MIPP: A Medical Image Processing Platform based on ITK, VTK and Eclipse RCP

Release 1.0

Roland Swoboda[1], Gerald Zwettler[1], Franz Pfeifer[1] and Werner Backfrieder[2]

December 31, 2007

[1]University of Applied Sciences Hagenberg, Bio- and Medical Informatics Research Group
Softwarepark 11, 4232 Hagenberg, Austria
[2]University of Applied Sciences Hagenberg, Software Engineering for Medicine Studies
Softwarepark 21, 4232 Hagenberg, Austria

Abstract

A novel platform for realizing software prototypes in the area of medical image processing, analysis and visualization is introduced: The "Medical Image Processing Platform" (MIPP). A novel feature is the combination of ITK/VTK with Java/Eclipse RCP, four potential technologies complementing each other. While ITK and VTK provide comprehensive functionality for image processing and visualization, Java and Eclipse RCP facilitate the development of a modern and rich application.
RCP offers everything for creating plug-in based software. The benefits of integrating ITK and VTK in such a modular architecture are reusability and extensibility. Functionality usually found in software for medical image processing and visualization (2D and 3D viewers, DICOM and application data management, etc.) is provided by the MIPP platform as plug-ins, can be easily reused by new applications and needn't be re-implemented. Developers can extend the platform's features by contributing new plug-ins and, again, make these available for others.
The paper describes the architecture of the MIPP platform and outlines its most important functionality. Finally, three clinical software systems implemented with MIPP are presented.

Contents

Project Homepage: http://mipp.fh-hagenberg.at

1 Introduction

The "Insight Segmentation and Registration Toolkit" (ITK)[1] and the "Visualization Toolkit" (VTK)[2] are among the leading open-source libraries in the domain of (medical) image processing and scientific visualization. Both libraries have been integrated in a lot of software systems (e.g. Analyze[3], MeVisLab[4], Slicer[5], VolView[6]). The considerable functionality that is available for reuse attracts many researchers and developers to implement their own algorithms and prototypes with ITK and VTK.

ITK offers data representations for images of arbitrary dimension and meshes plus standard linear and non-linear filters, image transformations and interpolators, intensity-, region- and model-based segmentation, rigid and non-rigid registration, finite elements, deformable models, level sets, mathematical morphology, etc. VTK supports a wide variety of visualization algorithms, including scalar, vector, tensor, texture, and volumetric methods as well as advanced modelling techniques like implicit modelling, polygon reduction, mesh smoothing, contouring and Delaunay triangulation. When it comes to graphical user interface (GUI) programming, both ITK and VTK rely on external libraries.

[1] http://www.itk.org
[2] http://www.vtk.org
[3] http://www.mayo.edu/bir/Software/Analyze/Analyze.html
[4] http://www.mevislab.de/
[5] http://www.slicer.org/
[6] http://www.kitware.com/products/volview.html

Because of the huge amount of data, performance and full control of memory are crucial for image processing and computer graphics. Consequently, C++ is often the preferred language. One drawback is that the C++ standard library offers very little functionality compared to modern platforms like Java and .NET. In general, application development is more convenient in Java than in C++ due to the large standard library that comes with it per default and the numerous third-party libraries/frameworks and tools that are available. Java is further a less complex programming language and frees the developer from manually managing memory, linking code and dealing with portability. One Java framework that has drawn a lot of attention during the last five years is Eclipse.

1.1 The Eclipse Project

Eclipse is an open-source software framework primarily written in Java. In its default form it is an integrated development environment (IDE) for Java developers (left part of figure 1). The fundamental part is the *Rich Client Platform* (RCP)[7]. RCP offers infrastructure for developing plug-in based software and manages plug-in dependencies, versions and patches. In fact, the IDE itself is a set of plug-ins.

RCP reuses two libraries for GUI programming, the *Standard Widget Toolkit* (SWT)[8] and *JFace*. SWT is a thin and portable layer on top of native widgets. JFace brings model-view-controller programming to SWT. The *generic workbench*, a set of plug-ins based on SWT and JFace, contains the main components for structuring the user interface: editors, views, perspectives, dialogs, wizards, forms, toolbars, actions and the workbench window.

Besides developing applications, RCP can be used to create other rich client platforms; and *MIPP RCP* is such a platform (see right part of figure 1). Its capabilities can be extended by writing new plug-ins. A MIPP RCP Application is integrated as, again, a set of plug-ins.

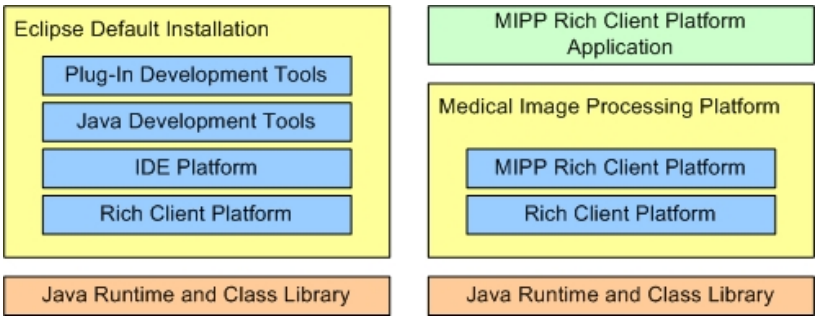

Figure 1: Eclipse Rich Client Platform: IDE vs. MIPP

Some other important features of Eclipse RCP not mentioned above are: preference pages, property editors, update sites (for deploying/updating plug-ins e.g. over the Web), native user interface, integration of OS-specific native code, portability, user-specific configurations, integrated help and internationalization.

1.2 Motivation

Developing medical software prototypes from scratch is a time-consuming task. Medical users expect functionality that is available in software systems they daily use in clinical practice. 2D and 3D viewers must

[7] http://www.eclipse.org/home/categories/rcp.php
[8] http://www.eclipse.org/swt/

visualize (multiple) image data sets and polygon models, allow standard and application specific user interaction and need to be synchronized among each other. DICOM and application specific data has to be stored and loaded, imported and exported. Besides all of that, the challenges of the actual scientific problem still have to be mastered.

A plug-in oriented architecture simplifies the implementation task. Functionality of existing plug-ins can be reused by applications, or better said, their plug-ins. The MIPP RCP plug-ins provide features that are usually needed for medical image processing and visualization: storing and browsing DICOM data, storing application data; 2D and 3D viewers with multi-object visualization (images, polygon models), camera and object transformation, inter-viewer synchronization; interactive widgets for clipping, orthogonal and oblique slice navigation, creating contour stacks, setting seed points; dialogs for parameterising and calling image filters, transfer function editors, wizards for importing DICOM data, etc. Another very important aspect is that – due to the extension facilities RCP provides – researchers can easily contribute to the platform and make their algorithms available for others.

The *aim of MIPP* is to provide a modular, reusable and extensible development platform for realizing research prototypes in the field of medical image processing, analysis and visualization. For the first time, ITK, VTK, Java and Eclipse RCP – four potential technologies – are combined into one platform.

The presented paper starts with a general overview of the Medical Image Processing Platform. Afterwards, the purpose, functionality and some technical details of each software layer are discussed. At the end, three real-world applications, developed in the course of three of our current research projects, are presented.

2 Platform Overview

The aim of MIPP is to provide a platform based on open-source software for developing prototypes/applications in the area of medical image processing, analysis, diagnosis and visualization. The platform is based on both *C++ and Java*. C++ offers performance and full control over memory, two important prerequisites for image processing and computer graphics algorithms. Application development is in general easier in Java than in C++.

Not all, but a lot of functionality of MIPP comes from the libraries and the frameworks that are reused. Usability, community acceptance, stability, portability and were important *criterions* for them to be selected for the platform. The platform – and consequently the libraries and frameworks too – must support Windows and also other operating systems that are often used in the scientific community (e.g. Linux and other Unix derivates). Besides that, a 64 bit version should be available. In contrast to 32 bit architectures, processes can then address more than 4GB[9]. The ”out of memory” scenario is often an issue when multiple and/or huge images have to be processed[10].

ITK, VTK and Eclipse RCP definitely meet the above mentioned criterions. All three run on multiple platforms, provide a 64 bit version and have proven their potential in numerous projects. The active community behind the three libraries/frameworks continuously improves existing features and introduces new functionality. Thereby, the Medical Image Processing Platform grows as well.

Figure 2 outlines the architecture of the platform and shows the most important libraries that were integrated.

A *MIPP application* – i.e. an application built on top of the MIPP platform – consists of two layers: a functional layer and a presentation layer. Each layer is, again, divided into two levels: a platform level and

[9] Of course, concepts like ”Physical Address Extension” (PAE) may increase that limit a bit

[10] Another solution could be to use ITK/VTK’s streaming support, but this feature has not yet been investigated

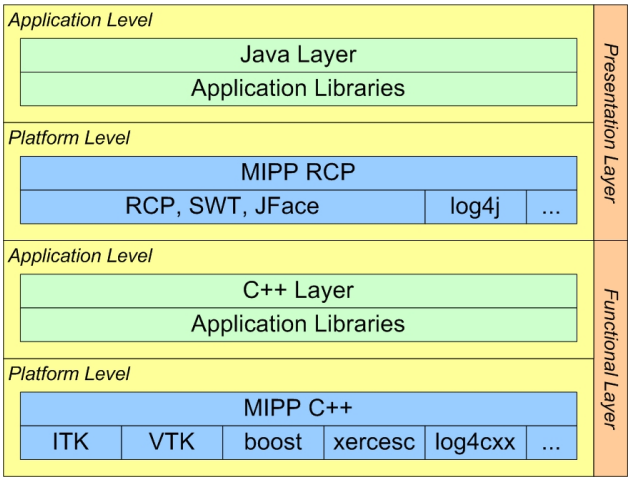

Figure 2: Overview of the Platform's Architecture

an application level. As the names imply, the former represents functionality provided by the platform and the latter represents functionality that has to be written or added by the application developer.

The *functional layer* is written in C++ and can be seen as a "kernel" where all the image processing, computer graphics and mathematical functionality resides. The kernel's functionality comes from the libraries integrated in the SDK (platform level) and is used by the developer for writing application-specific code (application level). Application-specific code can, of course, be based on other C++ libraries that are not part of the platform.

The *presentation layer* is written in Java and in the first instance responsible for providing the graphical user interface and for visualizing image data and polygon models. RCP, JFace and SWT offer a lot of functionality for creating a modern, modular user interface. In contrast to Swing, SWT uses native widgets and is therefore faster, more responsive to user interaction and lighter on system resource usage. Rendering windows are integrated in the GUI by using VTK's Java Wrapper. MIPP RCP is a set of plug-ins that provide widgets and functionality often required in medical applications. Application-specific code is implemented as RCP plug-ins and can, of course, reuse other Java libraries.

The *data layer* is written in C++ and responsible for storing DICOM image data and application data.

Note that there are also *alternative ways* for implementing a MIPP application. Instead of using Eclipse RCP, the GUI can be realized with Java AWT or Swing. For a pure C++ implementation, the platform integrates wxWidgets[11], a portable open-source GUI library. Nonetheless, we think that the most potential approach is combining ITK/VTK and RCP.

Figure 3 shows the binaries that are part of the platform. Their purpose and implementation is discussed in detail in the following sections.

[11] http://www.wxwidgets.org/

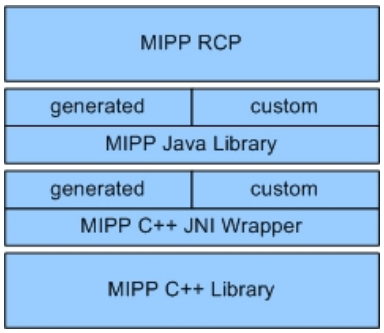

Figure 3: Platform-level "Binaries"

3 MIPP C++ Library

3.1 Data Concepts for Image Handling, Conversion and Filtering

The templated `itk::Image` class can not be used in a generic way because the image type (pixels/dimension) must be defined at compile time. Further conversion to `vtkImageData` is needed as visualization in the MIPP platform is performed based on VTK. Due to this reason we introduce a class for image handling in the MIPP, namely `mipp::image::ImageData`, see figure 4.

