Amira 5.5

Developer's Guide

Amira® 5.5

Amira Developer Option User's Guide

Intended Use
Amira ® is intended for research use only. It is not a medical device.
Copyright Information

Trademark Information:

Contents

1 Introduction to Developer Option **1**

1.1 Overview of Developer Option 1

1.1.1 Packages and Shared Objects 1

1.1.2 Package Resource Files 2

1.1.3 The Local Amira Directory 2

1.1.4 External Libraries 2

1.2 System Requirements 3

1.2.1 Windows 4

1.2.2 Linux 4

1.2.3 Mac 4

1.3 Structure of the Amira File Tree 4

1.3.1 The Amira Root Directory 5

1.3.2 The Local Amira Directory 5

1.4 Quick Start Tutorial 6

1.5 Compiling and Debugging 9

1.5.1 Windows Visual Studio 2005/2008 9

1.5.2 Linux 11

1.6 Maintaining Existing Code 12

1.6.1 Upgrading to Developer Option 5 12

1.6.2 Renaming an Existing Package 13

2 The Development Wizard **15**

2.1 Starting the Development Wizard 15

2.2 Setting Up the Local Amira Directory 16

2.3 Adding a New Package 18

2.4 Adding a New Component 19

2.5 Adding an Ordinary Module 19

2.6 Adding a Compute Module 20

2.7 Adding a Read Routine 21

2.8 Adding a Write Routine 22

2.9 Creating the Build System Files 23
2.10 The Package File Syntax 25

3 File I/O **29**
3.1 On file formats 29
3.2 Read Routines 30
3.2.1 A Reader for Scalar Fields 31
3.2.2 A Reader for Surfaces and Surface Fields 35
3.2.3 More About Read Routines 39
3.3 Write Routines 41
3.3.1 A Writer for Scalar Fields 42
3.3.2 A Writer for Surfaces and Surface Fields 44
3.4 The AmiraMesh API 49
3.4.1 Overview 49
3.4.2 Writing an AmiraMesh File 51
3.4.3 Reading an AmiraMesh File 52

4 Writing Modules **55**
4.1 A Compute Module 55
4.1.1 Version 1: Skeleton of a Compute Module 56
4.1.2 Version 2: Creating a Result Object 59
4.1.3 Version 3: Reusing the Result Object 62
4.1.4 Implement an ITK image to image filter 64
4.2 A Display Module 70
4.2.1 Version 1: Displaying Geometry 71
4.2.2 Version 2: Adding Color and a Parse Method 76
4.2.3 Version 3: Adding an Update Method 78
4.3 A Module With Plot Output 80
4.3.1 A Simple Plot Example 81
4.3.2 Additional Features of the Plot API 85

5 Data Classes **87**
5.1 Introduction 87
5.1.1 The Hierarchy of Data Classes 87
5.1.2 Remarks About the Class Hierarchy 89
5.2 Data on Regular Grids 91
5.2.1 The Lattice Interface 92
5.2.2 Regular Coordinate Types 95

5.2.3 Label Fields and the Label Lattice Interface 97
5.2.4 Color Fields 98
5.3 Unstructured Tetrahedral Data 99
5.3.1 Tetrahedral Grids 99
5.3.2 Data Defined on Tetrahedral Grids 101
5.4 Unstructured Hexahedral Data 102
5.4.1 Hexahedral Grids 102
5.4.2 Data Defined on Hexahedral Grids 104
5.5 Other Issues Related to Data Classes 105
5.5.1 Procedural Interface for 3D Fields 105
5.5.2 Transformations of Spatial Data Objects 106
5.5.3 Parameters and Materials 108

6 Documentation of Modules in Developer Option 111
6.1 The Documentation File 111
6.2 Generating the Documentation 113

7 Documentation Markup Language Reference Guide 115
7.1 Document Structure 116
7.1.1 Document Header: hxmodule, etc. 116
7.1.2 Table of Contents 119
7.1.3 Sections 120
7.1.4 Links & Labels 126
7.2 Text Styles 130
7.2.1 Text Formatting 130
7.2.2 Special Text-producing Commands 132
7.2.3 Lists & Enumerations 138
7.2.4 FAQ Environment 139
7.2.5 Quotes 140
7.2.6 Verbatim 140
7.3 Displaying Images and Tables 141
7.3.1 Figures 141
7.3.2 \hximage and \hximagescaled 142
7.3.3 Tables 143
7.4 Math 145
7.4.1 Inline 145
7.4.2 Equations 145

7.5 Meta-Level Commands 146
7.5.1 Conditions Using \ifdefined 146
7.5.2 Differentiate between PDF and HTML Output 147
7.5.3 Include Different Input Files 147
7.5.4 Exclude Files 148
7.5.5 Comments 148
7.6 CSS Attributes 149
7.6.1 Headlines 149
7.6.2 Front Page (index.doc) 149
7.6.3 hx Environments 150
7.6.4 Tables 152

8 Miscellaneous 153
8.1 Time-Dependent Data And Animations 153
8.1.1 Time Series Control Modules 153
8.1.2 The Class HxPortTime 154
8.1.3 Animation Via Time-Out Methods 156
8.2 Important Global Objects 157
8.3 Save-Network Issues 158
8.4 Using version information 160
8.5 Troubleshooting 161
8.5.1 Compile-Time Problems 161
8.5.2 Run-Time Problems 162
8.5.3 Debugging Problems 163

1 Introduction to Developer Option

Developer Option allows you to add to Amira new components such as file read or write routines, modules for visualizing data or modules for processing data. New module classes and new data classes can be defined as subclasses of existing ones. Note that it is not possible (or possible only to a very limited extent) to change or modify existing modules or parts of Amira's graphical user interface.
In the following sections we

- present an *overview of Developer Option*,
- discuss the *system requirements* for the different platforms,
- outline the *structure of the Amira file tree*,
- show how to compile the demo package in a *quick start tutorial*,
- provide additional hints on *compiling and debugging*,
- and mention how to *upgrade and maintain existing code*.

1.1 Overview of Developer Option

Developer Option is an extension to the ordinary Amira version. In addition to the files contained in the ordinary version, the developer version essentially provides all C++ header files needed to compile custom extensions.

