A tool bar displays actions, menus and editors.

Menu (`::gui::aspect::DefaultMenuSrv`) A menu displays actions and sub-menus.

Action (inherited from `::fwGui::IActionSrv`) An action is a service inherited from `::fwGui::IActionSrv`. It is called when the user clicks on the associated tool bar or menu.

Editors (inherited from `::gui::editor::IEditor`) An editor is a service inherited from `::gui::editor::IEditor`. It is used to creates your own gui container.

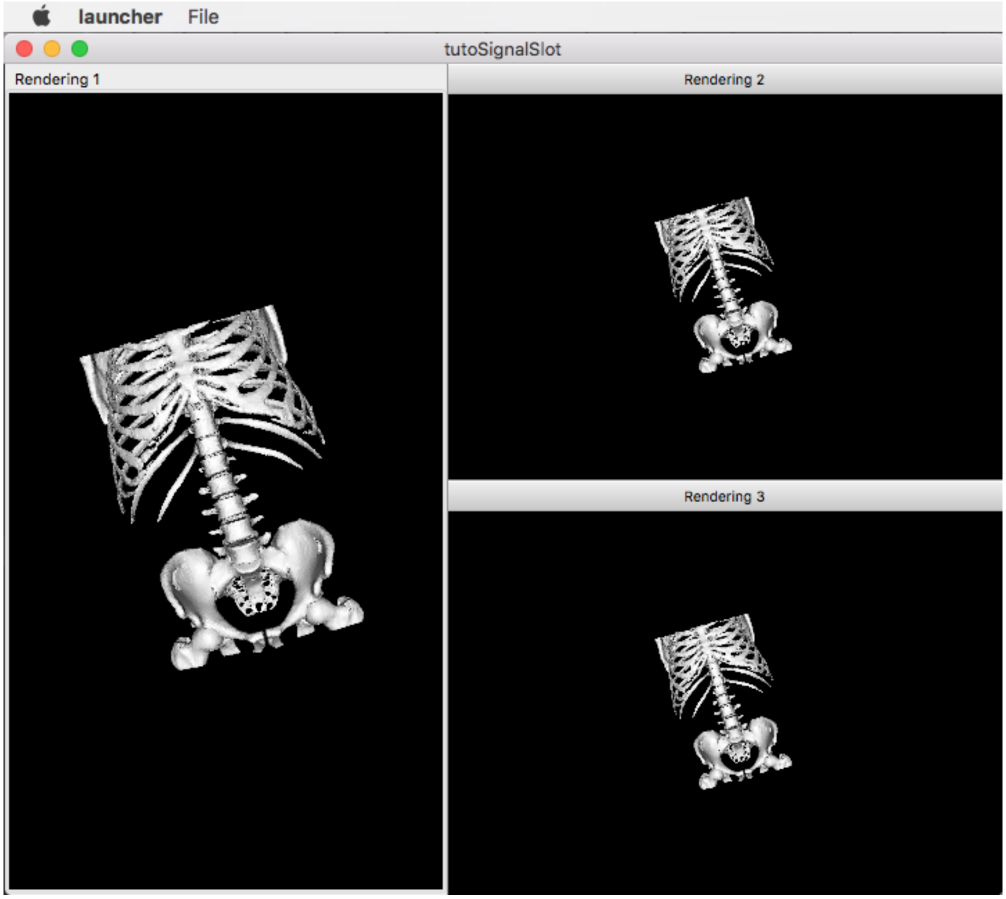
8.3.3 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto03DataService_0-1/profile.xml
```

8.4 [*Tuto04SignalSlot*] Signal-slot communication

The fourth tutorial explains the communication mechanism with signals and slots.



8.4.1 Prerequisites

Before to read this tutorial, you should have seen :

- *[Tuto03DataService] Display an image with menu*
- *Signal-slot communication*

8.4.2 Structure

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto04SignalSlot )
set( VERSION 0.1 )
set( TYPE APP )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io
    ioVTK
    uiIO
    visuVTK
    visuVTKQt
    vtkSimpleMesh # contains a visualization service of mesh.
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES tutoSignalSlotConfig)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<plugin id="Tuto04SignalSlot" version="@DASH_VERSION@">

    <requirement id="servicesReg" />

    <extension implements="::fwServices::registry::AppConfig">
        <id>tutoSignalSlotConfig</id>
        <config>

            <!-- The main data object is ::fwData::Mesh. -->
            <object type="::fwData::Mesh">

                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>tutoSignalSlot</name>
                            <icon>Bundles/Tuto04SignalSlot_0-1/tuto.ico</icon>
                            <minSize width="720" height="600" />
                        </frame>
                        <menuBar />
                    </gui>
                    <registry>
```

```
                        <menuBar sid="myMenuBar" start="yes" />
                        <view sid="myDefaultView" start="yes" />
                    </registry>
                </service>

                <service uid="myMenuBar" impl="::gui::aspect::SDefaultMenuBar">
                    <gui>
                        <layout>
                            <menu name="File" />
                        </layout>
                    </gui>
                    <registry>
                        <menu sid="myMenuFile" start="yes" />
                    </registry>
                </service>

                <!--
                    Default view service:
                    This service defines the view layout. The type '::fwGui::CardinalLayoutManager' i
                    central view and other views at the 'right', 'left', 'bottom' or 'top'.
                    Here the application contains a central view at the right.

                    Each <view> declared into the <layout> tag, must have its associated <view> into
                    A minimum window height and a width are given to the two non-central views.
                -->
                <service uid="myDefaultView" impl="::gui::view::SDefaultView">
                    <gui>
                        <layout type="::fwGui::CardinalLayoutManager">
                            <view caption="Rendering 1" align="center" />
                            <view caption="Rendering 2" align="right" minWidth="400" minHeight="100"
                            <view caption="Rendering 3" align="right" minWidth="400" minHeight="100"
                        </layout>
                    </gui>
                    <registry>
                        <view sid="myRendering1" start="yes" />
                        <view sid="myRendering2" start="yes" />
                        <view sid="myRendering3" start="yes" />
                    </registry>
                </service>

                <service uid="myMenuFile" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Open file" shortcut="Ctrl+O" />
                            <separator />
                            <menuItem name="Quit" specialAction="QUIT" shortcut="Ctrl+Q" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionOpenFile" start="yes" />
                        <menuItem sid="actionQuit" start="yes" />
                    </registry>
                </service>

                <service uid="actionOpenFile" impl="::gui::action::SStarter">
                    <start uid="myReaderPathFile" />
                </service>
                <service uid="actionQuit" impl="::gui::action::SQuit" type="::fwGui::IActionSrv" />

                <service uid="myReaderPathFile" impl="::uiIO::editor::SIOSelector">
                    <type mode="reader" /><!-- mode is optional (by default it is "reader") -->
                </service>

                <!--
                    Visualization services:
                    We have three rendering service representing a 3D scene displaying the loaded mes
                    shown in the windows defines in 'view' service.
                -->
                <service uid="myRendering1" impl="::vtkSimpleMesh::SRenderer" autoConnect="yes" />
                <service uid="myRendering2" impl="::vtkSimpleMesh::SRenderer" autoConnect="yes" />
                <service uid="myRendering3" impl="::vtkSimpleMesh::SRenderer" autoConnect="yes" />

                <!--
                    Each 3D scene owns a 3D camera that can be moved by the user on clicking in the s
                    - When the camera moved, a signal 'camUpdated' is emitted with the new camera in
                    focal, view up).
                    - To update the camera without clicking, you could called the slot 'updateCamPos

                    Here, we connect each rendering service signal 'camUpdated' to the others service
                    'updateCamPosition', so the cameras are synchronized in each scene.
                -->
                <connect>
                    <signal>myRendering1/camUpdated</signal>
                    <slot>myRendering2/updateCamPosition</slot>
                    <slot>myRendering3/updateCamPosition</slot>
                </connect>

                <connect>
                    <signal>myRendering2/camUpdated</signal>
                    <slot>myRendering1/updateCamPosition</slot>
                    <slot>myRendering3/updateCamPosition</slot>
                </connect>

                <connect>
                    <signal>myRendering3/camUpdated</signal>
                    <slot>myRendering2/updateCamPosition</slot>
                    <slot>myRendering1/updateCamPosition</slot>
                </connect>

                <start uid="myFrame" />
            </object>

        </config>
    </extension>

</plugin>
```