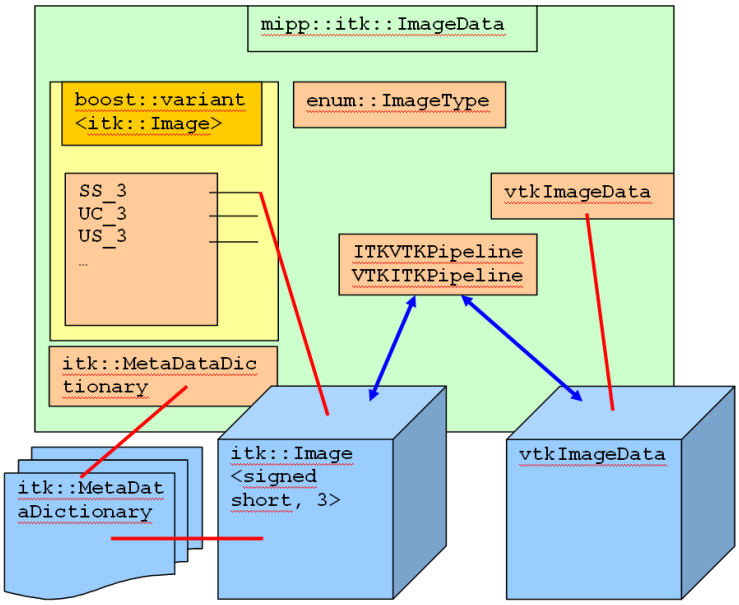

Figure 4: Structure of mipp::itk::ImageData. ITK and VTK image data conversion and templated data access is granted. Automated update is performed, when ITK or VTK data reference is changed.

To overcome the template instantiation problem, a certain predefined set of supported image types is established, e.g. including SS3, UC3, SC3, and UC2 to cite some examples. So `mipp::image::ImageData` is a container for the supported image types, storing the data via `boost::variant` generic container. Furthermore `mipp::image::ImageData` instances contain also a `vtkImageData` component and a conversion pipeline for dynamically casting ITK to VTK and vice versa. The ITK images are excluded from the automated wrapping process. Instead of that a conversion of `itk::Image` to `mipp::image::ImageBuffer`

is preserved, a 2D respectively 3D thin data container ready for easy wrapping and pixel/voxel access. `mipp::image::ImageBuffer` class for example makes thumbnail image data accessible for the Java GUI implementation.

Besides the image container functionality, `mipp::image::ImageData` grants access to the `itk::MetaDataDictionary`. The header file of `mipp::image::ImageData` is included in the appendix, documenting the above described design.

3.2 Imageprocessing Modules based on ITK utilizing MIPP ImageData

The MIPP platform is designed to provide many of the ITK filter implementations as a one call method. Therefore the ITK filters must be composed similar to the ITK example projects. The composition for `itk::BilateralImageFilter` is exemplarily presented in the appendix. The filter library is implemented as template class and the filters are implemented as static methods. The filter methods usually expect a pointer to `itk::image` and return a SmartPointer of `itk::image`.

The wrappable `mipp::imageprocessing::BasicImageFilterLibrary` only works on `mipp::image::ImageData` instances and delegates the filter call according to the current image type to the instantiated template method. In the course of this filter call switching, error handling can be introduced, e.g. when calling a filter method implemented for 3D data use only with 2D image data, exception handling becomes operative. Furthermore, intelligent input data casting can be introduced.

The described image filter process design is incompatible with the ITK/VTK pipeline concept. In contrast of this, static pipelines are implemented following the presented design and filter modules are aggregated for consecutive execution. Consequently classic filter pipelines with only one final `Update()` call can intentionally not be established with the MIPP platform.

As the methods are declared static, a concept for filter execution control must be designed and implemented. This task is performed in the course of `mipp::itk::ProgressHandler`, a parameter introduced in the following section. In addition to the default ITK filters, many proprietary implementations are added as they are needed for medical image processing purposes, e.g. fast 3D data skeletonization, fast 3D data morphology in general, Procrustes Modeling and Alignment as well as model based slice-wise LevelSet segmentation.

3.3 ProgressHandler for Cumulated Progress and Filter Execution Control

Out of the requirements discussed before, ITK filters, custom filters and image processing algorithms are composed to static methods. The composed static methods contain different filters and algorithms called directly, recursively or several times in control structures, each single call with a final progress of 100%. This requires a weighting of the relevant sub-filter executions to add the progress up to global 100% in total. To support custom algorithms and simple control structures, an ITK-independent concept (compared to `itk::ProgressAccumulator`) must be designed. For this purpose, class `mipp::itk::ProgressHandler` is introduced which is passed as parameter for each filter call when progress information should be preserved. Each call gets an input weight (summing up to 1.0 in total) and subdivides it according to the child filter call weights.

Besides progress information support, `mipp::itk::ProgressHandler` can be used to abort the static filter or algorithm execution. Via MACRO usage, check for abortion can be accomplished during filter execution. For simple usage of the progress concept, several MACROs are preserved:

- `MIPP_ITK_PROGRESSHANDLER_OUTER_CALL_CHECK(ph)`: check whether this filter call is the first one. Further check if `mipp::itk::ProgressHandler` is not NULL.
- `MIPP_ITK_PROGRESSHANDLER_REGISTER_PROCESS_OBJECT(ph, itkFilter, 0.6)`: current filter method relatively weights the current itkFilter call with e.g. 0.6 (relative to ph input weight).
- `MIPP_ITK_TRYCATCHSTART`: MAKRO for catching exceptions during filter execution, e.g. filter abortion or runtime exception.
- `MIPP_ITK_TRYCATCHEND`: catch possibly thrown exceptions
- `MIPP_ITK_PROGRESSHANDLER_CHECKFORABORTION(ph, defaultReturnVal)`: check if filter chain execution was aborted by the user; return the specified defaultReturnVal, e.g. NULL, when filtering was aborted.

After wrapping `mipp::itk::ProgressHandler`, filter progress and abortion mechanism is preserved for the possible application platforms, e.g. updating a progress bar in the GUI or implementing an abort-button.

3.4 Importing and Exporting ImageData

The intention of the MIPP platform is to support several image types and automatically choose the appropriate pixel format / dimensionality when importing the image data. This introduces flexibility compared to deciding the image type at compile time. Based on the generic `mipp::image::ImageData`, a generic concept for reading and writing image data is designed and implemented. Although currently only the file system is used as data source location and target, the chosen design with base class `mipp::image::DataSourceLocatorItem` easily allows extension to support a blob-database or a DICOM server, e.g. a Siemens Syngo host. For the currently used file system based importers and exporters, instances of `mipp::image::FileSystemDataSourceLocatorItem` are defined and parameterized with the single file path, or the directory to parse. Image importing is separated into different phases. The analysis is performed via a `mipp::io:FileFormatSupportRegistry` class implemented as singleton, where implementations for the supported file formats are registered, see figure 5 and 6.

First the `mipp::image::DataSourceLocatorItem` is preserved and the data referred to is ”analyzed” (for a single file the header is read, for a directory, the series of 2D files belonging to a 3D volume is assembled). The data source analysis process results in a `mipp::image::ImageInfo` instance returned as output. This class reports the image extent, dimensionality, spacing and preferred pixel type to use. Further possible meta data is preserved.

Based on this `mipp::image::ImageInfo` the data is imported in a second step. This concept provides great flexibility, for example when the user wants to select multiple DICOM series of a patient study for import. During data import, the pixel type and dimension of the `mipp::image::ImageData` is chosen according to the selected image info.

When importing 2D DICOM files belonging to a 3D volume, some parts of the meta data have to be corrected, i.e. exposure and KVP[12]. For series and study ID the policy is to keep the IDs unchanged.

When exporting image data into the `mipp::storage::DicomStorage`, DICOM is chosen as output file format. All 3D files assembled from 2D slices or Analyze7.5 input data are stored as single 3D DICOM files. Besides the image data, customizable thumbnail image data is generated. According to the DICOM

[12]DICOM tags describing the radiation dose affecting the patient during CT scan

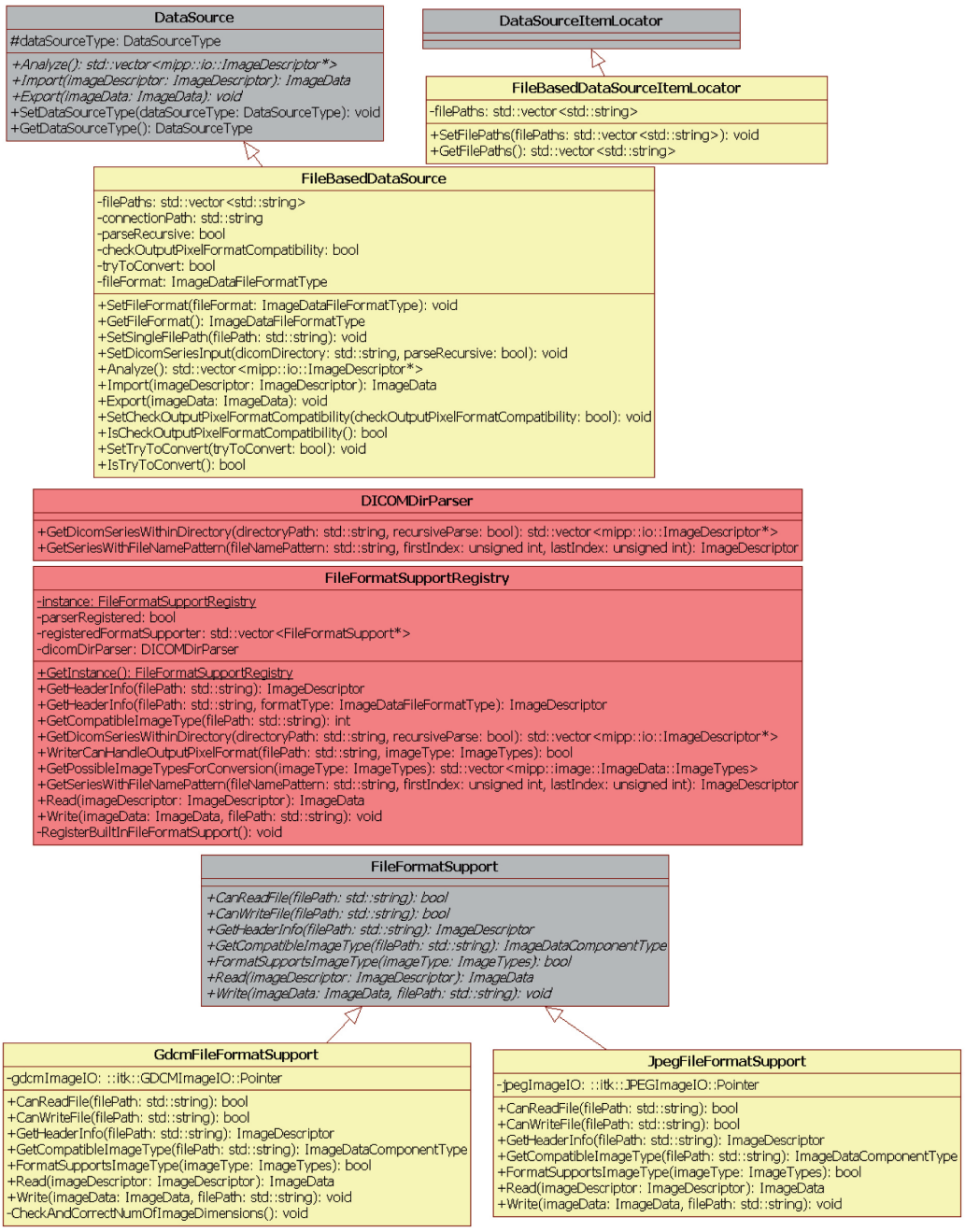

Figure 5: class diagram for file system Importer/Exporter classes and the FormatRegistry. The data source specifies the location where to import the image data from. File formats registered in the FileFormatSupportRegistry are applied during generic importing

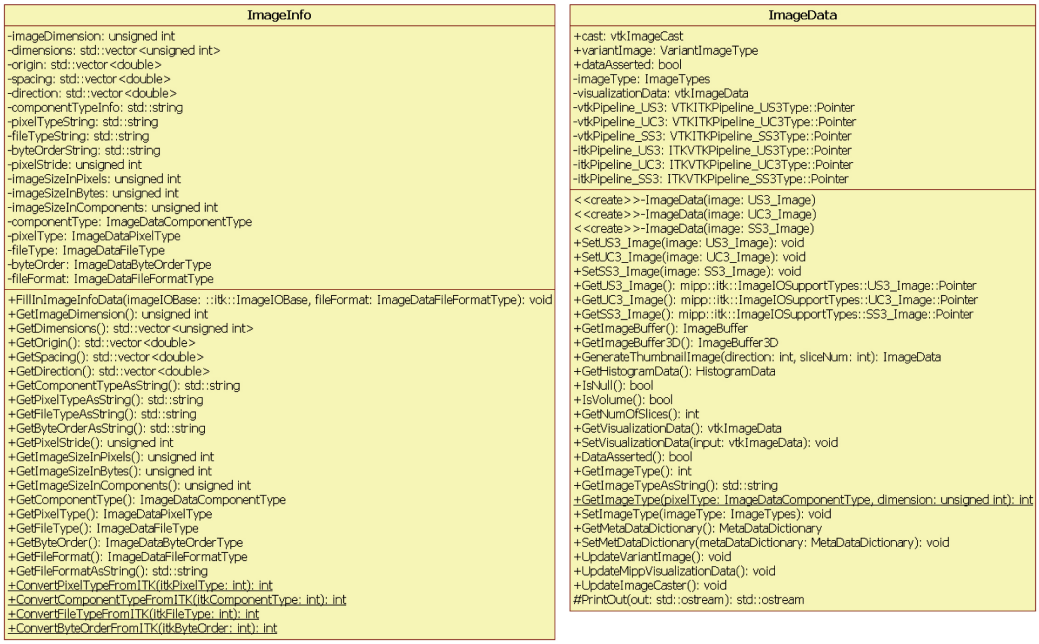

Figure 6: class diagram for ImageData, ImageInfo. ImageInfo is returned after analyzing the import directory. With one or several selected ImageInfo instances the ImageData can be read.

storage settings, all slices of the input volume, summed voxel projection, maximum intensity projection or minimum intensity projection are preserved at a certain maximum scale and a user defined slicing direction for thumbnail generation. Together with a description label the thumbnails are preserved e.g. in the RCP DICOM View and Import Dialog for detailed description before the user selects the data to open.