1.1.1 Packages and Shared Objects

Amira is an object-oriented software system. Besides the core components like the graphical user interface or the 3D viewers, it contains a large number of data objects, modules, readers and writers. Data objects and modules are C++ classes, readers and writers are C++ functions.
Instead of being compiled into a single static executable, these components are grouped into *packages*. A package is a shared object (usually called .so or .sl on Unix or .dll on Windows) which can be dynamically loaded by Amira at run time when needed. This concept has two advantages. On the one hand, the program

remains small since only those packages are loaded which are actually needed by the user. On the other hand it provides almost unlimited extensibility since new packages can be added any time without recompiling the main program.
Therefore, in order to add custom components to the Amira developer version, new packages or shared objects must be created and compiled. A package may contain an arbitrary number of modules and it is left up to the developer whether he wants to organize his modules into several packages or just in one.

1.1.2 Package Resource Files

Along with each package a *resource file* is stored. This file contains information about the components being defined in a particular package. When Amira starts, it first scans the resource files of all available packages and thus knows about all the components which may be used at run-time.
The resource files of the standard Amira packages are located under `share/resources` in the directory where Amira is installed. Details about registering read and write routines or different kinds of modules in a resource file are provided in Chapters 3 and 4.

1.1.3 The Local Amira Directory

Usually Amira will be installed by the system administrator at a location where ordinary users are not allowed to create or modify files. Therefore it is recommended that every user creates new packages in his own personal *local Amira directory*. The local Amira directory has essentially the same structure as the directory where Amira is installed. A new local Amira directory can most easily be created by using the *Development Wizard*, a special-purpose dialog box described in detail in Section 2.
Once a local Amira directory has been set up, resource files located in it will also be scanned by Amira when started. In this way new components can be added or existing ones redefined.

1.1.4 External Libraries

Amira is based on a number of industry standard libraries. The most important ones are *Open Inventor*, *OpenGL*, *Qt*, and *Tcl*.
Amira's 3D graphics is based on OpenGL and Open Inventor. OpenGL is the industry standard for professional 3D graphics. Open Inventor is a C++ library using

OpenGL which provides an object-oriented scene description layer. Writing new visualization modules for Amira essentially means creating an Open Inventor scene from the input data. If you already have code doing this, it will be straightforward to turn it into an Amira module. A subset of the Open Inventor headers is included in Developer Option, OpenGL must already be installed on your system.

Qt is a platform-independent C++ library for building graphical user interfaces (GUIs). Amira is built with Qt. However, the user interface elements used in standard Amira modules are encapsulated by special Amira classes called *ports*. Therefore you can develop your own modules without knowing Qt. You only need Qt if you plan to add completely new user interface components such as special purpose dialogs.

Finally, Tcl is a C library providing an extensible scripting language used by Amira. All required header files are included in Developer Option. Detailed knowledge of the Tcl API is usually not needed, as most of the code can be derived from existing examples.

1.2 System Requirements

In order to develop new Amira components as described in this document you need the developer extension for Amira (called Developer Option) as well as a C++ development environment. C++ compilers, however, are generally not compatible, therefore the compilers and compiler versions listed below should be used. Other compiler versions may work too, but this is not guaranteed. In particular, it is not possible to use the GNU gcc compiler except on Linux or Mac.

On all Unix platforms the GNU make utility (`gmake`) is needed in order to use the GNUmakefiles provided with Developer Option. To proceed you should create a link in a directory already listed in your path, e.g., in `/usr/bin`.

On Mac Os X platforms the GNU make utility (`gmake`) is basically needed in order to use the GNUmakefiles provided with Developer Option. You can also modify and build your code by using other development tools based on GNU gcc compilers as Xcode.

In the following text more specific system requirements are listed for each platform. More general hardware requirements such as installed memory or special graphics adapters are listed in the Amira *User's Guide*. On all systems an OpenGL library together with the OpenGL header files must be installed.

1.2.1 Windows

Operating system: Windows XP(SP3) or newer
Compiler: Microsoft Visual Studio 2005(SP1) (VC++ 8.0)

Operating system: Windows XP x64 Edition(SP3) or newer
Compiler: Microsoft Visual Studio 2008 (VC++ 9.0)

Note: Which compiler do you need depends on the version of Amira you have. You can obtain the version information by typing `app uname` into the Amira console.

1.2.2 Linux

Operating system: Red Hat Enterprise Linux 5.5 or compatible
Compiler: GNU gcc 4.1.x

Use `gcc -version` to find out the version of the GNU compiler.

1.2.3 Mac

Operating system: MacOS X 10.5/10.6/10.7
Compiler: GNU gcc 4.2.x

GNU gcc 4.2.x is also available with the latest version of XCode development environment provided by Apple.

1.3 Structure of the Amira File Tree

Like the ordinary version, the developer version of Amira (with Developer Option) is installed in a single directory called the *AmiraRoot Directory*. This directory contains all the binaries, shared objects, and resource files required to run Amira, as well as all the C++ header files required to compile new components. New components themselves are stored independently in the *AmiraLocal Directory*. Every user may define his/her own local Amira directory. The local Amira directory has a structure very similar to the AmiraRoot Directory. In the following two sections the structure of these two directories is described in more detail.

1.3.1 The Amira Root Directory

The contents of the Amira root directory may differ slightly from platform to platform. For example, on Windows there will be no subdirectory `lib`. Instead, the compiled shared objects are located under `bin/arch-Win32VC8-Optimize`. The online documentation directory (`share/devref/`) of Amira C++ classes does not exist on Windows. Instead, a compressed archive file `Amira.chm` is provided and is accessible by shortcut. This is how a typical Amira installation directory looks like:

```
AmiraRoot/ ........................................ Amira installation directory
 └─ bin/
     └─ arch-Win32VC8-Debug/ .......... Amira debug binaries (Windows only)
         └─ Amiramain.exe............. Amira debug executable (Windows only)
     └─ arch-Win32VC8-Optimize/ ................ Amira binaries (Windows)
         └─ Amiramain.exe ....................... Amira executable (Windows)
     └─ start .......................................... Amira start script (Unix)
 └─ include/ ................................................. C++ header files
 └─ make/.................................... Make environment for Unix systems
 └─ share/
     └─ resources/ ...................... Resource files of all standard packages
     └─ devref/ .......................... Documentation of Amira C++ classes
     └─ doc/.............................................. Amira documentation
```

1.3.2 The Local Amira Directory

The local Amira directory contains the source code and object files of custom modules, the resource files of custom packages, and the compiled custom packages themselves. The packages can be compiled either in a debug version or in an optimized version. The corresponding object files and compiled shared objects reside in different subdirectories called `arch-*-Debug` and `arch-*-Optimize`, respectively. Here the asterisk denotes the particular architecture, e.g., `Win32VC8`, `Win64VC9` for Windows systems or `LinuxAMD64` for Linux platforms.