You can use **proxy** instead of the <connect> tag: it allows to connect all the signals to all the slots for a given channel name.

```
<proxy channel="Camera" >
    <signal>myRenderingTuto1/camUpdated</signal>
    <signal>myRenderingTuto2/camUpdated</signal>
    <signal>myRenderingTuto3/camUpdated</signal>
    <slot>myRenderingTuto1/updateCamPosition</slot>
    <slot>myRenderingTuto2/updateCamPosition</slot>
    <slot>myRenderingTuto3/updateCamPosition</slot>
</proxy>
```

Tip: You can remove a connection to see the camera in the scene is no longer synchronized.

8.4.3 Signal and slot creation

RendererService.hpp

```
class VTKSIMPLEMESH_CLASS_API RendererService : public fwRender::IRender
{
public:
    // .....

    typedef ::boost::shared_array< double > SharedArray;

    typedef ::fwCom::Slot<void (SharedArray, SharedArray, SharedArray)> UpdateCamPositionSlotType;

    typedef ::fwCom::Signal< void (SharedArray, SharedArray, SharedArray) > CamUpdatedSignalType;

    // .....

    /// This method is call when the VTK camera position is modified.
    /// It notifies the new camera position.
    void notifyCamPositionUpdated();

private:

    /// Slot: receives new camera information (position, focal, viewUp).
    /// Update camera with new information.
    void updateCamPosition(SharedArray positionValue,
                           SharedArray focalValue,
                           SharedArray viewUpValue);

    // ....

    /// Slot to call updateCamPosition method
    UpdateCamPositionSlotType::sptr m_slotUpdateCamPosition;

    /// Signal emitted when camera position is updated.
    CamUpdatedSignalType::sptr m_sigCamUpdated;
}
```

RendererService.cpp

```
RendererService::RendererService() throw()
{
    m_sigCamUpdated = newSignal<CamUpdatedSignalType>("camUpdated");

    m_slotUpdateCamPosition = newSlot("updateCamPosition",
                                      &RendererService::updateCamPosition,
                                      this);
}

//------------------------------------------------------------------------------

void RendererService::updateCamPosition(SharedArray positionValue,
                                        SharedArray focalValue,
                                        SharedArray viewUpValue)
{
    vtkCamera* camera = m_render->GetActiveCamera();

    // Update the vtk camera
    camera->SetPosition(positionValue.get());
    camera->SetFocalPoint(focalValue.get());
    camera->SetViewUp(viewUpValue.get());
    camera->SetClippingRange(0.1, 1000000);

    // Render the scene
    m_interactorManager->getInteractor()->Render();
}


//------------------------------------------------------------------------------

void RendererService::notifyCamPositionUpdated()
{
    vtkCamera* camera = m_render->GetActiveCamera();

    SharedArray position = SharedArray(new double[3]);
    SharedArray focal    = SharedArray(new double[3]);
    SharedArray viewUp   = SharedArray(new double[3]);

    std::copy(camera->GetPosition(), camera->GetPosition()+3, position.get());
    std::copy(camera->GetFocalPoint(), camera->GetFocalPoint()+3, focal.get());
    std::copy(camera->GetViewUp(), camera->GetViewUp()+3, viewUp.get());

    {
        // The Blocker blocks the connection between the "camUpdated" signal and the
        // "updateCamPosition" slot for this instance of service.
        // The block is release at the end of the scope.
        ::fwCom::Connection::Blocker block(
                            m_sigCamUpdated->getConnection(m_slotUpdateCamPosition));

        // Asynchronous emit of "camUpdated" signal
        m_sigCamUpdated->asyncEmit (position, focal, viewUp);
    }
}

//------------------------------------------------------------------------------

// ......
```

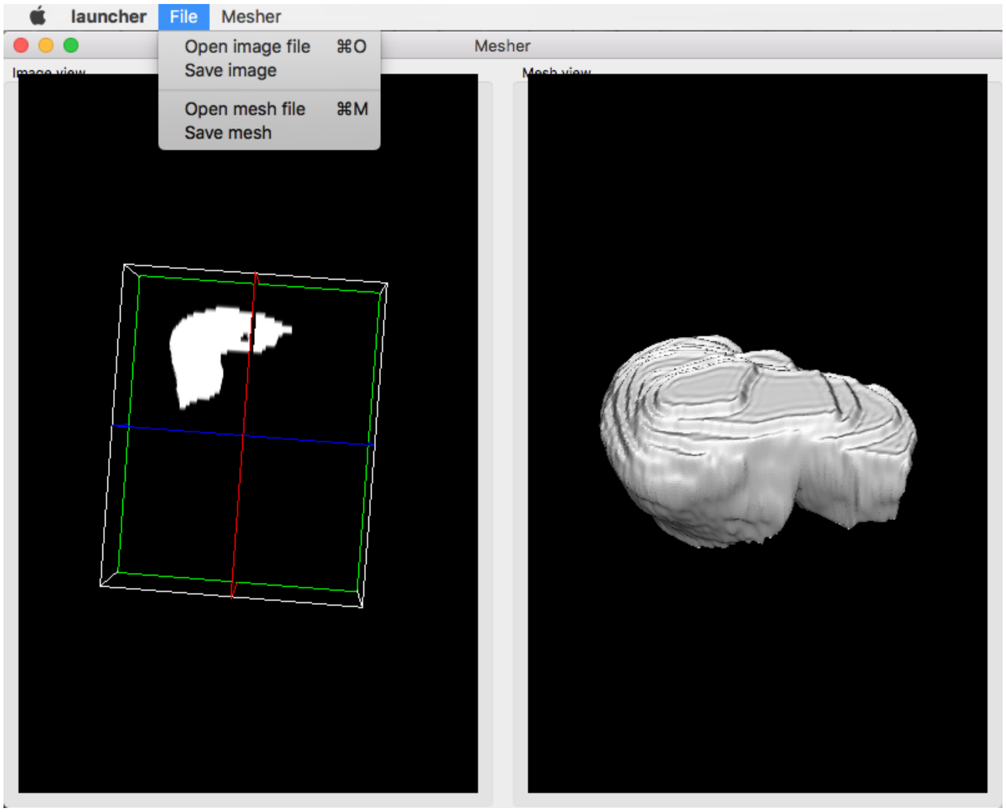
8.4.4 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto04SignalSlot_0-1/profile.xml
```

8.5 [*Tuto05Mesher*] Create a mesh from an image

The fifth tutorial explains how to use several object in an application. This application provides an action to creates a mesh from an image.



8.5.1 Prerequisites

Before to read this tutorial, you should have seen :

- *[Tuto04SignalSlot] Signal-slot communication*

8.5.2 Structure

Composite

A `::fwData::Composite` is an object that contains a map of `fwData::Object` associated to a key (`std::string`).

Using Composite in C++