3.5 DICOM Storages

A DICOM storage is responsible for storing and querying DICOM image data. Although the functionality could be implemented in Java, all code is written in C++. Thus the storages are also reusable in other C++ programs.

Figure 7 gives an overview of the involved classes; their purpose is discussed in the following subsections.

Storage Keys

Similar to the DICOM model of the world, data is organized in levels: a study level and a series level. The patient level is omitted since the focus lies on managing images and not patients. However, patient information can be queried as DICOM tags. The purpose of the `mipp::storage::StudyStorageKey` and `mipp::storage::SeriesStorageKey` class is to identify data at the corresponding levels. Both classes have a member `ID` of type `string`. IDs are derived from the tags of the DICOM image that the key references. Which tags are used is specified in the `storages.xml` configuration file. The rationale behind this is that – although the DICOM standard provides tags for uniquely identifying studies and series – not every image acquisition system uses the same tags. Hard-coding the tags to be used for grouping would therefore be deficient.

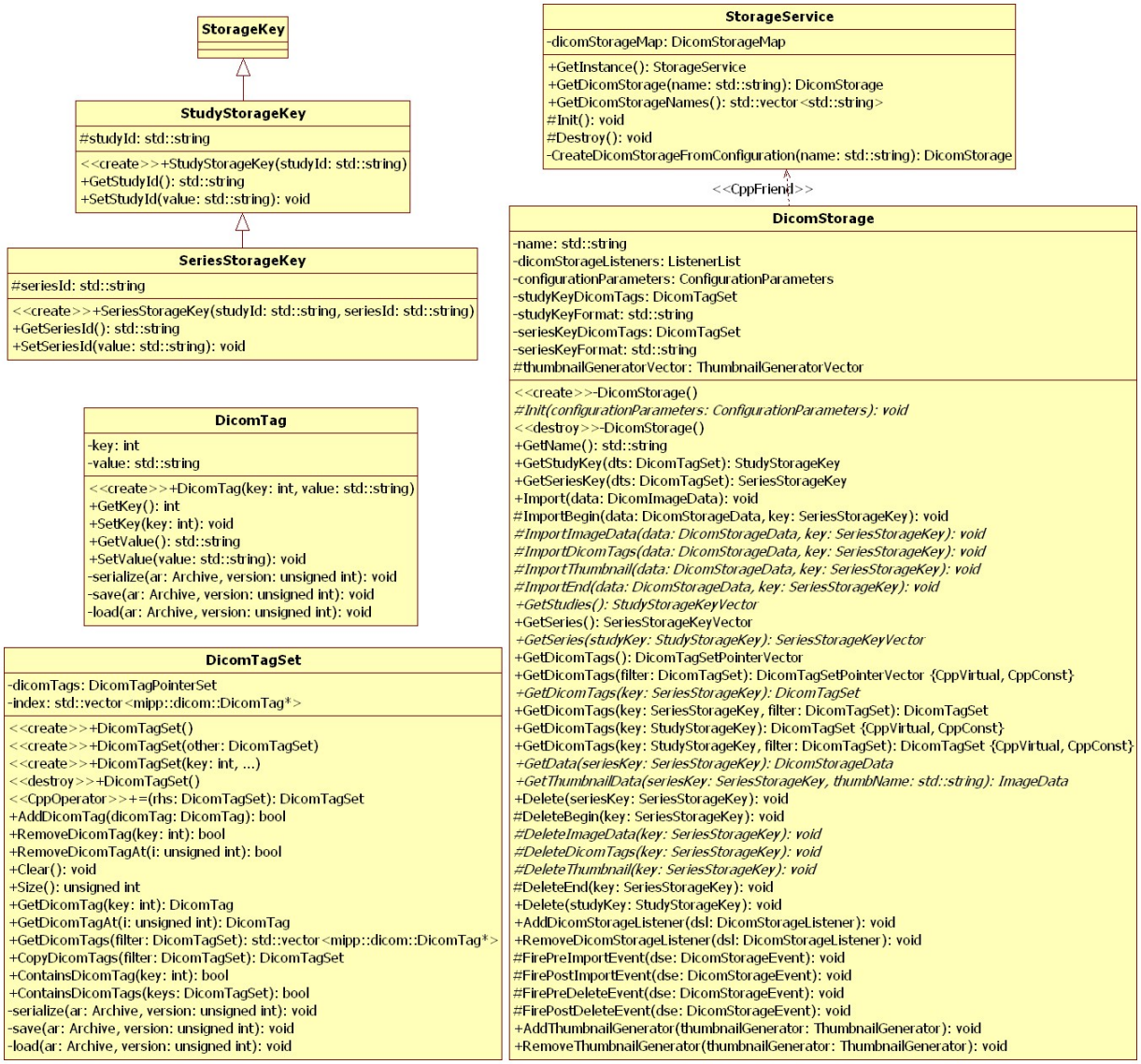

Figure 7: UML Diagram of DICOM Storage Implementation

Base Class DicomStorage

The design of the abstract class mipp::storage::DicomStorage specifies the following functionality:

- identifying a storage by a unique name, since there can be multiple instances
- creating a study and series key for a series based on its DICOM tags and the configuration in storages.xml
- importing a series
- querying storage keys to enumerate the studies and series that were imported
- querying the DICOM tags of a series, a study or the whole storage, optionally passing a filter (which works on DICOM tag keys)
- loading a persisted series referenced by its series key
- loading thumbnails created for a series during importing
- deleting a persisted series or study referenced by its storage key
- registering listeners that receive events before and after a series is imported and deleted

A DICOM series is represented by class DicomImageData which holds a pointer to an ImageData object of "type" DICOM and adds convenient methods for accessing DICOM tag information. A DicomTagSet is a collection that contains DicomTag objects. Thumbnails are instances of class ImageData, can be 2D or 3D and needn't be in DICOM format.

File-based Implementation

A concrete implementation of DicomStorage is mipp::storage::FileBasedDicomStorage, which stores all data in the file system. The advantages of a file based storage compared to a database are that it is more lightweight and needs no installation.

Each FileBasedDicomStorage has its own root directory. Settings like the root directory are specified in the storages.xml configuration file. For each study, a directory named after the study key ID is created in the root directory. Accordingly, for each series, a directory named after the series key ID is created in the study directory. The series directory contains the following files:

- imagedata.dcm: the ImageData instance contained in the DicomImageData object is written using the exporters and read using the importers described in one of the previous sections
- dicomtags.xml: DICOM tags are stored in a separate file to avoid the overhead of parsing imagedata.dcm. DICOM information contained in the itk::MetaDataDictionary of the ImageData instance is converted to a DicomTagSet object. Both the DicomTag and DicomTagSet class are made persistable by utilizing the boost::serialization library. The DicomTagSet object is written with boost::archive::xml_oarchive and is therefore human readable. Accordingly, boost::archive::xml_iarchive is used for reading.

- thumbnail files: these files are generated by the `ThumbnailGenerators` registered for a DICOM storage

The `boost::filesystem` library is used to access the file system in a platform independent way. Note that `boost::serialization` and other boost libraries are portable as well.

Storage Service and Configuration

DICOM storages are not instantiated directly. Instead, the `mipp::storage::StorageService` singleton class loads the `storages.xml` configuration file and (lazily) creates the storage instances. The following lines show an example `storages.xml` configuration file:

```
<StorageConfiguration>
   <DicomStorage name="Test" type="mipp::storage::FileBasedDicomStoragePersister">
      <ConfigurationParameter key="studyKeyDicomTag" value="0020|000D"/>
      <ConfigurationParameter key="studyKeyFormat" value="%1%"/>
      <ConfigurationParameter key="seriesKeyDicomTag" value="0020|000E"/>
      <ConfigurationParameter key="seriesKeyFormat" value="%1%"/>
      <ConfigurationParameter key="rootDir" value="C:\DicomStorages\Test"/>
   </DicomStorage>
      <DicomStorage name="AnotherTest" type="mipp::storage::FileBasedDicomStoragePersister">
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|0060"/>
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|1090"/>
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|0020"/>
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|0030"/>
      <ConfigurationParameter key="studyKeyFormat" value="%1%_%2%_%3%_%4%"/>
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|0021"/>
      <ConfigurationParameter key="studyKeyDicomTag" value="0008|0031"/>
      <ConfigurationParameter key="seriesKeyFormat" value="%1%_%2%"/>
      <ConfigurationParameter key="rootDir" value="C:\DicomStorages\AnotherTest"/>
   </DicomStorage>
</StorageConfiguration>
```

The code snippet above defines two file-based DICOM storages. The `StorageService` creates an instance of the specified `type` and passes all `ConfigurationParameters` as STL `multimap` to the storage's `init` method. DICOM storage "`Test`" uses the `StudyInstanceUID` and `SeriesInstanceUID` DICOM tag for the study and series key, respectively. DICOM storage "`AnotherTest`" uses `Modality`, `ManufacturersModelName`, `StudyDate`, `StudyTime`, and `SeriesDate`, `SeriesTime` for the study and series key, respectively. Unless a DICOM series contains all study key and series key DICOM tags, it cannot be imported. Note that the format of the study and series key can be specified. This feature is implemented utilizing the `boost::format` library.

4 Bringing together C++ and Java

In Java, C/C++ code is integrated via "Java Native Interface" (JNI). Usually, the programmer writes a Java class, declares the method to be implemented in C/C++ with the `native` keyword and runs the `jnih` tool[13] which generates C/C++ header files containing JNI method stubs. Then, the programmer has to

[13] The `jnih` tool is part of the Java Development Kit (JDK)

implement the stubs, link the code into a dynamic link library and finally load that library in Java using `System.loadLibrary("libName")`.

As the MIPP C++ library consists of many C++ classes, writing the JNI code would be a very time-consuming task. Fortunately, this task can be automated by utilizing the *"Simplified Wrapper and Interface Generator"* (SWIG) tool. For each C++ class needed on the Java side, a corresponding Java proxy class is generated by SWIG. The Java proxy class usually has the same methods (declared `native`) as the C++ class and contains a member that holds a pointer to the underlying C++ object. In addition to the proxy class, SWIG generates the JNI header file and implementation. The generated code delegates JNI method calls to the corresponding C++ methods and converts between JNI and C++ data types. Besides classes, SWIG can also wrap structures, unions, enums, templates and smart pointers. The way how SWIG maps a C++ data type to Java can be specified by a configuration file. For a full list of features and options, please refer to SWIG's documentation.

The "MIPP C++ JNI Wrapper" layer binary (see figure 3) contains all the C++ JNI wrapper code for the following implementations: ImageData, image processing libraries, importers and exporters, DICOM and application storages. The "MIPP Java Library" layer consists of two parts: a generated part, i.e. the Java proxy classes created by SWIG, and a custom part. Some methods of MIPP C++ classes require the caller to free memory. For instance, `mipp::storage::DicomStorage::getDicomTags()` allocates/returns an `mipp::dicom::DicomTagSet` object. Accordingly, the `mipp.storage.DicomStorage.getDicomTags()` proxy method on the Java side returns an instance of the `mipp.dicom.DicomTagSet` class, which holds a pointer to C++ memory that must be deallocated. For this purpose, SWIG adds a `delete()` method to proxy classes which calls the destructor of the underlying C++ object. To free Java programmers from such manual memory management issues, the custom wrappers were introduced. There, a subset of the MIPP C++ classes is re-implemented as "pure Java classes". They provide methods for converting to/from Java proxy classes and have the name prefix "`J`", e.g. `mipp.dicom.JDicomTagSet`. Another difference is that data resides on the Java side and not on the C++ side. This additionally eliminates the overhead of JNI calls when accessing data. However, the pure Java classes are only for convenience and implemented for a small subset of MIPP C++ classes, i.e.: StorageService, DicomStorage, the storage keys and DICOM tags data structures. The latter, for instance, is read-only information and used frequently in MIPP RCP. In such a case, it's more appropriate to convert the Java proxy class instance into a pure Java class instance once, since there is no need for manual deallocation and no overhead by JNI calls then. The `JStorageService` and `JDicomStorage` classes use the pure Java classes in their interfaces rather than the proxy classes.