In order to create a new local Amira directory the *Development Wizard* should be used. For details please refer to Section 2. Subdirectories like `AmiraLocal/lib` or `AmiraLocal/share/resources` are created automatically the first time a custom package is compiled. Again, the contents of the local Amira directory may differ slightly from platform to platform. For example, on Windows compiled shared objects are located under `bin` instead of `lib`.

```
AmiraLocal/
|__ AmiraLocal.vc80.sln ................ Visual Studio solution file (Windows)
|__ GNUmakefile ........................ Global makefile (Unix)
|__ src/ ............................... Directory containing source code of custom packages
|   |__ mypackage/ ...                   Directory containing source code of one particular package
|       |__ Package .................... Configuration file used to generate make / project files
|       |__ GNUmakefile ................ Unix makefile
|       |__ mypackage.vc80.vcproj ...... Visual Studio project file
|       |__ mypackageAPI.h ............. Windows storage-class specification
|       |__ MyModule.h ................. Header file of a custom module
|       |__ MyModule.cpp ............... Implementation of a custom module
|       |__ MyReader.cpp ............... Implementation of a custom read routine
|       |__ share/
|           |__ resource/
|               |__ mypackage.rc ....... Package resource file
|__ product/
    |__ obj/
    |   |__ arch-*-Debug/ .............. Object files on Unix (debug version)
    |   |__ arch-*-Optimize/ ........... Object files on Unix (optimized)
    |__ lib/
    |   |__ arch-*-Debug/ .............. Compiled debug shared objects (Unix)
    |   |__ arch-*-Optimize/ ........... Compiled optimized shared objects (Unix)
    |__ bin/
    |   |__ arch-Win*-Debug/ ........... Compiled debug binaries (Windows)
    |   |__ arch-Win*-Optimize/ ........ Compiled optimized binaries (Windows)
    |__ share/
        |__ resources/ ................. Package resource files are copied here
```

1.4 Quick Start Tutorial

This section contains a short tutorial on how to compile and execute the demo package provided with Developer Option. The demo package contains the source code of the example modules and IO routines described elsewhere in this manual. At this point you should just get a rough idea about the basic process required to develop your own modules and IO routines. Details will be discussed in the following chapters.
For the development of custom Amira packages a dedicated directory, the local

Amira directory, is required. Initially, this directory should be created using the *Development Wizard*. Lets see how this is done:

- Start Amira and choose the Development Wizard from the *Help* menu of the Amira main window.
- Make sure that the item *Set local Amira directory* is selected in the wizard's dialog window.
- Press the *Next* button.

You can now enter a directory name for the local Amira directory. For example you may choose `AmiraLocal` in your home directory. The directory must be different from directory where Amira has been installed.

- Enter the name for the local Amira directory.
- Select the button *copy demo package*.
- Press the *OK* button.

If the directory did not yet exist, Amira asks you if it should be created. The name of the directory is stored in the Windows registry or in the `.AmiraRegistry` file in the Unix or MacOS home directory, so that the next time Amira is started all modules or IO routines defined in this directory will be available.
The next step is to create the Visual Studio project files for Windows or the GNU-makefiles for Unix and MacOS. These files will be generated from a *Package* file which must be present in each custom package directory. The syntax of the Package file is described in Chapter 2.10. The demo package already contains a Package file, so there is no need to create one here.

- Select *Create build system* on the main page of the Development Wizard.
- Press the *Next* button.
- Choose *all local packages* as target.
- Choose which kind of build system you want to create.
- Press the *OK* button.

The files for the selected build system will now be created automatically. The advantage of the automatic generation is that the include and library paths are always set correctly. Also, any dependencies between local packages are taken into account.
Once the build system has been created you can close the Development Wizard and exit Amira. We are now ready to compile the demo package. This is different

for each platform:
Windows Visual Studio

- Start Visual Studio and load the solution file `AmiraLocal.vc80.sln / AmiraLocal.vc90.sln` from the local Amira directory. If your local Amira directory is not called `AmiraLocal`, the solution file also has some other name.
- Build all local packages in debug mode by pressing F7 or by choosing *Build Solution* from the *Build* menu.

Unix GNUmakefile system

- Change into the local Amira directory in a shell.
- Type in `gmake` to build all local packages in debug mode. If `gmake` is not already installed on your system you can find it in the subdirectory `bin` in the Amira root directory. Either add this directory in your path variable or create a link in a directory already listed in your path, e.g., `/usr/bin`.

Mac GNUmakefile system

- Open a `Terminal` window and change into the local Amira directory you created before.
- Type in `gmake` to build all local packages.

We are now ready to start Amira in order to test the demo package. However, because we have compiled the demo package in debug mode, we also need to start Amira in debug mode. Otherwise, Amira would not find the correct DLLs or shared libraries. For Linux, start Amira with the command line option `-debug`. On Windows use the *Amira (debug)* shortcut in the start menu. on Mac systems you can even start immediately the application, the new path will be automatically recognized.
In order to check if the demo package has been successfully compiled and can be loaded by Amira, you can for example choose the entry *DynamicColormap* from the *Create/Data* menu of the Amira main window. Then a new colormap object should be created. You can find the source code of this new object in the local Amira directory under `src/mypackage/MyDynamicColormap.cpp`. In the same directory there is also the header file for this class.
If you want to compile the demo package in release mode, you must change the active configuration in Visual Studio and recompile the code. On Unix, you have to call `gmake MAKE_CFG=Optimize`. You can also define `MAKE_CFG` as an

environment variable.