```cpp
// Create a Composite
::fwData::Composite::sptr composite = ::fwData::Composite::New();

::fwData::Image::sptr image = ::fwData::Image::New();
::fwData::Mesh::sptr mesh = ::fwData::Mesh::New();

// Add an image and a mesh
composite->getContainer()["myImage"] = image;
composite->getContainer()["myMesh"] = mesh;
```

```cpp
::fwData::Composite::sptr composite = ::fwData::Composite::New();

// Get the image
::fwData::Image::sptr image = composite->at< ::fwData::Image >("myImage");

// Get the mesh
::fwData::Mesh::sptr mesh = composite->at< ::fwData::Mesh >("myMesh");
```

```cpp
::fwData::Composite::sptr composite = ::fwData::Composite::New();

// Check if the image exists into the composite
::fwData::Composite::iterator iter = composite->find("myImage");
if (iter != composite->end())
{
    // Image is found
    ::fwData::Image::sptr image = ::fwData::Image::dynamicCast(iter->second);
}
else
{
    // Image is not found
}
```

Using Composite in XML

```xml
<object type="::fwData::Composite">

    <!-- Composite services -->

    <item key="myImage">
        <object uid="myImageUID" type="::fwData::Image">

            <!-- Image services -->

        </object>
    </item>

    <item key="myMesh">
        <object uid="myMeshUID" type="::fwData::Mesh">

            <!-- Mesh services -->

        </object>
    </item>

```

```
</object>
```

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto05Mesher )
set( VERSION 0.1 )
set( TYPE APP )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io
    ioVTK
    visuVTK
    visuVTKQt
    uiIO
    vtkSimpleNegato
    vtkSimpleMesh
    opVTKMesh # provides services to generate a mesh from an image.
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES MesherConfig)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<plugin id="Tuto05Mesher" version="@DASH_VERSION@">

    <requirement id="servicesReg" />

    <extension implements="::fwServices::registry::AppConfig">
        <id>MesherConfig</id>
        <config>


            <!--
                The main data object is ::fwData::Composite.
                A Composite, can contains sub-objects associated to a key.
            -->
            <object type="::fwData::Composite">

                <!-- Frame & View -->

                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
```

```
                    <gui>
                        <frame>
                            <name>Mesher</name>
                            <icon>Bundles/Tuto05Mesher_0-1/tuto.ico</icon>
                            <minSize width="800" height="600" />
                        </frame>
                        <menuBar />
                    </gui>
                    <registry>
                        <menuBar sid="myMenuBar" start="yes" />
                        <view sid="myDefaultView" start="yes" />
                    </registry>
                </service>

                <!--
                    Default view service:
                    The type '::fwGui::LineLayoutManager' represents a layout where the view are alig
                    horizontally or vertically (set orientation value 'horizontal' or 'vertical').
                    It is possible to add a 'proportion' attribute for the <view> to defined the prop
                    used by the view compared to the others.
                -->
                <service uid="myDefaultView" impl="::gui::view::SDefaultView">
                    <gui>
                        <layout type="::fwGui::LineLayoutManager">
                            <orientation value="horizontal" />
                            <view caption="Image view" />
                            <view caption="Mesh view" />
                        </layout>
                    </gui>
                    <registry>
                        <view sid="RenderingImage" start="yes" />
                        <view sid="RenderingMesh" start="yes" />
                    </registry>
                </service>


                <!-- Menu Bar, Menus & Actions -->


                <service uid="myMenuBar" impl="::gui::aspect::SDefaultMenuBar">
                    <gui>
                        <layout>
                            <menu name="File" />
                            <menu name="Mesher" />
                        </layout>
                    </gui>
                    <registry>
                        <menu sid="menuFile" start="yes" />
                        <menu sid="menuMesher" start="yes" />
                    </registry>
                </service>


                <service uid="menuFile" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Open image file" shortcut="Ctrl+O" />
                            <menuItem name="Save image" />
                            <separator />
                            <menuItem name="Open mesh file" shortcut="Ctrl+M" />
                            <menuItem name="Save mesh" />
                            <separator />
                            <menuItem name="Quit" specialAction="QUIT" shortcut="Ctrl+Q" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionOpenImageFile" start="yes" />
                        <menuItem sid="actionSaveImageFile" start="yes" />
                        <menuItem sid="actionOpenMeshFile" start="yes" />
                        <menuItem sid="actionSaveMeshFile" start="yes" />
                        <menuItem sid="actionQuit" start="yes" />
                    </registry>
                </service>


                <service uid="menuMesher" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Compute Mesh (VTK)" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionCreateVTKMesh" start="yes" />
                    </registry>
                </service>

                <service uid="actionQuit" impl="::gui::action::SQuit" />

                <service uid="actionOpenImageFile" impl="::gui::action::SStarter">
                    <start uid="readerPathImageFile" />
                </service>

                <service uid="actionSaveImageFile" impl="::gui::action::SStarter">
                    <start uid="writerImageFile" />
                </service>

                <service uid="actionOpenMeshFile" impl="::gui::action::SStarter">
                    <start uid="readerPathMeshFile" />
                </service>

                <service uid="actionSaveMeshFile" impl="::gui::action::SStarter">
                    <start uid="writerMeshFile" />
                </service>

                <service uid="actionCreateVTKMesh" impl="::opVTKMesh::action::SMeshCreation">
                    <image uid="myImageUID" />
                    <mesh uid="myMeshUID" />
                    <percentReduction value="0" />
                </service>


                <!-- Image object associated to the key 'myImage' -->
                <item key="myImage">
                    <object uid="myImageUID" type="::fwData::Image">

                        <!--
```

```
                            Services associated to the Image data :
                            Visualization, reading and writing service creation.
                        -->
                        <service uid="RenderingImage" impl="::vtkSimpleNegato::SRenderer" autoConnect

                        <service uid="readerPathImageFile" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>

                        <service uid="writerImageFile" impl="::uiIO::editor::SIOSelector">
                            <type mode="writer" />
                        </service>

                    </object>
                </item>

                <!-- Mesh object associated to the key 'myMesh' -->
                <item key="myMesh">
                    <object uid="myMeshUID" type="::fwData::Mesh">

                        <!--
                            Services associated to the Mesh data :
                            Visualization, reading and writing service creation.
                        -->
                        <service uid="RenderingMesh" impl="::vtkSimpleMesh::SRenderer" autoConnect="y

                        <service uid="readerPathMeshFile" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>

                        <service uid="writerMeshFile" impl="::uiIO::editor::SIOSelector">
                            <type mode="writer" />
                        </service>

                    </object>
                </item>

                <start uid="myFrame" />

            </object>

        </config>
    </extension>
</plugin>
```