Note that VTK C++ classes don't have to be wrapped with SWIG since VTK already provides Java wrapper. In fact, VTK's Java wrapping binaries – `vtkParseJava` and `vtkWrapJava` – are reused to wrap the implemented 2D and 3D viewers, as well as the implemented widgets for clipping, slice navigation, seed points and contour stack.

As described in section 2, the functional layer of a MIPP application is usually written in C++. Note that if C++ types need to be used in Java, SWIG can/should be utilized for automating the generation of the application's wrapper code.

5 MIPP Rich Client Platform

5.1 Integration of Native Code

For the integration of platform-specific content, RCP offers the concept of *"fragments"*. Fragments are like regular plug-ins, except that they may only have one dependency, namely back to its "host plug-in". Typi-

cally, the fragment contains the native dynamic link libraries and is named after the operating system, windowing system and architecture the libraries are compiled for (e.g. `org.kitware.vtk.win32.win32.x86`, `org.kitware.vtk.linux.gtk.x86_64`). To ensure that a fragment runs exclusively on the system it is targeted for, RCP allows specifying a "platform filter". In this case, RCP will only load the fragment if the underlying system corresponds to the operating system, windowing system and architecture specified by the filter.

Each C++ library that is part of MIPP is integrated using five plug-ins, as demonstrated exemplarily for VTK:

- `org.kitware.vtk`: host plug-in with Java wrapper classes and Activator
- `org.kitware.vtk.feature`: updateable plug-in with dependency to host plug-in and release fragment
- `org.kitware.vtk.feature.debug`: updateable plug-in dependency to host plug-in and debug fragment
- `org.kitware.vtk.win32.win32.x86`: fragment with Windows x86 release binaries, platform filter, host plug-in is `org.kitware.vtk`
- `org.kitware.vtk.win32.win32.x86.debug`: fragment with Windows x86 debug binaries, platform filter, host plug-in is `org.kitware.vtk`

The plug-ins have the same version number as the library (e.g: `5.1.0.20061107` for VTK nightly build). As mentioned before, the dynamic link libraries are part of a fragment; they are not installed on the system and not added to the system path. This, and the fact that version numbers are respected for plug-in dependencies, helps to avoid the "DLL hell" under Windows. Another benefit is that a fragment and its contained dynamic link libraries can be installed/updated with RCP's update management functionality.

Of course, Java wrapper classes can only be found in that host plug-ins whose C++ library is wrapped, i.e. VTK and the MIPP C++ library. The reason why plug-ins exist for unwrapped C++ libraries is that the dynamic link library of the MIPP C++ library has dependencies to them.

An RCP plug-in can have an "Activator" class, whose `start` and `stop` methods are called when the plug-in is activated and (de)activated, respectively. In our case, the Activator of the host plug-in issues the `System.loadLibrary()` calls to load the dynamic link libraries.

MIPP RCP offers both debug and release version of the fragments (dynamic link libraries). In the former case, the suffix `.debug` is appended to the fragment's name. When developing a MIPP RCP application with the Eclipse IDE, four combinations are possible for executing the application: running the Java part in debug/release mode and running the C++ part in debug/release mode (using the debug/release fragments). For *cross-language debugging* with the Eclipse IDE and Visual Studio, the following steps are necessary: 1) open the corresponding C++ project in Visual Studio and set breakpoints, 2) start the Java application with Eclipse, 3) attach the Visual Studio debugger to the started Java process. Whenever a breakpoint is hit on C++ side, the Visual Studio debugger gets activated and the Eclipse IDE is halt.

Besides fragments, RCP provides the concept of *features*. Features collect sets of plug-ins that go together to form some coherent unit of function. They define dependencies to other plug-ins, fragments and features and can be installed/updated with RCP's update management functionality. In our example, the feature plug-in `org.kitware.vtk.feature` has a dependency to host plug-in `org.kitware.vtk` and fragment plug-in `org.kitware.vtk.win32.win32.x86`. Note that features are also categorised in debug (`xxx.feature.debug`) and release (`xxx.feature`).

MIPP RCP is available in two versions: a debug version (`mipp.rcp.runtime.feature.debug`) and a release version (`mipp.rcp.runtime.feature`). Note that both are updatable – e.g. over the web – using RCP's update management functionality.

5.2 Perspectives, Views, and Editors in Eclipse RCP

Figure 8 illustrates the composition of perspectives, views and editors at run-time. For more details about these topics, the interested reader may be referred to more elaborate Eclipse RCP resources[14].

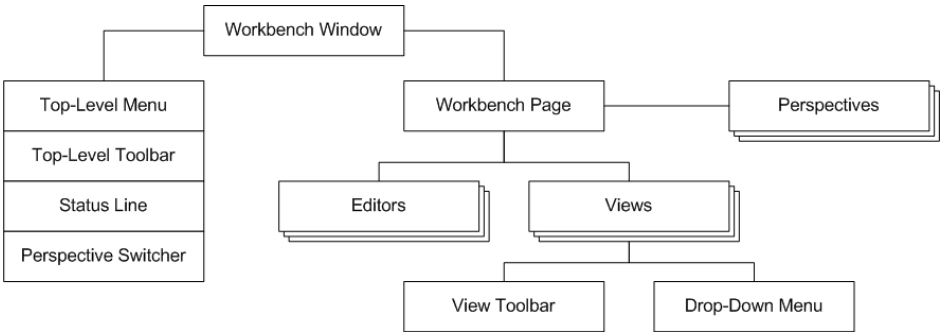

Figure 8: Runtime Composition of Perspectives, Views, Editors

The following citations are taken from [6]:

"... Perspectives group together and organize UI elements that relate to a specific task or workflow. ... Think of perspectives as a set of layout hints for a window. Every Workbench Window has one page. The page owns its editor and view instances and uses the active perspective to decide its layout. The perspective details where, and where not, to show certain things, such as views, the editor area and actions. ... A perspective factory provides the initial layout for a perspective. ... You can have multiple perspectives in the same window, allowing you to switch tasks without having to change windows. ..."

5.3 DICOM Storage Perspective

The "DICOM Storage" perspective and its views and top-level menu contributions allow the user to interact with DICOM storages, i.e.: import new series, browse and open imported series, show the DICOM tags of selected series, display the thumbnails of a series and delete imported series. The initial layout of the perspective is shown in figure 9.

DICOM Storage Browser View

The `mipp.rcp.dicomstorage.ui.DicomStorageBrowser` view is responsible for presenting the content of DICOM storages in tree form. The tree view is implemented using the JFace `TreeViewer` widget. Instead of accessing the storages with the generated MIPP Java wrappers and having to deal with memory deallocation, the custom wrappers are used (see section 4). The view retrieves a list of available DICOM storages from `JStorageService`, obtains a reference to each `JDicomStorage` and queries the DICOM tags of all of its series. All tags of a single series are contained in a `JDicomTagSet` object.

[14] Recommendations: the Eclipse FAQs (http://wiki.eclipse.org/The_Official_Eclipse_FAQs), the "Eclipse Series" books offered by Addison-Wesley

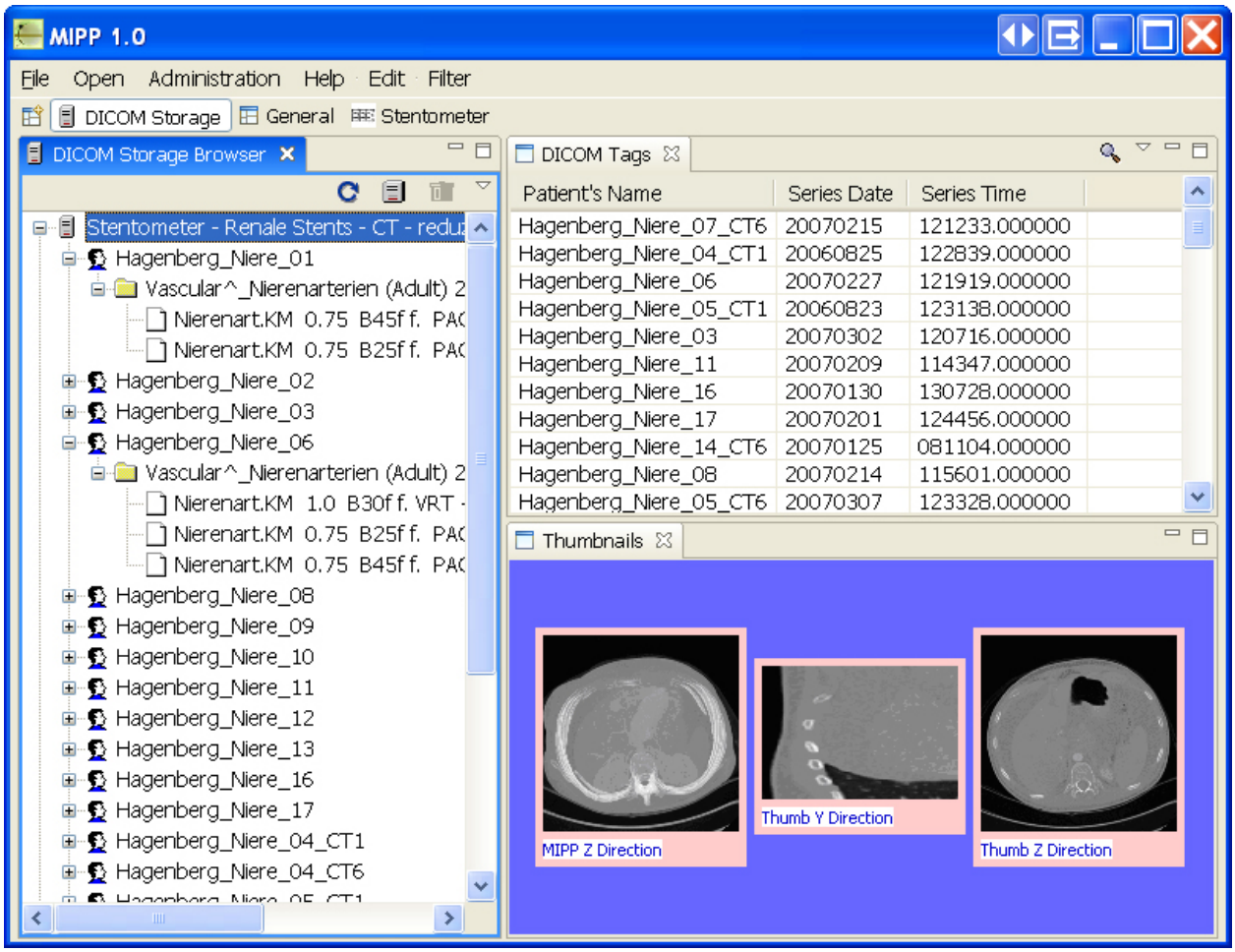

Figure 9: Screenshot of MIPP RCP: DICOM Storage Perspective

For each DICOM storage, a separate node exists in the tree view. The `JDicomTagSets` of a storage are converted into a hierarchical tree of nodes. Grouping is based on DICOM tag keys. The number of levels and the keys to be used at each level can be configured by the user at run-time. Figure 9 shows that DICOM storage ”Stentometer - Renal Stents ...” is configured to have three levels: patient - study - series. The DICOM tag keys used for grouping are: PatientName - StudyDescription, StudyDate, StudyTime - SeriesDescription, SeriesDate, SeriesTime. Another configuration could be: modality - device - patient - series. The hierarchy settings can be specified separately for each DICOM storage. In such a case, the specified default hierarchy settings are overridden.

The same view can usually be opened in more than one perspective. New MIPP RCP applications add new perspectives. Consequently, the `DicomStorageBrowser` view can also be used in application-specific perspectives. Each application might also have its own DICOM storage(s). Instead of always showing all storage nodes, it would be better to show only the application-specific DICOM storage(s) if the view was used in an application-specific perspective. Therefore, the action in the middle of the view’s toolbar (figure 9) allows filtering the DICOM storage nodes to be shown. Each perspective has its own filter settings. The filter concept is implemented using the JFace `ViewerFilter` class and the `IPerspectiveListener` interface.