1.5 Compiling and Debugging

This section provides additional information not covered by the *quick start guide* on how to compile and debug custom Amira packages. You may skip it the first time you are reading this manual. The information will not become relevant until you are actually developing your own code.
It has already been mentioned that the development of custom Amira packages should take place in a local Amira directory. Initially, such a directory should be created using the Development Wizard described in Chapter 2. The name of the local Amira directory is stored in the Windows registry or in the file `.AmiraRegistry` in your Unix home directory. On both Windows and Unix, the name of the local Amira directory can be overridden by defining the environment variable `AMIRA_LOCAL`. This might be useful if you want to switch between different local Amira directories. However, in general it is recommended not to set this variable.
For each local package there is a resource file stored in the subdirectory `share/resources` in the local Amira directory. This file contains information about all modules and IO routines provided by that package. A local package can be compiled in debug mode suitable for debugging or in release mode with compiler optimization turned on. In the first case the DLLs or shared libraries are stored under `bin/arch-*-Debug` on Windows and `lib/arch-*-Debug` on Unix. In the second case they are stored under `bin/arch-*-Optimize` or `lib/arch-*-Optimize`. Here the '*' indicates the actual architecture name. In the following it will be described how to compile local packages in both modes on the different platforms and how to debug the code using a debugger.

1.5.1 Windows Visual Studio 2005/2008

Note: Mixing code generated with different versions of Visual Studio (Visual Studio 2005, 2008, etc.) is not officially supported. So, generally you need the same Visual Studio version Amira was compiled with (see *system requirements*). However, compiling your own modules using another version of Visual Studio *may* work if you install the corresponding Visual Studio runtime together with Amira on any Windows PC that make use of your custom modules.

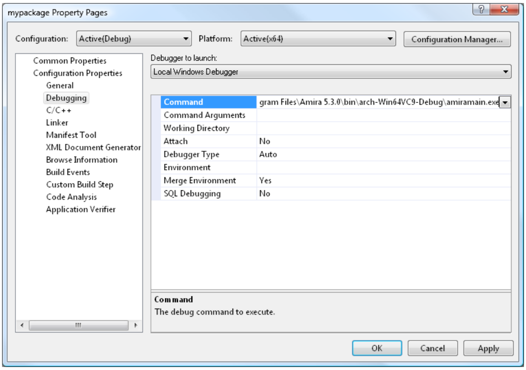

Figure 1.1: Specifying the name of the executable in Visual Studio.

The workspace and project files for Visual Studio are generated automatically from the Amira Package files by the Development Wizard. There should be no need to change the project settings manually.
By default, Visual Studio will compile in debug mode. In order to generate optimized code, you need to change the active configuration. This is done by choosing *Configuration Manager...* from the *Build* menu. In the *Active Solution Configuration* pulldown menu, select *Release*.
In order to execute the debug mode version of your local packages, you must start Amira with the debug Amiramain.exe located in the bin/arch-*-Debug folder. For convenience, a link *Amira (Debug)* is provided in the start menu. However, if you want to debug your code, you need to start Amira from Visual Studio. Thus you need to specify the correct executable in the project properties dialog.
You can bring up the project properties dialog by pressing Alt-F7 or by choosing *Properties* from the *Project* menu. Select *Debugging* from the left pane. In the field *Command*, choose the file bin/arch-*<version>*-Debug/Amiramain.exe located in the Amira root directory (see Figure 1.1). Replace *<version>* with the version of Amira you have (see *system requirements*).
Note: In previous Developer Option releases the -debug switch was needed to use the debug libraries of developed package. Since a while we ship debug libraries with Developer Option and provide debug binaries.
You can now start Amira from Visual Studio by pressing F5 or by choosing *Start*

from the *Debug* menu. In order to debug your code you may set breakpoints at arbitrary locations in your code. For example, if you want to debug a read routine, set a breakpoint at the beginning of the routine, execute Amira and invoke the read routine by loading some file.

1.5.2 Linux

In order to compile a local package under Linux you need to change into the package directory and execute gmake in a shell. The gmake utility is provided in the bin subdirectory of the Amira root directory. Either add this directory to your path or create a link in a directory already listed in your path, e.g., in /usr/bin.
The required GNUmakefiles will be generated automatically from the Amira Package files by the Development Wizard. There should be no need to edit the GNUmakefiles manually. Depending on the contents of the Package file all source files in a package directory will be compiled, or a subset only. By default all files will be compiled. The Development Wizard will put the name of the Amira root directory into the file GNUmakefile.setroot. You may overwrite the name by defining the environment variable AMIRA_ROOT. For example, this might be useful when working simultaneously with two different Amira versions.
By default gmake will compile debug code. In order to compile optimized code, invoke gmake MAKE_CFG=Optimize. Alternatively, you may set the environment variable MAKE_CFG to Optimize.
If you have a multi-processor machine you may compile multiple files at once by invoking gmake with the option PARALLELFLAGS=-j<n>. Here <n> denotes the number of compile jobs to be run in parallel. Usually twice the number of processors is a good choice.
If you have compiled debug code, you must invoke Amira with the command line option -debug. Otherwise, the optimized version will be executed. If no such version exists an error occurs. Instead of specifying -debug at the command line you may also set the environment variable MAKE_CFG to Debug.
In order to run Amira in a debugger, invoke the Amira start script with the -gdb or -ddd command line option.
Note that usually you cannot set a breakpoint in a local package right after starting the debugger. This is because the package's shared object file will not be linked to Amira until the first module or read or write routine defined in the package is invoked. In order to force the shared object to be loaded without invoking any module or read or write routine, you may use the command dso open lib<name>.so, where <name> denotes the name of the local

package. Once the shared object has been successfully loaded, breakpoints may be set. It depends on the debugger whether these breakpoints are still active when the program is started the next time.

1.6 Maintaining Existing Code

This section is directed to programmers who have already developed custom modules using a previous version of Developer Option. In particular, we describe

- *how to upgrade to Developer Option 5*, and
- *how to rename an existing package.*

1.6.1 Upgrading to Developer Option 5

In Developer Option 5 the structure of the local Amira directory has been slightly changed. In order to recompile existing packages it is recommended to create a complete copy of the local Amira directory and to adapt this copy as described below. Developer Option 5 and earlier versions store the path of the local Amira directory under different names in the Windows registry or in the Unix `.AmiraRegistry` file. This means that both versions can be used in parallel. The following changes are required to adjust an existing local Amira directory so that it can be used with Developer Option 5:

- The subdirectory *packages* must be renamed to *src* **(if upgraded from Amira versions former to 3.1)**.
- In each package directory a *Package* file must be created. This file contains the package name, the name of dependent packages, and optionally an explicit list of all source and header files belonging to the package. From the *Package* file a Visual Studio project file or a GNUmakefile for Unix can be generated automatically **(if upgraded from Amira versions former to 3.1)**.
- Instead of modifying and reusing the old Visual Studio project files or GNUmakefiles these files should be created from scratch using the Amira development wizard.