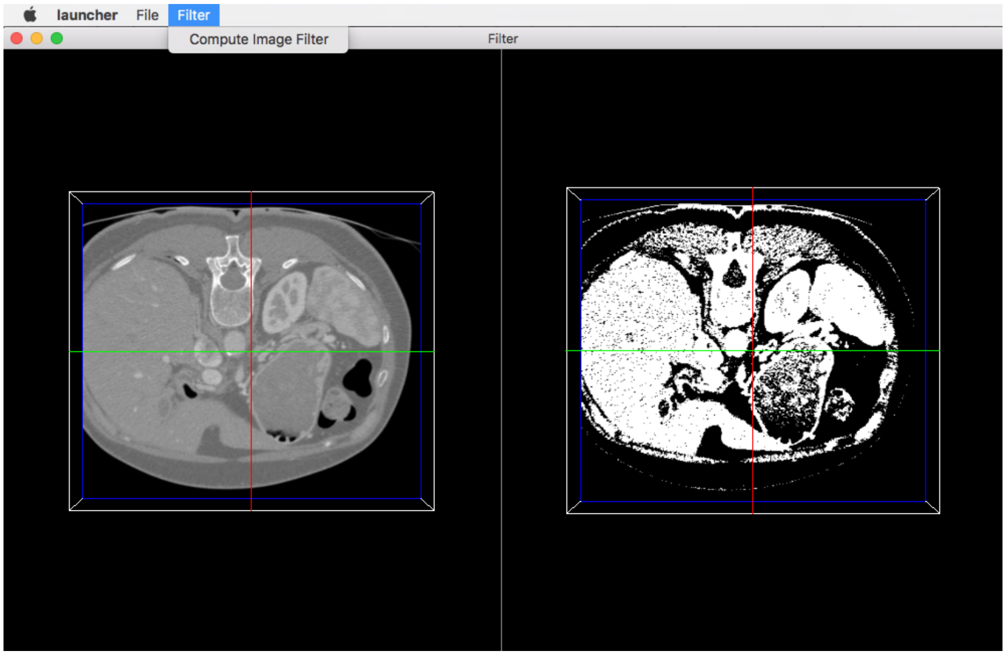
8.5.3 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto05Mesher_0-1/profile.xml
```

8.6 [*Tuto06Filter*] Apply a filter on an image

This tutorial explains how to perform a filter on an image. Here, the filter applied on the image is a threshold.



8.6.1 Prerequisites

Before to read this tutorial, you should have seen :

- *[Tuto05Mesher] Create a mesh from an image*

8.6.2 Structure

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto06Filter )
set( VERSION 0.1 )
set( TYPE APP )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io
    ioVTK
    uiIO
    visuVTK
    visuVTKQt
    vtkSimpleNegato
    opImageFilter # bundle containing the action to performs a threshold
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES FilterConfig)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<plugin id="Tuto06Filter" version="@DASH_VERSION@">

    <requirement id="dataReg" />
    <requirement id="servicesReg" />
    <requirement id="visuVTKQt" />

    <extension implements="::fwServices::registry::AppConfig">
        <id>FilterConfig</id>
        <config>

            <!-- Root object -->
            <object type="::fwData::Composite">

                <!-- Windows & Main Menu -->
                <service uid="myFrame" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>Filter</name>
                            <icon>Bundles/Tuto06Filter_0-1/tuto.ico</icon>
                            <minSize width="720" height="600" />
                        </frame>
                        <menuBar />
                    </gui>
                    <registry>
                        <menuBar sid="myMenuBar" start="yes" />
                        <view sid="myDefaultView" start="yes" />
                    </registry>
                </service>

                <service uid="myMenuBar" impl="::gui::aspect::SDefaultMenuBar">
                    <gui>
                        <layout>
                            <menu name="File" />
                            <menu name="Filter" />
                        </layout>
                    </gui>
                    <registry>
                        <menu sid="menuFile" start="yes" />
                        <menu sid="menuFilter" start="yes" />
                    </registry>
                </service>

                <service uid="myDefaultView" impl="::gui::view::SDefaultView">
                    <gui>
                        <layout type="::fwGui::CardinalLayoutManager">
                            <view align="center" />
                            <view align="right" minWidth="500" minHeight="100" />
                        </layout>
                    </gui>
                    <registry>
                        <view sid="RenderingImage1" start="yes" />
                        <view sid="RenderingImage2" start="yes" />
                    </registry>
                </service>

                <!-- Menus -->
                <service uid="menuFile" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Open image file" shortcut="Ctrl+O" />
                            <separator />
                            <menuItem name="Quit" specialAction="QUIT" shortcut="Ctrl+Q" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionOpenImageFile" start="yes" />
                        <menuItem sid="actionQuit" start="yes" />
                    </registry>
                </service>

                <service uid="menuFilter" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Compute Image Filter" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="actionImageFilter" start="yes" />
                    </registry>
                </service>

                <!-- Actions -->
                <service uid="actionQuit" impl="::gui::action::SQuit" />
                <service uid="actionOpenImageFile" impl="::gui::action::SStarter" >
                    <start uid="readerPathImageFile" />
                </service>

                <!--
                    Filter action:
                    This action applies a threshold filter. The source image is 'myImage1' and the
                    output image is 'myImage2'.
                    The two images are declared below.
                 -->
                <service uid="actionImageFilter" impl="::opImageFilter::action::SThreshold">
                    <imageIn uid="myImage1" />
                    <imageOut uid="myImage2" />
                </service>

                <!-- Image declaration: -->

                <!--
                    1st Image of the composite:
                    This is the source image for the filtering.
                -->
                <item key="myImage1">
                    <object uid="myImage1" type="::fwData::Image">
```

```
                        <service uid="RenderingImage1" impl="::vtkSimpleNegato::SRenderer" autoConnec
                        <service uid="readerPathImageFile" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>
                    </object>
                </item>

                <!--
                    2nd Image of the composite:
                    This is the output image for the filtering.
                -->
                <item key="myImage2">
                    <object uid="myImage2" type="::fwData::Image">
                        <service uid="RenderingImage2" impl="::vtkSimpleNegato::SRenderer" autoConnec
                    </object>
                </item>

                <start uid="myFrame" />

            </object>

        </config>
    </extension>
</plugin>
```

Filter service

Here, the filter service is inherited from ::fwGui::IActionSrv but you can inherit from another type (like `::arServices::IOperator` in fw4spl-ar repository).