DICOM Tags View

The `mipp.rcp.dicomstorage.ui.DicomTags` view is responsible for showing the DICOM tags of the series that are selected in the `DicomStorageBrowser` view.

For connecting two parts together (views/editors), RCP provides the *selection service* concept. The part that wants to publish selection changes registers as `ISelectionProvider` at the service, the other one registers as `ISelectionListener` to receive selection changes. In our case, the `TreeViewer` of the `DicomStorageBrowser` registers as `ISelectionProvider`, the `DicomTags` view as `ISelectionListener`.

When the `DicomTags` view is notified, it retrieves the `JDicomTagSet`s of the selected series from the `DicomStorageBrowser`. To avoid re-fetching DICOM tag data from storages, each tree node stores the original `JDicomTagSet`s it aggregates. Custom SWT/JFace widgets were developed for presenting the DICOM tags of one or multiple `JDicomTagSet`s: a list and vertical table widget for the former case and a list, horizontal table, vertical table and tree widget for the latter case. These widgets are reused when realizing the `DicomTags` view.

When multiple `JDicomTagSet`s are available as input, the view allows the user to switch between three layouts/widgets: horizontal table, vertical table and tree. This functionality is added as action to the view's drop-down menu. The view's toolbar contains an action that allows specifying a filter which works on DICOM tag keys.

In the screenshot (figure 9), storage "Stentometer - Renal Stents ..." is selected and the `DicomTags` view shows the DICOM tags for all of its studies. The view uses a horizontal table layout, i.e. one row corresponds to one series. The filter is configured to show only three DICOM tags: PatientsName, SeriesDate and SeriesTime.

Thumbnails View

The purpose of the `mipp.rcp.dicomstorage.ui.Thumbnails` view is to display the thumbnails for a series selected in the `DicomStorageBrowser`. To receive selection changes, the `Thumbnails` view registers as `ISelectionListener` at the selection service. When notified, it queries the thumbnails of the selected series from the corresponding DICOM storage. The returned thumbnails are MIPP `ImageData` objects and have to be converted into SWT images. Already converted thumbnails are stored in a cache for performance reasons.

Which thumbnails are available depends on the thumbnail generators that are registered for a DICOM storage. Each thumbnail generator has a unique name. The `Thumbnails` view retrieves a list of thumbnail generator names that are available for the DICOM storage of the selected series. A filter allows selecting the thumbnail generators whose thumbnails have to be shown.

5.4 General Perspective

The "General" perspective is not associated with a concrete clinical application, but rather a general image viewer. Besides showing the slices and volume rendering of a 3D image, filter operations (registered as top-level menu contributions) can be applied. The initial layout of the perspective is shown in figure 10. Details about the views are described in the following subsections.

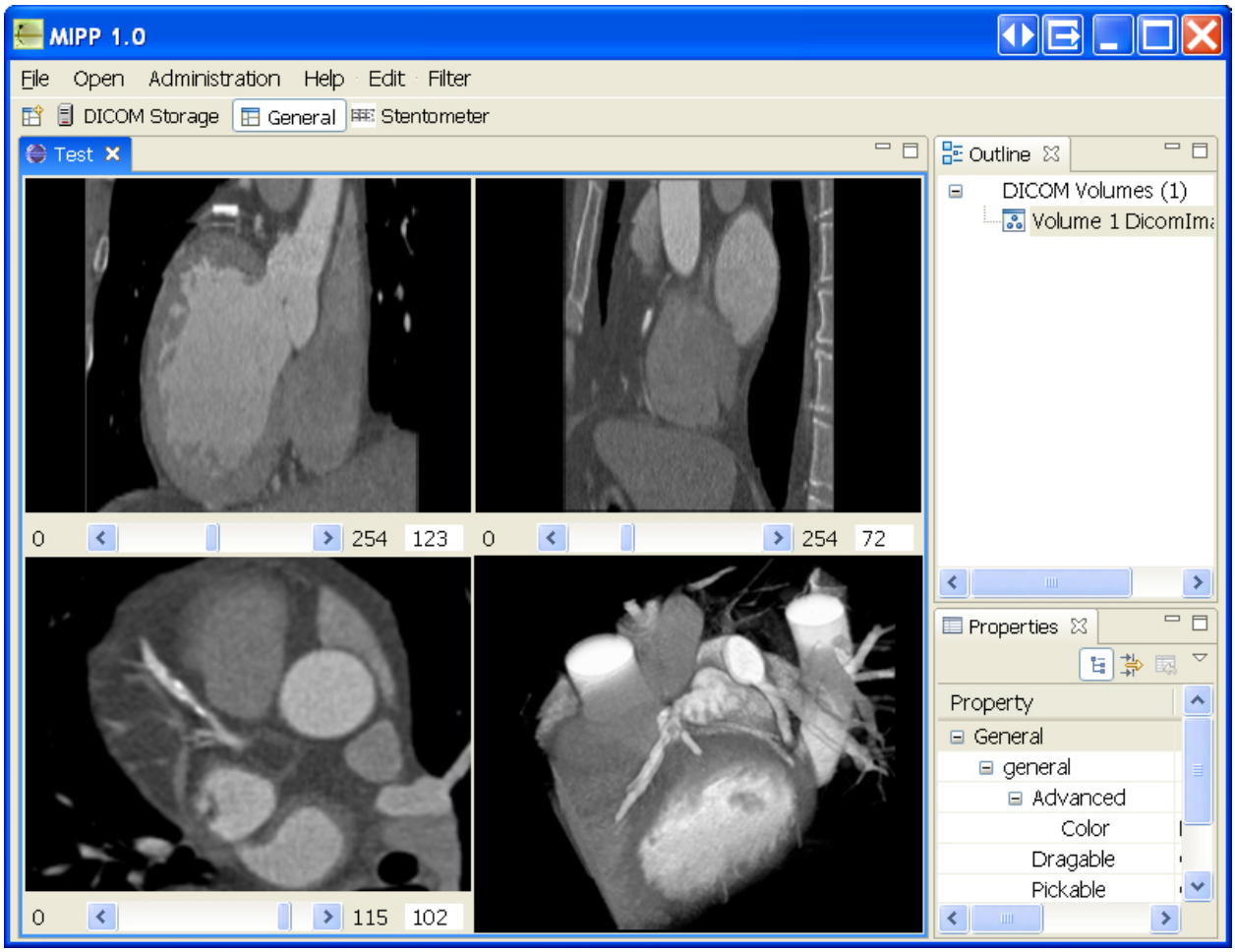

Figure 10: Screenshot of MIPP RCP: General Perspective

Multiple Object Visualisation Editor

The multiple object visualisation (MOV) editor is a very versatile component. As indicated by its name, it allows the visualisation of multiple volumes and polygon models. The editor contains a 3D viewer and a coronal, sagittal and axial 2D viewer. Both kind of viewers differ between camera and actor interaction mode. The former allows the user to zoom, rotate and translate the camera, the latter to translate and rotate the picked object. Thus, the editor already supports manual registration of objects (volume-to-volume, volume-to-mesh, mesh-to-mesh).

If multiple volumes are loaded, the 2D viewers show the slices of each volume as overlay. Another important feature is that the volume to be used for slicing can be selected. In case of windowing and clipping, user interaction is continuously synchronized among the 2D viewers. Synchronization with the 3D view is usually done at the end of the interaction, but can also be done during interaction. The latter is only applicable if the (volume) rendering performance is close to real-time (e.g. due to the use of a VolumePro graphics card).

The order of the 2D and 3D viewers can be rearranged so that users can reproduce the layout they are used to. Instead of displaying all four viewers, the layout can be changed to display only one viewer in the editor. If the editor is additionally maximized in the workbench page the display gets close to full-screen. Note that

multiple instances of an editor with different input can be opened in RCP. This comes in handy if different patients or if follow ups of the same patient have to be examined. However, special care has to be taken of memory to avoid running out of resources.

The screenshot in figure 10 shows the editor with one cardiac CT dataset loaded. Note the clipping in the volume rendering.

Not every feature described above is utilised if the editor is displayed in the general perspective. However, the editor provides enough functionality to be reused in application-specific perspectives.

Scene Graph View

One feature of RCP that is also well-established in the Eclipse IDE is the *outline view*. If a Java text editor is active in the IDE, the view shows the structure of the edited Java class (imports, fields, methods, ...) as a tree. This feature is used in MIPP RCP, too. In case of the MOV editor, the outline view is used as a kind of ”scene graph”, showing the objects that are part of the scene. If the user switches to another (MOV) editor, the view automatically updates its content.

Each object in the scene is associated with a tree node in the scene graph view. The context menu of a tree node is therefore a good location for registering and starting object-specific actions. An important feature of the scene graph view is that also MIPP RCP applications can register their application-specific actions. The following enumeration lists three of the predefined actions that can be added to the context menu of a ”volume tree node”:

- `SliceThroughVolumeAction`: instructs the editor to use the node’s volume for slicing
- `ClipVolumeAction`: starts the MIPP RCP clipping widget for the node’s volume
- `EditTransferFunctionAction`: opens the MIPP RCP transfer function editor for the node’s volume

At the time writing, only two types of object tree nodes are supported: volumes and meshes. Future work will focus on supporting additional types (e.g. volume of interest, contour, contour stack) and even application-specific types (e.g. stents).

Properties View

Another RCP feature that is particularly suitable for MIPP RCP is the *properties view*. The properties view can be compared to a property editor found in GUI builders and is, more or less, a table with two columns (property name and value) and one row for each property. The RCP framework enables the developer to:

- implement properties (`IPropertyDescriptor` interface),
- group properties into categories,
- build hierarchies of categories,
- make properties read-only,
- reuse existing property editors,
- implement custom property editors for complex types.

In case of MIPP RCP, the properties view is bound to the scene graph view selection, i.e. it shows the properties of the scene object that is associated with the selected scene graph tree node. Some properties are common to each property type, but in general, different scene object types have different properties. An example for a type-independent property is visibility. Examples for type-dependent properties are: window level/width, transfer function and lookup table for volumes; colour, opacity and shading interpolation for meshes.

The list of supported properties is not yet complete and will be extended in future. Another feature that will be added is the support for custom properties of custom tree node types,

6 MIPP Software Development Kit

All the binaries and tools necessary for developing a MIPP application are merged into a software development kit (SDK). Release names include the version, platform and architecture for which the SDK was build, e.g. `mipp-1.0-win-x86`. Before installation, the `MIPP_SDK` environment variable has to be set by the user, since each release is installed in the directory where `MIPP_SDK` points to.

The directory tree of the SDK is organized as follows:

- `\3rdparty`: installers of software required/recommended for development (e.g. CMake, Doxygen, Eclipse)
- `\bin`: C++ libraries, dynamic linking, debug and release versions
- `\build`: source code releases of libraries
- `\include`: C/C++ header files
- `\java`: Java libraries, e.g. VTK Java wrapper, MIPP RCP plug-ins, log4j
- `\lib`: C++ libraries, static linking, debug and release versions
- `\doc`: API documentation, additional resources found on the web
- `\tools`: various tools that simplify the development/deployment of MIPP applications

Within these directories, each SDK library has its own subdirectory (e.g. `\include\boost-1.33.1`, `\bin\VTK-5.1_061107\debug`).

Although the binaries are pre-built, there is still the possibility to customize them. The `\build` directory contains the original source code and project files of each library. Ant scripts are available for installing a rebuilt library in the SDK directory tree.

There are two possibilities to import MIPP RCP plug-ins into the Eclipse IDE. The first is to install them from the SDK's `\java` directory which contains a local update site. The second is to download the plug-ins from the web per remote update site and allows developers to regularly check for MIPP RCP updates.

The `\tools` directory includes Visual Studio property sheets for the MIPP platform. These property sheets contain all the compiler and linker settings that are necessary to compile and link against the SDK. The time-consuming task of configuring C++ project settings can be reduced to a minimum since all that has to be done is to add the property sheets to your project.

7 Real-World Applications

The following section briefly introduces research projects that are accomplished based on the MIPP platform and contribute new features in the area of imageprocessing, visualization and GUI design.

7.1 RAPS: Rapid Prototyping in Surgery

Overview

Rapid Prototyping is a term which has been mainly affected by the industry in the past. Today, however, rapid prototyping gains more and more importance in medicine as well. Especially surgery planning is a promising field of application for this technique. Traditional surgery planning is based on images originating from two-dimensional radiography and thus limiting the surgeons scope [3]. By adding a third dimension, the quality of the planning process increases dramatically. This third dimension is typically made available by computed tomography (CT).