Modules, data classes and IO routines developed for Amira Developer Pack 3.1 should compile without changes with Developer Option 5 (with the possible exception of calls to the C++ standard library, see below). The API of existing classes has been extended in several cases, but no incompatible changes have been introduced. For details about the interface of particular classes please refer to the online reference guide. In addition, the following things have changed:

- *C++ standard library* **(if upgraded from Amira versions former to 3.1)**: On all platforms new-style C++ standard libraries are being used now. In Amira Developer Pack 3.0, the new-style interface was used only on Windows.

  In contrast to the old-style headers the new-style headers do not contain the suffix `.h`, e.g., you need to include `iostream` instead of `iostream.h`. In addition, all symbols are defined inside the `std` namespace, i.e., you need to write for example `std::cout` instead of just `cout`, unless you specifically write `using namespace std;` in your code.

  On most platforms the new-style and the old-style C++ standard library cannot be used together. This means that you may need to switch to the new-style interface, if you have used the old-style interface before.

- *Open Inventor and Qt:* Amira uses Open Inventor 8.1 and Qt 4. The Open Inventor headers are partly, the Qt headers are completley provided with Developer Option. Moreover **(if upgraded from Amira versions former to 3.1)**, instead of the SoWin and SoXt classes (which are not fully platform independent) in Amira now the new SoQt interface is used. This means that the Amira viewer is now derived from `SoQtExaminerViewer` instead of `SoWinExaminerViewer` or `SoXtExaminerViewer`, respectively. However, because we didn't want the class `HxViewer` to depend on any Qt headers, the actual Amira viewer has been renamed to `QxViewer`, while `HxViewer` now is a pure wrapper class.
  This change should not affect existing code unless platform-dependent methods of the former SoWin or SoXt base classes were used.

1.6.2 Renaming an Existing Package

Sometimes you may want to rename an existing Amira package, for example when using an existing package as a template for a new custom package. In order to do so the following changes are required:

- Rename the package directory:
  `AmiraLocal/src/oldname` becomes
  `AmiraLocal/src/newname`

- Rename the following files in the package directory:
  `oldnameAPI.h` becomes `newnameAPI.h`
  `share/resources/oldname.rc` becomes `share/resources/newname.rc`
- In the package resource file `share/resources/newname.rc` and in the `Package` file replace `oldname` by `newname`.
- In the file `newnameAPI.h` replace `OLDNAME_API` by `NEWNAME_API`.
- In all header and source files of the package, adjust the include directives if necessary, i.e., instead of
  `#include <oldname/SomeClass.h>`
  now write `#include <newname/SomeClass.h>`

All replacements can be performed using an arbitrary text editor. After all files have been modified as necessary a new Visual Studio project file or a new GNU-makefile should be created using the Amira development wizard.

2 The Development Wizard

The development wizard is a special tool which helps you to set up a local Amira directory tree so that you can write custom extensions for Amira. In addition, the development wizard can be used to create templates of Amira modules or of read or write routines. The details of developing such components are treated in other chapters. At this point we want to give a short overview about the functionality of the development wizard.
In particular, we discuss

- *how to invoke the development wizard*
- *how to set or create the local Amira directory*
- *how to add a package to the local Amira directory*
- *how to add components to an existing package*
- *how to create the files for the build system*

Finally, a section describing the *Package file syntax* is provided.

2.1 Starting the Development Wizard

In order to invoke the development wizard, first start Amira. Then select *Development Wizard* from the main window's *Help* menu. Note that this menu option will only be available if you are running the developer version of Amira (with Developer Option installed).
The layout of the development wizard is shown in Figure 2.1. Initially, the wizard informs you about the local Amira directory currently being used. If no local Amira directory is defined, this is indicated too. Furthermore, the wizard lets you select between four different tasks to be performed. These are

- *setting the local Amira directory (or creating a new one)*
- *adding a new package to the local Amira directory*
- *adding a component to an existing package*

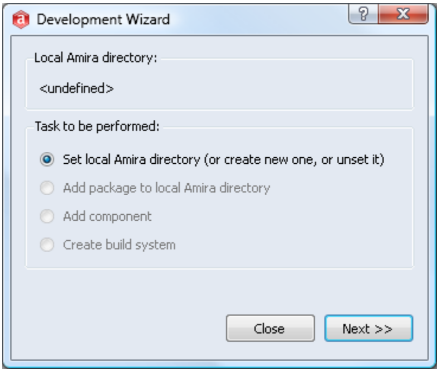

Figure 2.1: Initial layout of the Amira development wizard.

- *creating the files for the build system*

The first option is always available. A new package can only be added if a valid local Amira directory has been specified. For the local Amira directory to be valid, among others, it must contain a subdirectory called `src`. If at least one package exists in the `src` directory of the local Amira directory, a new component, i.e., a module or a read or write routine, can be added to a package. Finally, the last option allows you to create all files required by the build system, i.e., Visual Studio project files or GNUmakefiles for Unix platforms.

2.2 Setting Up the Local Amira Directory

The local Amira directory contains the source code and the binaries of all custom extensions developed by a user. The name of this directory can be most easily specified using the development wizard (see Figure 2.2). Since potentially every user can write his/her own extensions for Amira it is usually recommended that the local Amira directory is created in the user's home directory.
If the specified directory does not exist the development wizard asks you whether it should be created. If you confirm, the directory itself together with some subdirectories will be created. You may also specify an existing empty directory in the text field. Then the subdirectories will be created in there.