For an action, the `updating()` method is called by the click on the button. This method retrieves the two images and applies the threshold algorithm.

The `::fwData::Image` contains a buffer for pixel values, it is stored as a `void *` to allows several types of pixel (uint8, int8, uint16, int16, double, float ...). To use image buffer, we need to cast it to the image pixel type. For that, we use the `Dispatcher` : it allows to invoke a template functor according to the image type.

```
void SThreshold::updating() throw ( ::fwTools::Failed )
{
    SLM_TRACE_FUNC();

    // threshold value: the pixel with the value less than 50 will be set to 0, else the value is se
    // value of the image pixel type.
    const double threshold = 50.0;

    ThresholdFilter::Parameter param; // filter parameters: threshold value, image source, image targ

    // Get source image
    OSLM_ASSERT("Image 1 not found. UID : " << m_imageSrcUID, ::fwTools::fwID::exist(m_imageSrcUID));
    param.imageIn = ::fwData::Image::dynamicCast( ::fwTools::fwID::getObject(m_imageSrcUID) );

    // Get target image
    OSLM_ASSERT("Image 2 not found. UID : " << m_imageTgtUID, ::fwTools::fwID::exist(m_imageTgtUID));
    param.imageOut = ::fwData::Image::dynamicCast( ::fwTools::fwID::getObject(m_imageTgtUID) );

    param.thresholdValue = threshold;
    /*
     * The dispatcher allows to apply the filter on any type of image.
     * It invokes the template functor ThresholdFilter using the image type.
     */
    ::fwTools::DynamicType type = param.imageIn->getPixelType(); // image type

    // Invoke filter functor
    ::fwTools::Dispatcher< ::fwTools::IntrinsicTypes, ThresholdFilter >::invoke( type, param );

    // Notify that the image target is modified
    auto sig = param.imageOut->signal< ::fwData::Object::ModifiedSignalType >(::fwData::Object::s_MOD
    {
        ::fwCom::Connection::Blocker block(sig->getConnection(m_slotUpdate));
        sig->asyncEmit();
    }
}
```

The functor is a *structure* containing a *sub-structure* for the parameters (inputs and outputs) and a template method `operator(parameters)`.

```
/**
 * Functor to apply a threshold filter.
 *
 * The pixel with the value less than the threshold value will be set to 0, else the value is set to
 * value of the image pixel type.
 *
 * The functor provides a template method operator(param) to apply the filter
 */
struct ThresholdFilter
{
    struct Parameter
    {
        double thresholdValue; ///< threshold value.
        ::fwData::Image::sptr imageIn; ///< image source
        ::fwData::Image::sptr imageOut; ///< image target: contains the result of the filter
    };

    /**
     * @brief Applies the filter
     * @tparam PIXELTYPE image pixel type (uint8, uint16, int8, int16, float, double, ....)
     */
    template<class PIXELTYPE>
    void operator()(Parameter &param)
    {
        const PIXELTYPE thresholdValue = static_cast<PIXELTYPE>(param.thresholdValue);
        ::fwData::Image::sptr imageIn  = param.imageIn;
        ::fwData::Image::sptr imageOut = param.imageOut;
        SLM_ASSERT("Sorry, image must be 3D", imageIn->getNumberOfDimensions() == 3 );
        imageOut->copyInformation(imageIn); // Copy image size, type... without copying the buffer
        imageOut->allocate(); // Allocate the image buffer

        ::fwComEd::helper::Image imageInHelper(imageIn); // helper used to access the image source bu
        ::fwComEd::helper::Image imageOutHelper(imageOut); // helper used to access the image target

        // Get image buffers
        PIXELTYPE *buffer1 = (PIXELTYPE *)imageInHelper.getBuffer();
        PIXELTYPE *buffer2 = (PIXELTYPE *)imageOutHelper.getBuffer();
        // Get number of pixels
        const size_t NbPixels = imageIn->getSize()[0] * imageIn->getSize()[1] * imageIn->getSize()[2

        // Fill the target buffer considering the thresholding
        for( size_t i = 0; i<NbPixels; ++i, ++buffer1, ++buffer2 )
        {
            *buffer2 = ( *buffer1 < thresholdValue ) ? 0 : std::numeric_limits<PIXELTYPE>::max();
        }
    }
};
```