The CT device enables the acquisition of three-dimensional volumetric image data in excellent temporary and spatial resolution. The aim of MSCT project is to use the advantages of the CT technology not only for rapid prototyping, but also for liver tumor diagnostics and evaluation of the cardiovascular system.

The aim of the Rapid Prototyping in Surgery (RAPS) project, as part of the MSCT project, is to produce solid models based on the volumetric image data acquired during CT examinations, see figure 11 and 12. To produce such models special rapid prototyping systems are needed. Currently, there are many different prototyping systems available, all of them differ in the procedure and materials to generate output. The most common ones are Selective Laser Sintering (SLS), Stereo Lithography (SLA), 3D milling and 3D printing. SLS uses a laser beam to fuse powdered materials whereas SLA uses a laser beam to harden layers of epoxy resin. 3D printing on the other hand is a combination of conventional inkjet technology and SLA, but instead of ink, 3D printers use liquid treacle and powdered plaster instead of epoxy resin. Compared to SLA and SLS 3D printing is considerably cheap. The produced models, however, have to undergo a special post processing treatment to ensure solid surfaces. Not much attention has been paid to 3D milling in medical applications since this technique is not able to produce cavities [2].

Methods

Staring with a manually performed denture registration, algorithms for automatic registration is applied to further improve results. Automatic registration module is assembled from ITK components. Correlation between fixed and moving volume [`itk::NormalizedCorrelationImageToImageMetric`] is utilized as metric, interpolation is carried out in a linear way and for task of optimization for rigid 3D transform `itk::VersorRigid3DTransformOptimizer` is applied.

Using the developed registration module two image volumes of same morphology and at same scale are efficiently and accurately registered. In the given scenario CT scans of a head and the dental plaster model exhibit different morphology. The difference in morphology results from production process of the dental plaster model. The plaster model forms an imprint in the area of tooth, gingival and palate. The fixation basement at the top and bottom of upper and lower jaw has no morphological correspondent morphology in the CT scan of the head. Consequently these basements would lead to errors during automated registration process. To ensure correct results of good quality, the dental plaster model has to be reduced to the area of tooth and gingival by removing the basements to claim correspondence between fixed and moving volume.

Thereby it is no fundamental condition that all teeth are perfectly extracted from dental plaster scan. Merely few teeth are needed to perform registration process and adjust to match tooth positions in the CT scan of the head.

Results

Detection of theeth is performed in a fully automated way. In this process the following property is utilized: edge detection filters result in a dense area of return intensities around concave and convex shaped teeth. In the area around the tooth density of edge detection return values is much higher than around the planes of the basements to remove. For task of edge detection a classic Sobel kernel of size 3x3x3 is applied. For estimation of edge density morphological operators are used. Iterative use of erosion and dilation reduces the basements whereas the main dental parts remain unchanged.

A further major problem of automated registration is the factor that the dental intensity level of CT volume and plaster model are very different. Intensity adjustment between the fixed and moving volume has to be accomplished.

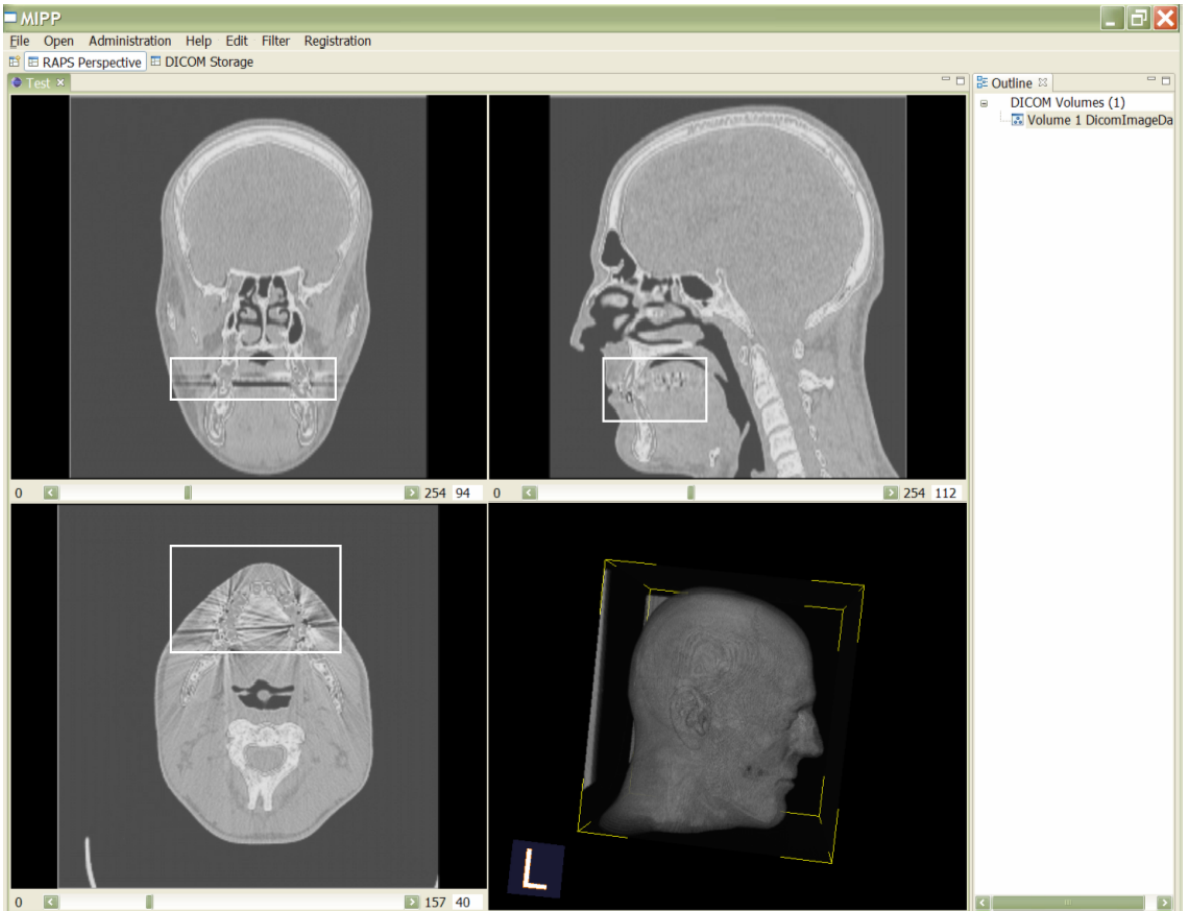

Figure 11: Aritifacts in the area of braces and plmbage

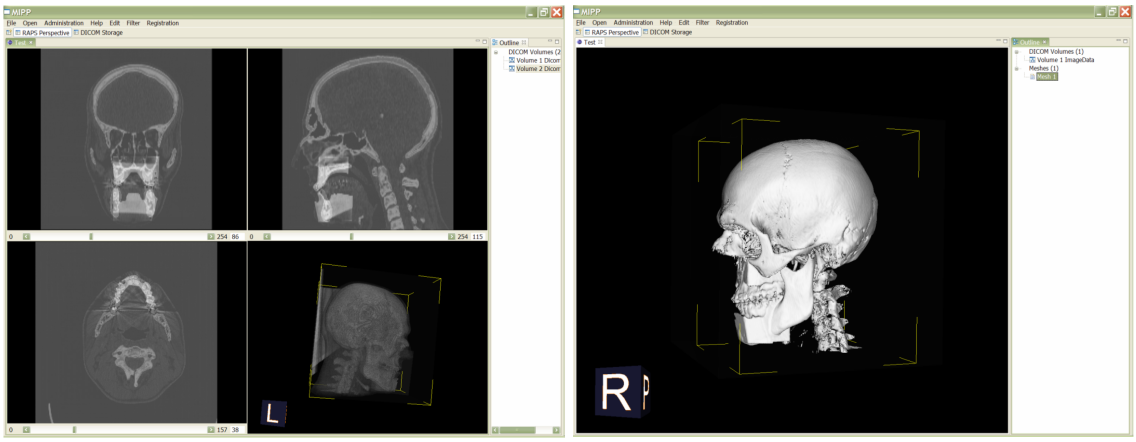

Figure 12: Manual correction of the manual registration results

7.2 LIVIA: Liver Image Analysis

Multi-slice computed tomography (MSCT) allows accurate investigation of liver morphology providing detailed structural information of liver parenchyma, liver vessel trees and even small lesions. Based on the MIPP Platform, algorithms and tools are developed for liver diagnosis, therapy planning and therapy evaluation are developed in the course of LIVIA project [1, 13].

Overview

Liver tumor resection and living liver donation are surgery methods that strongly depend on CT image data for diagnostics and surgery planning. The goal of this project is the development of an computer-based planning software for liver surgery. On the one hand planar CT slices should be assembled to a 3D model that is automatically segmented covering liver parenchyma, main hepatic vessel systems and possible lesions and tumors, allowing a 3D view on the abdominal organs. Based on these binary segmentations measurements like size or dimensions can be measured ad hoc. Further more, liver lobe classification [10, 9] is performed based on accommodative vessel systems, useful for planning tumor resections. Another aspect is tumor measurement and comparison before and after therapy for quantification of therapy success, see figure 13.

Methods

Arterial and venous phase for CT abdominal series are acquired at different points in time and show diverse anatomic structures at good contrast agent saturation. As the patient in bad general condition will not be able to hold-on breathing for the duration of the whole image acquisitions, registration is needed to match arterial and venous phase due to expected motion artifacts.

For the deformable registration task, registration concepts of ITK [4] are applied. Deformable image registration is based on a regular grid that superimposes the pixel grid. The grid points of the mesh are transformed according to minimization of a least squares metric term. The pixel matrix is adapted in the range between registered grid positions using a B-Spline interpolator.

The main research focus lies on the segmentation methods, as segmentation results are the starting basis for all evaluations. Most segmentation approaches currently used in medical practice are semi-automatically

performed and require manual correction for each contour or at least for every stack of 5 contours. Parenchyma segmentation developed in this project is based on level set theory [7, 12]. In contrast to other deformable model / active contour approaches, level sets do not only simulate the moving of some discrete border points under a pressure term F, increasing and reducing the number of control points along the contour according to the changed size, but level sets directly simulate the deformation of the entire contour called ”interface” meeting level set methodology. So problems like the sufficient number of contour control points, collapsing contour points or changes in region topology need no special procedure as they are implicitly handled by the level sets. This means that each problem is transformed into a function of higher dimension by adding time component. The upper example shows propagation of a circular initial surface. LevelSet convergence is reached when no new grid points are reached during a time step (fluid or earthquake stops to expand). In case of threshold LevelSets, image intensities can be interpreted as the topology the LevelSet initial contour expands. Intensities other than the target intensity level lead to erosion of the contour by and by [1].

For liver lobe classification not the segmented vessel tree mask but the graph of vessel centerlines is relevant [11]. In contrast to 2D images, detection of the centerline (thinning, skeletonization) in 3D is not a simple task. Jonker [5] presents an all-purpose approach for 3D object thinning that is based on several iteratively applied shape primitives. But for the task of thinning tubular structures in 3D a specialized and fast algorithm is developed [13]. The algorithm iteratively erodes the region until a thickness of one pixel is reached. Further it is guaranteed that the remaining tree of centerlines remains fully connected. The implemented thinning approach is based on a quite simple concept. For all voxel of a 3x3x3 neighborhood mask a connection check is performed. If the hot spot is needed to connect other voxels in the neighborhood it has to remain, otherwise the hot spot can be removed (if there are redundant voxel trails). The hot spot connectivity test is performed utilizing a hash function for all possible neighborhood permutations. Further parameters ensure that dilation is always performed on the surface of the remaining region. The developed algorithm is robust and results are perfect for vessel volumes. After thinning the vessel tree, graph analysis is used to correct the tree by pruning of artifacts and union of separated vessel courses. The resulting tree can easily be split into 8 fourth level sub-branches of vena porta taking centerline length, child splitting evaluations and the vessel volume into consideration. After extraction of the 8 sub-branches, vena porta vessel volume is partitioned to these branches according to minimum distances. For this partitioning process a dilation concept is used. Tubular expansion is simulated with alternating appliance of simple 3x3x3 mathematical morphology structure elements [13, 14].

Results

A robust and fast segmentation method for liver parenchyma was developed. The algorithm was tested on 16 abdominal CT scans and showed good results. Accuracy was tested and quantified by comparing automated results with manually performed segmentations. The comparison with the manual segmentation showed an average mismatch of voxel of 6.4% (false-positive and false-negative). Median slice mismatch is 6.1%. Overall average difference in volume is only 1.68%. A significant part of the mismatch results from manual reference segmentation performed with MeVisLab (Medical Image Processing and Visualization, University of Bremen, Germany) [8] live wire contour segmentation tool. With arterial slices artifacts of the abdominal rib cage are the main problem due to low contrast differentiation, whereas lower ventricle part leads to some problems segmenting venous CT scans. Those deficiencies will be eliminated with future algorithmic adaptions.