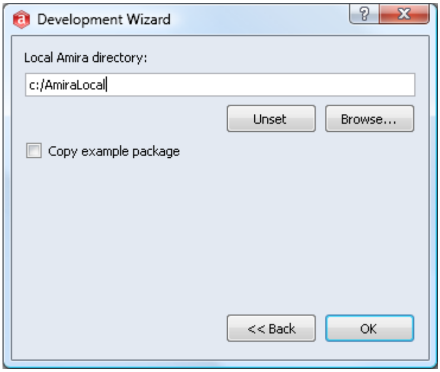

Figure 2.2: Setting the local Amira directory.

Finally, you may choose an existing directory which has been created by the development wizard before. In this case a simple check is performed to determine whether the specified directory is valid. If you want to use a directory created with Amira 3.0 or an earlier version, please first refer to Section 1.6 (upgrading existing code).

In order to unset the local Amira directory you should clear the text field and press *OK*. The directory will not be deleted, but the next time Amira is started modules and IO routines defined in the local Amira directory will not be available anymore.

Once you have set up the local Amira directory, the name of the directory is stored permanently, so the next time Amira is started the .rc-files located in the subdirectory `share/resources` of the local Amira directory can be read. In this way custom components are made known to Amira. On Windows the name of the local Amira directory is stored in the Windows registry. On Unix systems it is stored in the file `.AmiraRegistry` in the user's home directory. In both cases, these setting can be overridden by defining the environment variable `AMIRA_LOCAL`.

The development wizard also provides a toggle for copying a demo package to the local Amira directory. You will get a warning if this button is activated and an existing local Amira directory already containing the demo package has been specified. The demo package is copied to the subdirectory `src/mypackage` in the local Amira directory. It contains all read and write routines and modules presented as examples in this guide.

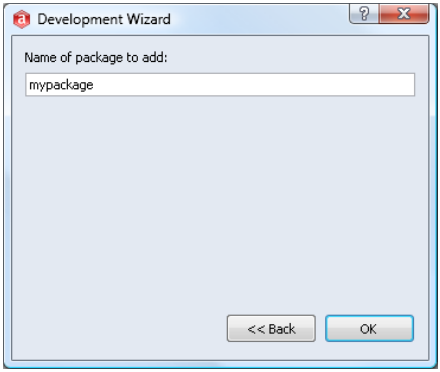

Figure 2.3: Adding a new package to the local Amira directory.

Please note: On Unix systems shared objects in the local Amira directory cannot use other shared objects in this directory. To circumvent this limitation add the lib directory (`amiralocal/lib/arch-LinuxAMD64-*`) to the `LD_LIBRARY_PATH` environment variable.

2.3 Adding a New Package

All Amira components are organized in *packages*. Each package will be compiled into a separate shared object (or DLL file on Windows). Therefore, before any components can be defined at least one package must be created in the local Amira directory. In order to do so, choose *add package to local Amira directory* on the first page of the wizard and press *Next*. On the next page you can enter the name of the new package (see Figure 2.3).

The name of a package must not contain any white spaces or punctuation characters. When a package is added, a subdirectory of the same name is created under `src` in the local Amira directory. In this directory the source code and header files of all the modules and IO routines of the package are stored. In addition in each package directory there must be a *Package* file from which the build system files can be generated.

Initially, a default *Package* file will be copied into a new package directory. This default file adds the most common Amira libraries for linking. It also selects all

C++ source files in the package directory to be compiled. In order to generate the build system from the *Package* file, please refer to Section 2.9.
In addition to the *Package* file also the files `version.cpp` and `packageAPI.h` will be copied into a new package directory. The first file allows you to put version information into your package, which can later be viewed in the Amira system information dialog. The second file contains a macro required for putting symbols in a DLL on Windows. Finally, also an empty file `package.rc` will be copied into `share/resources`. In this file later modules and IO routines will be registered.

2.4 Adding a New Component

If you choose the *add component* option on the first page of the development wizard, you will be asked what kind of component should be added to which package. Remember that the *add component* option will only be available if a valid local Amira directory with at least one existing package is found. In particular, templates

- of an *ordinary module*,
- of a *compute module*,
- of a *read routine*,
- or of a *write routine*

may be created (see Figure 2.4). The option menu in the lower part of the dialog box lets you specify the package to which the component should be added. After you press the *Next* button, you will be asked to enter more specific information about the particular component you want to add. Up to this point no real operation has been performed, i.e., no files have been created or modified.

2.5 Adding an Ordinary Module

An ordinary module in Amira usually directly visualizes the data object it is attached to. For example, the Isosurface module, the Voltex module, and the SurfaceView module are of this type. Such modules, sometimes also called *display modules*, are represented by yellow icons in the Pool.
In order to create the template for an ordinary module using the development wizard, you must enter the C++ class name of the module, the name to be shown in the pop-up menu of possible input data objects, the C++ class name of possible input

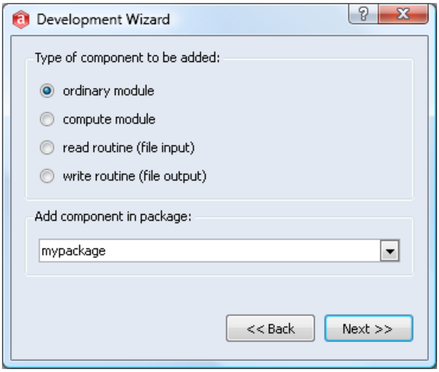

Figure 2.4: Adding a new component to an existing package.

data objects, and finally the package where the input class is defined (see Figure 2.5).
Once you press *OK*, two files are created in the package directory, namely a header file and a source code file for the new module. In addition, a new module statement is added to the package resource file located under `share/resources` in the package directory.
After you have added a new module to a package you need to recreate the build system files before you can compile the module. Details are described in Section 2.9.

2.6 Adding a Compute Module

A *compute module* in Amira usually takes one or more input data objects, performs some kind of computation, and puts back a resulting data object in the Pool. Compute modules are represented by red icons in the Pool.
The only difference between an ordinary module and a compute module is that the former is directly derived from `HxModule` while the latter is derived from `HxCompModule`. When creating a template for a compute module using the development wizard, the same input fields must be filled in as for an ordinary module. The meaning of these input fields is described in Section 2.5.