8.6.3 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto06Filter_0-1/profile.xml
```

8.7 [*Tuto08GenericScene*] Generic scene

This tutorial explains how to use the generic scene.

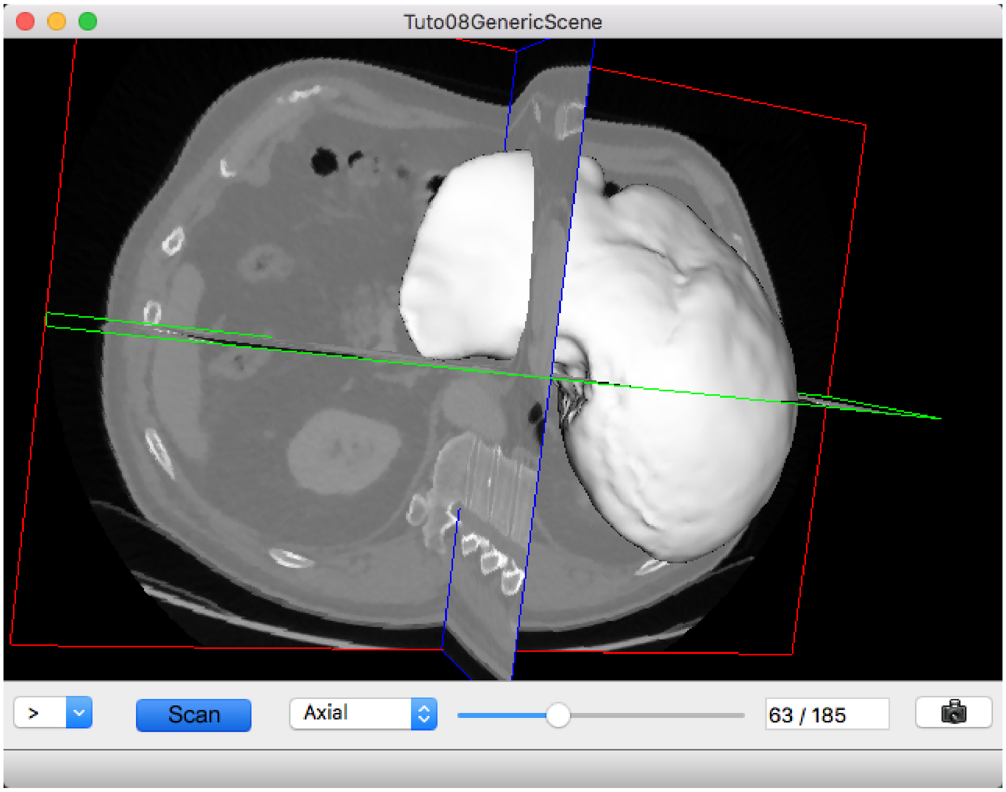


Fig. 8.1: Image and mesh

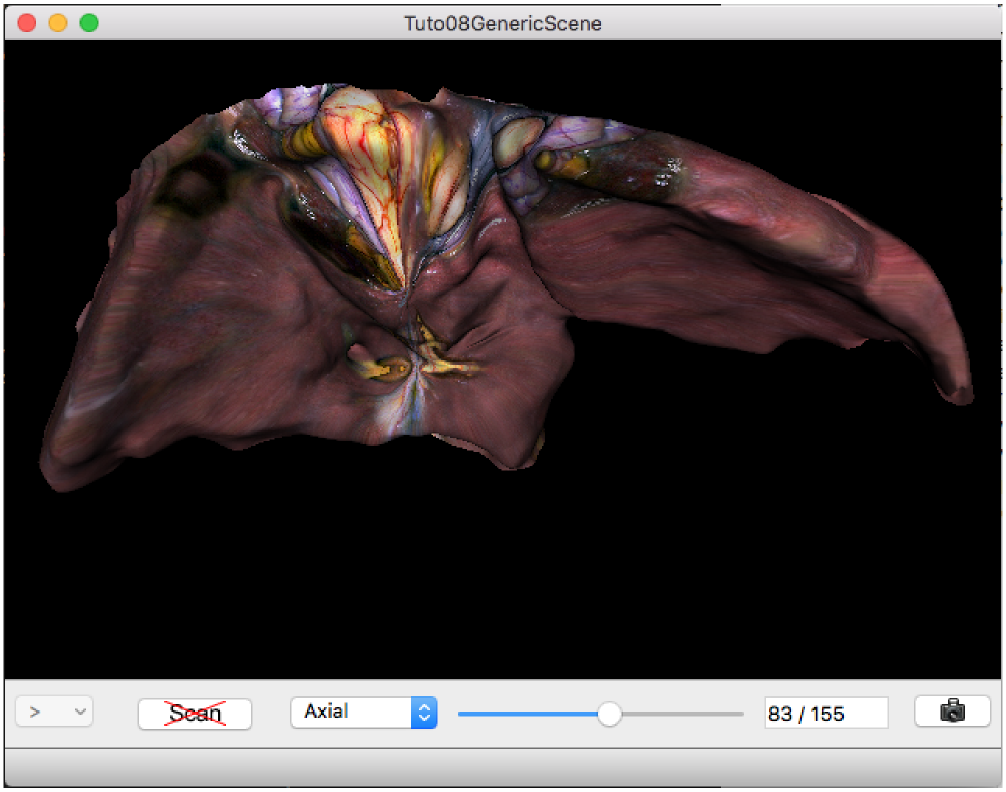


Fig. 8.2: Mesh with texture

8.7.1 Prerequisites

Before to read this tutorial, you should have seen :

- *Generic Scene*
- *[Tuto06Filter] Apply a filter on an image*

8.7.2 Structure

Properties.cmake

This file describes the project information and requirements :

```
set( NAME Tuto08GenericScene )
set( VERSION 0.1 )
set( TYPE APP )
set( UNIQUE TRUE )
set( DEPENDENCIES  )
set( REQUIREMENTS
    dataReg
    servicesReg
    gui
    guiQt
    io
    ioData # contains reader/writer for mesh (.trian) or matrix (.trf)
    ioVTK
    uiIO
    uiVisuQt # contains several editors for visualization
    uiImageQt # contains several editors on image
    visuVTK
    visuVTKQt
    visuVTKAdaptor # contains adaptors for the generic scene
    ctrlSelection # contains services to manage object selection (and associated services)
    launcher
    appXml
)

bundleParam(appXml PARAM_LIST config PARAM_VALUES Tuto08GenericScene)
```

Note: The Properties.cmake file of the application is used by CMake to compile the application but also to generate the `profile.xml`: the file used to launch the application.

plugin.xml

This file is in the `rc/` directory of the application. It defines the services to run.

```
<!--
    This tutorial shows a VTK scene containing a 3D image and a textured mesh.
    To use this application, you should open a 3D image, a mesh and/or a 2D texture image.
-->
<plugin id="Tuto08GenericScene" version="@DASH_VERSION@">

    <requirement id="dataReg" />
    <requirement id="servicesReg" />
    <requirement id="visuVTKQt" />


    <extension implements="::fwServices::registry::AppConfig">
        <id>Tuto08GenericScene</id>
        <config>

            <object type="::fwData::Composite">
                <service uid="MyIHM" impl="::gui::frame::SDefaultFrame">
                    <gui>
                        <frame>
                            <name>Tuto08GenericScene</name>
                            <icon>Bundles/Tuto08GenericScene_0-1/tuto.ico</icon>
                        </frame>
                        <menuBar />
                    </gui>
                    <registry>
                        <menuBar sid="myMenuBar" start="yes" />
                        <view sid="mainView" start="yes" />
                    </registry>
                </service>

                <!-- Status bar used to display the progress bar for reading -->
                <service uid="progress_statusbar" impl="::gui::editor::SJobBar" />

                <service uid="myMenuBar" impl="::gui::aspect::SDefaultMenuBar">
                    <gui>
```

```
                        <layout>
                            <menu name="File" />
                        </layout>
                    </gui>
                    <registry>
                        <menu sid="menu_File" start="yes" />
                    </registry>
                </service>

                <service uid="menu_File" impl="::gui::aspect::SDefaultMenu">
                    <gui>
                        <layout>
                            <menuItem name="Open image" shortcut="Ctrl+I" />
                            <menuItem name="Open mesh" shortcut="Ctrl+M" />
                            <menuItem name="Open texture" shortcut="Ctrl+T" />
                            <separator />
                            <menuItem name="Quit" specialAction="QUIT" shortcut="Ctrl+Q" />
                        </layout>
                    </gui>
                    <registry>
                        <menuItem sid="action_openImage" start="yes" />
                        <menuItem sid="action_openMesh" start="yes" />
                        <menuItem sid="action_openTexture" start="yes" />
                        <menuItem sid="action_quit" start="yes" />
                    </registry>
                </service>