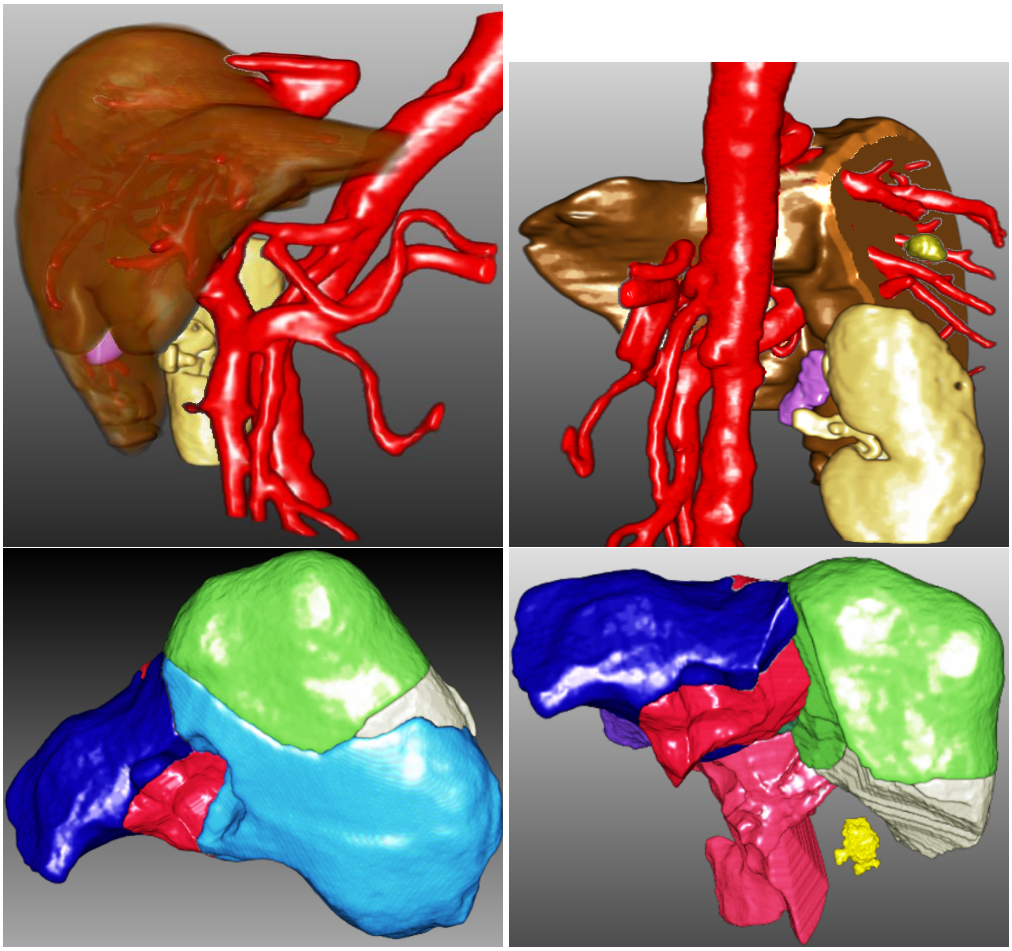

Figure 13: Overview of LIVIA liver analysis. The upper images show 3D segmented parenchyma, the right kidney, the gall bladder, a tumor and the aorta; The lower images show a liver lobe classified parenchyma and the tumor after resecting one lobe. Visualization performed with MeVisLab [8]

7.3 Stentometer: Evaluation of In-Stent-Restenosis based on CT data

Introduction

Over the last two decades the progress in the area of computed tomography was remarkable, from both a medical and technological point of view. Although the human body can already be scanned at sub-millimetre range, modern CT hardware reaches its limitations in some fields of clinical application. One example is the evaluation of coronary in-stent restenosis.

The standard therapy for treating narrowed calcified heart vessels is coronary angioplasty. There, a balloon catheter is guided to the stenosis and inflated in order to dilate the vessel and improve blood flow. A stent, i.e. a small tubular wire mesh, is additionally implanted via the catheter to prevent that the vessel gets narrowed again. However, stenosis often re-occurs within the stent months after the initial treatment. To reveal the existence of this ”in-stent-restenosis”, cardiologists must perform a heart catheter examination again since there is no other diagnostic possibility for the time being.

The ambitious aim of this project is to develop algorithms and a software system (”Stentometer”) that sup-

ports cardiologists/radiologists in the evaluation of in-stent-restenosis based on CT data. The benefit of this approach is that it is non-invasive, as opposed to a conventional heart catheter procedure. Due to their small size coronary stents pose a challenge for current CT equipment (2.5 to 3.5mm stent diameter vs. 0.6mm slice thickness). Image artefacts, introduced by the metallic material and cardiac motion, and the small size make stents poorly visible in CT images. The fact that stents have similar intensity values as plaque further complicates the medical diagnosis and the development of image processing algorithms.

Methods

The geometry of a stent is modelled as a tubular structure. The centreline is a 3D spline whose control points are manually placed in the 2D MIPP viewers by the user with MIPP's point widget. Interpolated points are calculated using the `vtkParametricSpline` class. The length of the centreline corresponds to the length of the stent. Note that the dimension of an implanted stent, i.e. its length and inflated radius, is usually known by clinicians. The radius along the centreline is expected to be constant for coronary stents. Since this might not be the case for other stents, the model allows specifying a different radius at each control point as well.

Stentometer uses VTK to visualize a stent as tubular structure or as a mesh of struts, with or without volume rendering (see figure 14). Combining polygon and volume rendering helps the user to visually check the manual segmentation. Both the tube and the struts are generated using `vtkParametricSpline` and `vtkTubeFilter`.

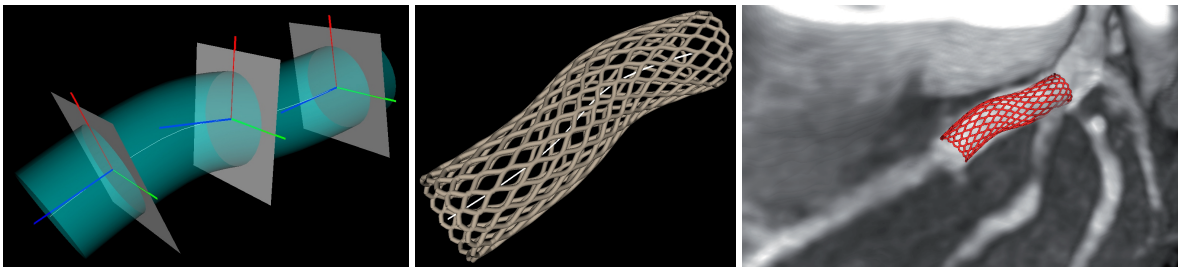

Figure 14: Screenshots of Stentometer prototype. Left: model of stent geometry, middle: stent polygon model with struts, right: visualization of stent in volume-rendered CT image.

Based on the stent model and the image data, a curved multi-planar reformation can be performed with Stentometer. The axes of the cross-sectional planes orthogonal to the centreline are calculated utilizing `vtkTubeFilter`. Resampling is done in ITK (`itk::ResampleImageFilter`) rather than in VTK, since ITK provides more advanced interpolation methods. The cross-sectional slices help cardiologists to assess the opacity of a stent.

Another usage of the stent model is the derivation of a binary mask: all voxels that lie within the tube are set to scalar value one. As expected, the histogram of a stent's bounding box show significant differences compared to the histogram based on the binary mask.

The diagnostic features realized so far are mainly based on visual and histogram analysis. At the time writing, the project is still under progress. Future work will focus on more algorithmic approaches.

7.4 Teaching Context

Besides the application of MIPP platform in the field of medical research projects, the importance for the utilization in the teaching and lecture work at the University of Applied Sciences Upper Austria is increasing.

Parts of the MIPP have been the fundament in three student projects by now and a VTK introduction lecture has been included in the studies curriculum the last term.

8 Conclusion and Future Work

In this paper a novel platform for realizing prototypes/applications in the field of medical image processing, analysis and visualization is presented: the MIPP platform. By reusing rich and well-established libraries and frameworks, a lot of functionality is available from scratch for efficiently developing a sophisticated prototype. The implementation of three prototypes addressing real-world problems in clinical practice illustrated the potential of the platform.

ITK and VTK offer state-of-the-art image processing and visualization algorithms and their design supports the integration of custom functionality. Eclipse RCP, and consequently MIPP RCP too, provide everything for a plug-in based software architecture and for realizing a rich client application with native look and feel. The plug-in based architecture facilitates modularity, reusability and extensibility. Reusing the implemented MIPP RCP plug-ins relieves the developer from writing time-consuming code for features usually found in medical software. For the integration of new, application-specific features, well-defined extension points are defined. Researchers may contribute their algorithms by writing new plug-ins and make these available to others. Java and its third-party libraries can be used to introduce even more features (e.g. database access, report generation, charts).

Another benefit of the MIPP platform is that its functionality grows with the functionality of the reused libraries and their third-party extensions. Indeed, there are a lot of extensions already available for ITK and VTK. Reusing well-established libraries/frameworks not only reduces the design and programming task, but also ”outsources” testing, bug fixing and writing documentation, all making the maintenance of the platform much more easier.

A drawback of reusing large software frameworks like ITK, VTK and RCP is the effort for becoming acquainted with them. The MIPP SDK documentation package tries to ease this effort by providing a collection of resources for programmers. Another benefit of the SDK is that it is a prepared ”development package” that reduces the effort for choosing, compiling and organizing the libraries to be reused for development.

One of the challenges was to bring C++ and Java together. For that matter, SWIG has proven to be an efficient tool for the automated generation of Java wrapper. However, a lot of details about memory management, event handling and threading were omitted in this paper for the sake of lucidity. Note that VTK needn’t be wrapped since it already provides Java wrapper.

In future, further research and student projects at our university will be implemented based on the MIPP platform. Functionality that could be reusable later will be separated from application-specific code and added to the platform as MIPP RCP plug-ins.

The current design suggests that the functional layer of a MIPP application should be implemented in C++. If only existing ITK and VTK classes need to be reused and no new C++ classes are introduced, it will probably be simpler to write the functional layer in Java. This implies that Java wrapper need to be available for both VTK and ITK. As mentioned before, VTK already provides Java wrapper. In case of ITK, CableSwig can be used for wrapper generation. This option has not yet been considered and will probably be worked on in future.

Due to limited time resources, only the Windows x86 version of MIPP is currently supported. Future work will focus on porting MIPP to other platforms, mainly Windows x64, Linux x86 and x64. Preliminary

attempts showed that MIPP and all of its libraries could be successfully compiled on Gentoo Linux x64. Note that the effort for porting MIPP to other platforms should be rather small, since the reused C++ libraries and Java already consider portability.

This paper intentionally gives only an outline of the Medical Image Processing Platform. For the sake of lucidity, a lot of details were omitted, e.g.: implementation of the application storages, the 2D/3D viewers and their synchronization, the widgets and their synchronisation, C++/Java memory management and event handling issues, etc. Some of the details will presumably be topic of future publications.