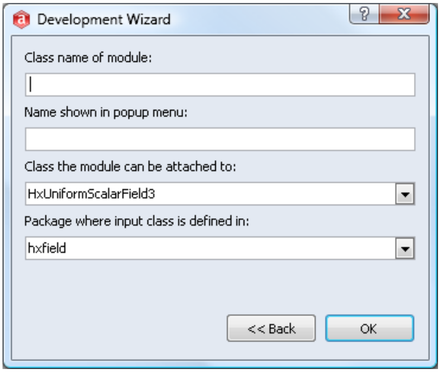

Figure 2.5: Creating the template of a custom module.

2.7 Adding a Read Routine

As will be explained in more detail in Section 3.2, read routines are global C++ functions used to create one or more Amira data objects from the contents of a file stored in a certain file format. To create the template of a new read routine, first the name of the routine must be specified (see Figure 2.6). The name must be a valid C++ name. It must not contain blanks or any other special characters.

Moreover, the name of the file format and the preferred file name extension must be specified. The extension will be used by the file browser in order to identify the file format. The format name will be displayed next to any matching file.

Finally, a toggle can be set in order to create the template of a read routine supporting the input of multiple files. Such a routine will have a slightly different signature. It allows you to create a single data object from multiple input files. For example, multiple 2D image files can be combined in a single 3D image volume. Details are provided in Section 3.2.3.

After you press *OK* a new file `<name>.cpp` will be created in the package directory, where `<name>` denotes the name of the read routine. In addition, the read routine will be registered in the package resource file. Some file formats can be identified by a unique file header, not just by the file name extension. In such a case you may want to modify the resource file entry as described in Section 3.2.1. Remember, that after you have added a new read routine to a package you need to recreate the build system files before you can compile it. Details are described in

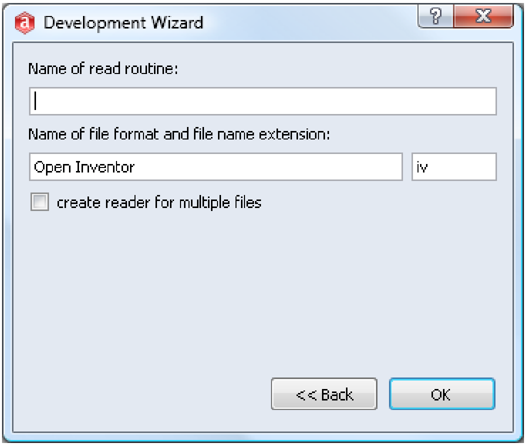

Figure 2.6: Creating the template of a read routine.

Section 2.9.

2.8 Adding a Write Routine

A write routine is a global C++ function which takes a pointer to some data object and writes the data to a file in a certain file format. The details are explained in Section 3.3. In order to create the template of a new write routine, the name of the routine must first be specified (see Figure 2.7). The name must be a valid C++ name. It must not contain blanks or any other special characters.
In addition, the name of the file format and the preferred file name extension must be specified. Before saving a data object, both the name and the extension will be displayed in the file format menu of the Amira file browser.
Finally, the C++ class name of the data object to be saved must be chosen as well as the package this class is defined in. Some important data objects such as a `HxUniformScalarField3` or a `HxSurface` are already listed in the corresponding combo box. However, any other class, including custom classes, may be specified here. Instead of the name of a data class, even the name of an interface class such as `HxLattice3` may be used (see Section 5.2.1).
After you press *OK*, a new file `<name>.cpp` will be created in the package direc-

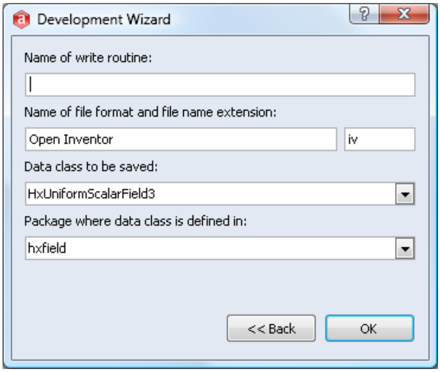

Figure 2.7: Creating the template of a write routine.

tory, where `<name>` denotes the name of the write routine. In addition, the write routine will be registered in the package resource file.
Remember, that after you have added a new write routine to a package you need to recreate the build system files before you can compile it. Details are described in Section 2.9.

2.9 Creating the Build System Files

Before you can actually compile your own packages you need to create project files for Visual Studio on Windows or GNUmakefiles for Unix. These files contain information such as the source code files to be compiled, or the correct include and library paths. Since it is not trivial to set up and edit these files manually, Amira provides a mechanism to create them automatically. In order to do this, a so-called *Package* file must exist in each package. The Package files contains the name of the package and a list of dependent packages. It may also contain additional tags to customize the build process. The syntax of the package file is described in Section 2.10. However, usually there is no need to modify the default Package file created by the development wizard.
While the automatic generation of the build system files is a very helpful feature it also means that you better do not modify the resulting project or GNUmakefiles manually, because they might be easily overwritten by Amira.

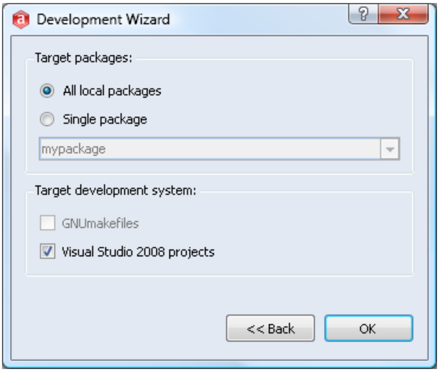

Figure 2.8: Creating the build system files.

If you select *Create build system* on the main page of the development wizard and then press the *Next* button, the controls shown in Figure 2.8 will be activated. You can choose if you want to create the build system files for all local packages or just for a particular one. Depending on the selected build system the following files will be created:

GNUmakefiles
`share/src/mypackage/GNUmakefile`: A GNUmakefile for building *mypackage*. If *all local packages* is selected such a file will be created in every subdirectory containing a *Package file*.
In order to compile all local packages at once, you can type *gmake* in the local Amira directory. In this directory there is a standard GNUmakefile which calls *gmake* in all package directories.