                <!-- Actions to call readers -->
                <service uid="action_openImage" impl="::gui::action::SStarter">
                    <start uid="imageReader" />
                </service>
                <service uid="action_openMesh" impl="::gui::action::SStarter">
                    <start uid="meshReader" />
                </service>
                <service uid="action_openTexture" impl="::gui::action::SStarter">
                    <start uid="textureReader" />
                </service>

                <!-- Quit action -->
                <service uid="action_quit" impl="::gui::action::SQuit" />

                <!-- main view -->
                <service uid="mainView" impl="::gui::view::SDefaultView">
                    <gui>
                        <layout type="::fwGui::CardinalLayoutManager">
                            <view align="center" />
                            <view align="bottom" minWidth="400" minHeight="30" resizable="no" />
                        </layout>
                    </gui>
                    <registry>
                        <view sid="genericScene" start="yes" />
                        <view sid="editorsView" start="yes" />
                    </registry>
                </service>

                <!-- View for editors to update image visualization -->
                <service uid="editorsView" impl="::gui::view::SDefaultView">
                    <gui>
```

```
                        <layout type="::fwGui::LineLayoutManager">
                            <orientation value="horizontal" />
                            <view proportion="0" minWidth="30" />
                            <view proportion="0" minWidth="50" />
                            <view proportion="1" />
                            <view proportion="0" minWidth="30" />
                        </layout>
                    </gui>
                    <registry>
                        <view sid="sliceListEditor" start="yes" />
                        <view sid="showScanEditor" start="yes" />
                        <view sid="sliderIndexEditor" start="yes" />
                        <view sid="snapshotScene1Editor" start="yes" />
                    </registry>
                </service>

                <!--
                    Editor used for scene snapshot:
                    It allows to select the snapshot filename and emits a "snapped" signal with this
                -->
                <service uid="snapshotScene1Editor" impl="::uiVisu::SnapshotEditor" />

                <!--
                    Generic scene:
                    This scene display a 3D image and a textured mesh.
                -->
                <service uid="genericScene" impl="::fwRenderVTK::SRender" autoConnect="yes">
                    <scene>
                        <!-- Image picker -->
                        <picker id="myPicker" vtkclass="fwVtkCellPicker" />
                        <!-- Renderer -->
                        <renderer id="default" background="0.0" />

                        <!-- Mesh adapor -->
                        <adaptor id="meshAdaptor" class="::visuVTKAdaptor::Mesh" objectId="mesh">
                            <config renderer="default" picker="" uvgen="sphere" texture="textureAdapt
                        </adaptor>

                        <!-- Texture adaptor, used by mesh adaptor -->
                        <adaptor id="textureAdaptor" class="::visuVTKAdaptor::Texture" objectId="text
                            <config renderer="default" picker="" filtering="linear" wrapping="repeat"
                        </adaptor>

                        <!-- 3D image negatoscope adaptor -->
                        <adaptor id="imageAdaptor" uid="imageAdaptorUID" class="::visuVTKAdaptor::Neg
                            <config renderer="default" picker="myPicker" mode="3d" slices="3" sliceIn
                        </adaptor>

                        <!-- Snapshot adaptor: create a snapshot of the scene. It has a slot "snap"
                        <adaptor id="snapshotAdaptor" uid="snapshotUID" class="::visuVTKAdaptor::Snap
                            <config renderer="default" />
                        </adaptor>

                        <!--
                            Connection for snapshot:
                            connect the editor signal "snapped" to the adaptor slot "snap"
                        -->
                        <connect>
```

```
                            <signal>snapshotScene1Editor/snapped</signal>
                            <slot>snapshotUID/snap</slot>
                        </connect>

                        <!--
                            Connection for 3D image slice:
                            Connect the button (showScanEditor) signal "toggled" to the image adapto
                            slot "showSlice", this signals/slots contains a boolean.
                            The image slices will be show or hide when the button is checked/unchecke

                            The "waitForKey" attribut means that the signal and slot are connected o
                            "image" is present in the scene composite. It is recommanded to used beca
                            exists only if the object is present.

                        -->
                        <connect waitForKey="image">
                            <signal>showScanEditor/toggled</signal>
                            <slot>imageAdaptorUID/showSlice</slot>
                        </connect>

                        <!--
                            Connection for 3D image slice:
                            Connect the menu button (sliceListEditor) signal "selected" to the image
                            (MPRNegatoScene3D) slot "updateSliceMode", this signals/slots contains a
                            This integer defines the number of slice to show (0, 1 or 3).
                        -->
                        <connect waitForKey="image">
                            <signal>sliceListEditor/selected</signal>
                            <slot>imageAdaptorUID/updateSliceMode</slot>
                        </connect>

                    </scene>
                </service>


                <!-- ************************************************
                                    Displayed objects
                    ************************************************ -->

                <!-- Image displayed in the scene -->
                <item key="image">
                    <object uid="imageUID" type="::fwData::Image">

                        <service uid="imageReader" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>

                        <!--
                            Generic editor representing a menu button.
                            It send signal with the current selected item.
                        -->
                        <service uid="sliceListEditor" impl="::guiQt::editor::SSelectionMenuButton">
                            <toolTip>Manage slice visibility</toolTip><!-- button tooltip -->
                            <selected>3</selected><!-- Default selection -->
                            <items>
                                <item text="One slice" value="1" /><!-- first item, if selected the
                                <item text="three slices" value="3" /><!-- second item, if selected
                            </items>
                        </service>
                        <!--
                            Generic editor representing a simple button with an icon.
                            The button can be checkable. In this case it can have a second icon.
                            - It emits a signal "clicked" when it is clicked.
                            - It emits a signal "toggled" when it is checked/unchecked.

                            Here, this editor is used to show or hide the image. It is connected to
                        -->
                        <service uid="showScanEditor" impl="::guiQt::editor::SSignalButton">
                            <config>
                                <checkable>true</checkable>
                                <icon>Bundles/media_0-1/icons/sliceHide.png</icon>
                                <icon2>Bundles/media_0-1/icons/sliceShow.png</icon2>
                                <iconWidth>40</iconWidth>
                                <iconHeight>16</iconHeight>
                                <checked>true</checked>
                            </config>
                        </service>

                        <!-- Editor representing a slider to navigate into image slices -->
                        <service uid="sliderIndexEditor" impl="::uiImage::SliceIndexPositionEditor" a
                            <sliceIndex>axial</sliceIndex>
                        </service>

                    </object>
                </item>

                <!-- texture displayed on the mesh -->
                <item key="textureImage">
                    <object uid="textureUID" type="::fwData::Image">
                        <service uid="textureReader" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>
                    </object>
                </item>

                <!-- Mesh displayed in the scene -->
                <item key="mesh">
                    <object uid="meshUID" type="::fwData::Mesh">
                        <service uid="meshReader" impl="::uiIO::editor::SIOSelector">
                            <type mode="reader" />
                        </service>
                    </object>
                </item>