References

[1] W. Backfrieder, G. Zwettler, R. Swoboda, F. Pfeifer, H. Kratochwill, and F. Fellner. Towards automated segmentation and classification of liver lesions for diagnosis and surgery planning. In *International Journal of Computer Assisted Radiology and Surgery*. CARS2007, 2007. 7.2, 7.2

[2] VE. Beal, CH. Ahrens, and PW. Wendhausen. The use of stereolithography rapid tools in the manufactoring of metal powder injection modeling parts. *J. of the Braz. Soc. of Mech. Sci & Eng.*, 26(1):40–46, 2004. 7.1

[3] ELS. daRosa, CF. Oleskovicz, and BN. Aragao. Rapid prototyping in maxillofacial surgery and traumatology : Case report. *J. of the Braz. Soc. of Mech. Sci & Eng.*, (15):243–247, 2004. 7.1

[4] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second edition, 2005. 7.2

[5] PP. Jonker. Skeletons in n dimensions using shape primitives. Number 23, pages 677–686. Pattern Recognition Letters 2002, 2002. 7.2

[6] J. McAffer and JM. Limeux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison Wesley, 2005. 5.2

[7] S. Osher and SA. Sethian. Fronts propagating with curvature-dependent speed. *Journal of Computational Physics*, (79):12–49, 1988. 7.2

[8] MeVis Research. *MeVisLab*. MeVis Research, University of Bremen, http://www.mevislab.de/index.php, 2007. 7.2, 13

[9] S. Rutkauskas, V. Gedrimas, J. Pundzius, G. Barauskas, and A. Basevicius. Clinical and anatomical basis for the classification of the structural parts of the liver. Medicina (Kaunas) 2006, 2006. 7.2

[10] A. Schaeffler and N. Menche. *Biologie Anatomie Physiologie*. Urban & Fischer, 2000. 7.2

[11] D. Selle, W. Spindler, B. Preim, and HO. Peitgen. Mathematical methods in medical image processing: Analysis of vascular structures for preoperative planning in liver surgery. pages 1039–1059. Springer, 2000. 7.2

[12] JA. Sethian. Level set methods and fast marching methods. *Cambridge Monographs on Applied and Computational Mathematics*, 1999. 7.2

[13] G. Zwettler, W. Backfrieder, R. Swoboda, F. Pfeifer, H. Kratochwill, and F. Fellner. Automatic liver classification with multi-slice ct data. 2007. 7.2, 7.2

[14] G. Zwettler, R. Swoboda, F. Pfeifer, and W. Backfrieder. Accelerated skeletonization algorithm for tubular structures in large datasets by randomized erosion. 2008. 7.2

A Appendix: mipp::image::ImageData

The following code snipped delineates the design of `mipp::image::ImageData`:

```
#ifndef __MIPP_IMAGE_IMAGE_DATA_H__
#define __MIPP_IMAGE_IMAGE_DATA_H__

//pre compiler headers for enabling all supported image types
//not activated for development compilation speed
#ifndef _MIPP_ALL_IMAGETYPES_SUPPORT__
#define _MIPP_ALL_IMAGETYPES_SUPPORT__
#endif
//-------------------------------------------------

//MSCT library includes
#include "mipp/common/InvalidArgumentException.h"

#include "mipp/image/ImageInfo.h"
#include "mipp/itk/ImageIOSupportTypes.h"

#include "itkMetaDataDictionary.h"

//C++ library includes
#include <iostream>
#include <string>

//boost library includes
#include "boost/variant.hpp"

//vtkMov library includes
//#include "vtk/Mov/mippVisualizationData.h"

//insight application includes to convert between ITK and VTK
#include "itkImageToVTKImageFilter.h"
#include "itkVTKImageToImageFilter.h"


//forward declarations
namespace mipp {
  namespace common {
    class InvalidArgumentException;
  } //namespace common
} //namespace mipp
```

```
namespace mipp {
  namespace image {
    class MetaDataDictionary;
  }
  namespace itk {
    class HistogramData;
    class ImageBuffer;
    class ImageBuffer3D;
  } //namespace itk
} //namespace mipp

//vtk forwards
class vtkImageCast;
class vtkImageData;


namespace mipp {
  namespace image {

    //! \class ImageData
    //! \brief wrapper for templated itk::Image instances
    //
    //! as there can be no base class to use for templated instance inheritance,
    //! this class ha to handle all supported image types and remember which
    //! image is set at each point in time for use in filter libraries this
    //! strategy leads to several advantages as instances of this class
    //! can be used as quasi generic image objects
    //! \author Gerald Zwettler
    //! \version 1.2
    //! \date created: 28.07.2006 last modified: 04.09.2006

    class MIPP_IMAGE_EXPORT ImageData {
    private:
      typedef boost::variant<mipp::itk::ImageIOSupportTypes::US3_Image::Pointer,
                             mipp::itk::ImageIOSupportTypes::UC3_Image::Pointer,
                             mipp::itk::ImageIOSupportTypes::SS3_Image::Pointer,
                             ...> VariantImageType;
      //typedef for boost has to be private as this construct shouldn't be wrapped

    public: //enum
      //! type of the itk::Image stored in this class
      enum ImageTypes {NOT_SET = -1, US3 = 0, UC3 = 1, SS3 = 2, SC3 = 3,
                       ...};


      //typedefs for conversion between itk and vtk
```

```
  typedef ::itk::ImageToVTKImageFilter<mipp::itk::ImageIOSupportTypes::US3_Image>
     ITKVTKPipeline_US3Type;
  typedef ::itk::ImageToVTKImageFilter<mipp::itk::ImageIOSupportTypes::UC3_Image>
     ITKVTKPipeline_UC3Type;
  typedef ::itk::ImageToVTKImageFilter<mipp::itk::ImageIOSupportTypes::SS3_Image>
     ITKVTKPipeline_SS3Type;
  ...

  typedef ::itk::VTKImageToImageFilter<mipp::itk::ImageIOSupportTypes::US3_Image>
     VTKITKPipeline_US3Type;
  typedef ::itk::VTKImageToImageFilter<mipp::itk::ImageIOSupportTypes::UC3_Image>
     VTKITKPipeline_UC3Type;
  typedef ::itk::VTKImageToImageFilter<mipp::itk::ImageIOSupportTypes::SS3_Image>
     VTKITKPipeline_SS3Type;
  ...

public: //member functions
  ImageData();  //constructor

  ~ImageData(); //destructor

  ImageData(mipp::itk::ImageIOSupportTypes::US3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  ImageData(mipp::itk::ImageIOSupportTypes::UC3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  ImageData(mipp::itk::ImageIOSupportTypes::SS3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  ...

  void SetUS3_Image(mipp::itk::ImageIOSupportTypes::US3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  void SetUC3_Image(mipp::itk::ImageIOSupportTypes::UC3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  void SetSS3_Image(mipp::itk::ImageIOSupportTypes::SS3_Image* image)
     throw(mipp::common::InvalidArgumentException);
  ...

  mipp::itk::ImageIOSupportTypes::US3_Image::Pointer GetUS3_Image();
  mipp::itk::ImageIOSupportTypes::UC3_Image::Pointer GetUC3_Image();
  mipp::itk::ImageIOSupportTypes::SS3_Image::Pointer GetSS3_Image();
  ...

  const mipp::itk::ImageIOSupportTypes::US3_Image::Pointer GetUS3_Image() const;
  const mipp::itk::ImageIOSupportTypes::UC3_Image::Pointer GetUC3_Image() const;
  const mipp::itk::ImageIOSupportTypes::SS3_Image::Pointer GetSS3_Image() const;
  ...

  //only for 2D image data instances
```

```
  mipp::itk::ImageBuffer* GetImageBuffer();

  //only for 3D ImageData instances
  mipp::itk::ImageBuffer3D* GetImageBuffer3D();

  ImageData* GenerateThumbnailImage(int direction, int sliceNum) const;

  mipp::itk::HistogramData* GetHistogramData();

  bool IsNull(); //=> is valid data set ? => false, if itk::Image is NULL

  bool IsVolume(); // => is valid data for one of the 4 3D image types set ??

  int GetNumOfSlices();


  //mippVisualizationData* GetVisualizationData();
  //const mippVisualizationData* GetVisualizationData() const;
  //void SetVisualizationData(mippVisualizationData* visualizationData);
  vtkImageData* GetVisualizationData();
  const vtkImageData* GetVisualizationData() const;
  void SetVisualizationData(vtkImageData* input);

  bool DataAsserted() const;
  int GetImageType() const;

  const std::string GetImageTypeAsString() const;

  static int GetImageType(
     mipp::image::ImageInfo::ImageDataComponentType pixelType,
     unsigned int dimension);

  void SetImageType(ImageTypes imageType);

  const MetaDataDictionary& GetMetaDataDictionary() const;
  void SetMetDataDictionary(MetaDataDictionary& metaDataDictionary);


 //-- methods needed to convert between vtkImageData and ITK images
  void UpdateVariantImage();
  void UpdateMippVisualizationData();
  void UpdateImageCaster();
 //-----------------------------------------------------------------------


protected:
  //print out content of ImageInfo instance
  virtual std::ostream& PrintOut(std::ostream &out);
```

```
    private: //member functions

      ImageData(const ImageData& other);

      //overloaded output operator
      friend std::ostream& operator<<(std::ostream &out, ImageData &data);

    public:
      //cast between vtk and itk ---------
       vtkImageCast* cast;

       //data representation
      VariantImageType variantImage;

    private: //data members
      ImageTypes imageType;


      //visualization representation via vtkObject for multiple object visualization
      //mippVisualizationData* visualizationData;
      vtkImageData* visualizationData;

       //itk <<>> vtk converter for all supported types
       VTKITKPipeline_US3Type::Pointer vtkPipeline_US3;
       VTKITKPipeline_UC3Type::Pointer vtkPipeline_UC3;
       VTKITKPipeline_SS3Type::Pointer vtkPipeline_SS3;
       ...

     ITKVTKPipeline_US3Type::Pointer itkPipeline_US3;
       ITKVTKPipeline_UC3Type::Pointer itkPipeline_UC3;
       ITKVTKPipeline_SS3Type::Pointer itkPipeline_SS3;
       ...

    }; //class ImageData

    inline std::ostream& operator<< (std::ostream &out, ImageData &data) {
      return data.PrintOut(out);
    } //operator<<

  } //namespace image
} //namespace mipp

#endif //__MIPP_IMAGE_IMAGE_DATA_H__
```

B Appendix: BilateralImageFilter example implementation

The following code snipped presents itk::BilateralImageFilter composition in mipp::itk::ImageFilters used in mipp::imageprocessing:BasicImageFilterLibrary for single call filter methods with mipp:image::Imagedata input data:

```
template <class T, class T_p>
  T_p ImageFilters<T, T_p>::BilateralImageFilter(T* inputImage, double rangeSigma,
                      double* domainSigmas, int kernelSize, ProgressHandler* ph) {

    typedef ::itk::BilateralImageFilter<T, T> BilateralFilterType;
    typename BilateralFilterType::Pointer filter = BilateralFilterType::New();
  MIPP_ITK_PROGRESSHANDLER_OUTER_CALL_CHECK(ph)
  MIPP_ITK_PROGRESSHANDLER_REGISTER_PROCESS_OBJECT(ph, filter, 1.0)

    const unsigned int dimension = T::ImageDimension;

    filter->SetInput(inputImage);
    filter->SetDomainSigma(domainSigmas);
    filter->SetRangeSigma(rangeSigma);
    filter->AutomaticKernelSizeOff();
    filter->SetRadius(kernelSize);

  MIPP_ITK_TRYCATCHSTART
    filter->Update();
  MIPP_ITK_TRYCATCHEND
  MIPP_ITK_PROGRESSHANDLER_CHECKFORABORTION(ph, inputImage)

    return filter->GetOutput();
  }//BilateralImageFilter
```

C Appendix: BilateralImageFilter in imageprocessing library

```
T_p ImageFilters<T, T_p>::BilateralImageFilter(T* inputImage, double rangeSigma,
                      double* domainSigmas, int kernelSize, ProgressHandler* ph) {

//---- typedefs from ImageFilters.h ----
typedef mipp::itk::ImageFilters<mipp::itk::ImageIOSupportTypes::US3_Image,
    mipp::itk::ImageIOSupportTypes::US3_Image::Pointer> US3_ImageFilters;
typedef mipp::itk::ImageFilters<mipp::itk::ImageIOSupportTypes::UC3_Image,
    mipp::itk::ImageIOSupportTypes::UC3_Image::Pointer> UC3_ImageFilters;
typedef mipp::itk::ImageFilters<mipp::itk::ImageIOSupportTypes::SS3_Image,
    mipp::itk::ImageIOSupportTypes::SS3_Image::Pointer> SS3_ImageFilters;
```

```
...

   void BasicImageFilterLibrary::BilateralImageFilter(
        mipp::image::ImageData* inputImage,
        double rangeSigma, double* domainSigmas, int kernelSize,
        mipp::itk::ProgressHandler* ph) {

      switch(inputImage->GetImageType()) {
        case ImageData::SS3 :
          inputImage->SetSS3_Image(SS3_ImageFilters::BilateralImageFilter(
            inputImage->GetSS3_Image(), rangeSigma, domainSigmas, kernelSize, ph));
          break;
#ifdef _MIPP_ALL_IMAGETYPES_SUPPORT__
        case ImageData::SC3 :
          inputImage->SetSC3_Image(SC3_ImageFilters::BilateralImageFilter(
            inputImage->GetSC3_Image(), rangeSigma, domainSigmas, kernelSize, ph));
          break;
        case ImageData::UC3 :
          inputImage->SetUC3_Image(UC3_ImageFilters::BilateralImageFilter(
            inputImage->GetUC3_Image(), rangeSigma, domainSigmas, kernelSize, ph));
          break;

        ...

#endif
        default:
          MIPP_THROW_EXCEPTION(InvalidArgumentException, "unknown image type");
      } //switch

    } //BilateralImageFilter
```