                <!-- Connects readers to status bar service -->
                <connect>
                    <signal>meshReader/jobCreated</signal>
                    <slot>progress_statusbar/showJob</slot>
                </connect>

                <connect>
                    <signal>imageReader/jobCreated</signal>
                    <slot>progress_statusbar/showJob</slot>
                </connect>
                <connect>
                    <signal>textureReader/jobCreated</signal>
                    <slot>progress_statusbar/showJob</slot>
                </connect>

                <!--
                    Connects showScanEditor signal "toggled" to sliceListEditor slot "setEnable", th
                    contains a boolean, so the sliceListEditor can be disabled when the image is not
                -->
                <connect>
                    <signal>showScanEditor/toggled</signal>
                    <slot>sliceListEditor/setEnabled</slot>
                </connect>

                <start uid="MyIHM" />
                <start uid="progress_statusbar" />
                <start uid="medicalImageConverter" />

            </object>

        </config>
    </extension>

</plugin>
```

GUI

This tutorials used multiple editors to manage the image rendering:

- show/hide image slices
- navigate between the image slices
- snapshot

The two editors (`SSelectionMenuButton` and `SSignalButton`) are generic, so we need to configure their behaviour in the xml file.

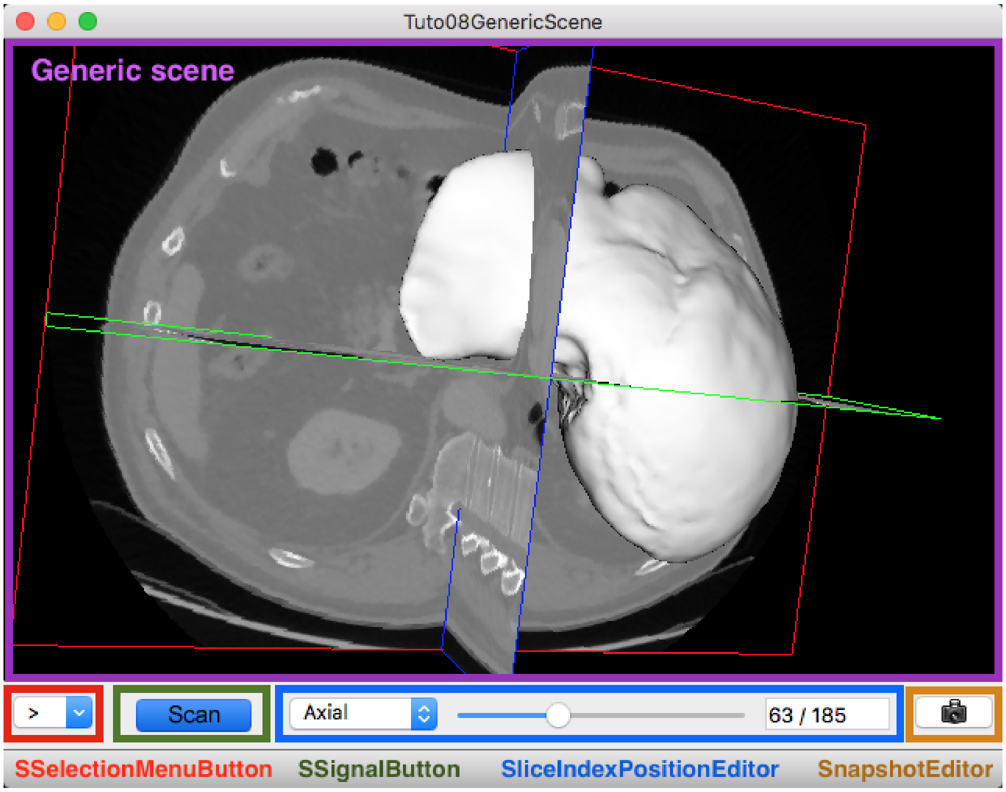
The editor aspect is defined in the service configuration. They emit signals that must be manually connected to the scene adaptor.

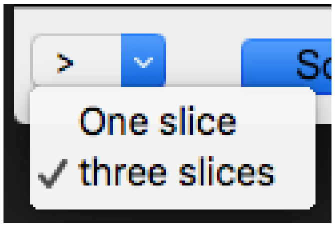
SSelectionMenuButton

This editor displays a menu when the user click on the button. Then the user can select one item.

```
<service uid="selectionMenuButton" impl="::uiImage::SSelectionMenuButton">
    <text>...</text>
    <toolTip>...</toolTip>
    <items>
        <item text="One" value="1" />
        <item text="Two" value="2" />
        <item text="Six" value="6" />
    </items>
    <selected>2</selected>
</service>
```

text (optional, default “>”) Text displayed on the button


Tuto08GenericScene
Generic scene
>
Scan
Axial
63 / 185
SSelectionMenuButton
SSignalButton
SliceIndexPositionEditor
SnapshotEditor


>
Sc
One slice
✓ three slices

toolTip (optional) Button tool tip

items List of the menu items

item One item

- **text** The text displayed in the menu
- **value** The value emitted when the item is selected

selected The value of the item selected by default

When the user select an item, a signal is emitted: the signal is `selected(int selection)`. It sends the value of the selected item.

In our case, we want to change the number of image slices displayed in the scene. So, we need to connect this signal to the image adaptor slot `updateSliceMode(int nbSlice)`.

```xml
<connect>
    <signal>selectionMenuButton/selected</signal>
    <slot>imageAdaptor/updateSliceMode</slot>
</connect>
```

SSignalButton

This editor shows a simple button.

```xml
<service uid="signalButton" impl="::guiQt::editor::SSignalButton" >
    <config>
        <checkable>true|false</checkable>
        <text>...</text>
        <icon>...</icon>
        <text2>...</text2>
        <icon2>...</icon2>
        <checked>true|false</checked>
        <iconWidth>...</iconWidth>
        <iconHeight>...</iconHeight>
    </config>
</service>
```

text (optional) Text displayed on the button

icon (optional) Icon displayed on the button

checkable (optional, default: false) If true, the button is checkable

text2 (optional) Text displayed if the button is checked

icon2 (optional) Icon displayed if the button is checked

checked (optional, default: false) If true, the button is checked at start

iconWidth (optional) Icon width

iconHeight (optional) Icon height

This editor provides two signals:

clicked() Emitted when the user click on the button.

toggled(bool checked) Emitted when the button is checked or unchecked.

In our case, we want to show (or hide) the image slices when the button is checked (or unckecked). So, we need to connect the `toogled` signal to the image adaptor slot `showSlice(bool show)`.

```
<connect>
    <signal>signalButton/toggled</signal>
    <slot>imageAdaptor/showSlice</slot>
</connect>
```

8.7.3 Run

To run the application, you must call the following line into the install or build directory:

```
bin/launcher Bundles/Tuto08GenericScene_0-1/profile.xml
```