Visual Studio
`AmiraLocal.vc8.sln / AmiraLocal.vc90.sln`: A workspace containing projects for all local packages. This file will only be written if *all local packages* is selected in the development wizard.
`allAmiraLocal.vc80.vcproj / allAmiraLocal.vc90.vcproj`:
A project file which depends on all other projects. Choose this project as active project in Visual Studio if you want to compile all local packages.
`docAmiraLocal.vc80.vcproj / docAmiraLocal.vc90.vcproj`:
A project for creating the documentation for all local packages. This file will only

be written if *all local packages* is selected in the development wizard.
`share/src/mypackage/mypackage.vc80.vcproj` / `share/src/mypackage/mypackage.vc90.vcproj`: A project for building *mypackage*. If *all local packages* is selected such a file will be created in every subdirectory containing a *Package* file.
The syntax of the *Package* file is described in the *following section*.

2.10 The Package File Syntax

The Package file contains information about a local package. From this file Visual Studio project files or GNUmakefiles can be generated. The Package file is a Tcl file. It defines a set of Tcl variables indicating things like the name of the package, dependent libraries, or additional files to be copied when building the package. The default Package file created by the Development Wizard looks as follows:

```
set PACKAGE {mypackage}

set LIBS {
    hxplot hxtime hxsurface hxcolor hxfield
    hxcore amiramesh mclib oiv tcl
}

set SHARE {
    share/resources/mypackage.rc
}
```

In most cases the default file works well and need not to be modified. However, in order to accomplish special tasks the default values of the variables can be changed or additional variables can be defined. Here is a detailed list describing the meaning of the different variables:

PACKAGE

The variable `PACKAGE` indicates the name of the package. This should be the same as the name of the package directory. The package name must not contain any characters other than letters or digits.

LIBS

Lists all libraries the package depends on. By default, the most common Amira packages are inserted here. You can modify this list as needed. For example, if you want to link against a library called *foo.lib* on Windows or *libfoo.so* on Unix, you should add *foo* to `LIBS`.
In addition to a real library name you may use the following aliases in the `LIBS` variable:

`oiv` - for the Open Inventor libraries
`tcl` - for the Tcl library
`opengl` - for the OpenGL library
`qt` - for the Qt library

If you want to link against a library only on a particular platform, you can set a dedicated variable `LIBS-arch`, where `arch` denotes the platform. You may further distinguish between the debug and release version of the code. Here is an example:

```
set LIBS {mclib amiramesh schedule hxz qt oiv opengl tcl}
set LIBS-Unix {hxgfxinit}
set LIBS-Win32 {hxgfxinit}
set LIBS-Win32VC8-Debug {msvcrtd mpr}
set LIBS-Win32VC8-Optimize {msvcrt mpr}
```

SHARE

Lists all files which should be copied from the package directory into the local Amira directory. By default, only the package resource file will be copied. However, you may add additional files here if necessary. Instead of explicit file names you may use wildcards. These will be resolved using the standard Tcl command `glob`. For example, if you have some demo scripts in your package you could copy them in the following way:

```
set SHARE {
    share/resources/mypackage.rc
    share/demo/mydemos/*.hx
}
```

As for the `LIBS` variable you may append an arch string here, i.e., `SHARE-arch`. The files then will only be copied on the specified platforms.

INCLUDES

This variable may contain a list of additional include paths. These paths are used by the compiler to locate header files. By default, the include path is set to `$AMIRA_ROOT/include`, `$hxroot/include/oiv`, `$AMIRA_LOCAL/src`, and the local package directory.

COPY

This may contain a list of files which are copied from a location other than the local package directory. You need to specify the name of the target file followed by the name of the destination file relative to the local Amira directory. For example, you may want to copy certain data files from some archive into the Amira directory. This can be achieved in the following way.

```
set COPY {
    D:/depot/data/28523763.dcm data/test
    D:/depot/data/28578320.dcm data/test
    D:/depot/data/28590591.dcm data/test
}
```

As for the `LIBS` variable you may append an arch string here, i.e., `COPY-arch`. The files then will only be copied on the specified platforms. A common application is to copy external libraries required on a particular platform into the Amira directory.

SRC

This variable specifies the source code files to be compiled for the package. The default value of this variable is

```
set SRC {*.cpp *.c}
```

This means, that By default all .cpp and .c files in the local package directory will be compiled. Sometimes you may want to replace this default by an explicit list of source files.
Again, you may append an arch string to the `SRC` variable, so that certain files will only be compiled on a particular platform.

INCSRC

This variable specifies the header files to be included into the package project file. The default value of this variable is

```
set INCSRC {*.h *.hpp}
```

This means that by default all .h and .hpp files in the local package directory will be considered.
Again, you may append an arch string to the `INCSRC` variable, so that certain header files will only be considered on a particular platform.

3 File I/O

This chapter describes how user-defined read and write routines can be added to Amira. The purpose of custom read and write routines is to add support for file formats not available in Amira.
First, some general hints *on file formats* are given. Then we discuss how *read routines* are expected to look in Amira. *Write routines* are treated subsequently. Finally, the *AmiraMesh API* is discussed. Using this API, file I/O for new custom data objects can be implemented rather easily.

3.1 On file formats

Before going into detail, let us clarify some general concepts. In Amira, all data loaded into the system are encapsulated in C++ *data classes*. Chapter 5 provides more information about the standard data classes. For example, there is a class to represent tetrahedral grids (*HxTetraGrid*), a separate one for scalar fields defined on tetrahedral grids (*HxTetraScalarField*), and another one for 3D image data (*HxUniformScalarField3*). Every instance of a data class is represented by a green icon in the Amira Pool.
The way in which data are stored in a disk file is called a file format. Although there is a relationship between data classes and file formats, these are two different things. It is especially important to understand that there is no one-to-one correspondence between them.
Typically, a specific data class (like 3D image data) can be stored in many different file formats (3D TIFF, DICOM, a set of 2D JPEG files, and so on). On the other hand, a specific file format does not necessarily correspond to exactly one data class. For example, a data file in Fluent UNS format can either contain hexahedral grids (*HxHexaGrid*) or tetrahedral grids (*HxTetraGrid*).
Note that there is also no one-to-one correspondence between the instance of a data class (a green icon in Amira) and the instance of a file format (the actual file). Often multiple files correspond to a single data object, for example 2D images forming a single 3D image volume. On the other hand, a single file can contain

the data of multiple data objects. For example, an AVS UCD file can contain a tetrahedral grid as well as multiple scalar fields defined on it.
Finally, note that information may get lost when saving a data object to a file in a specific format. For example, when saving a 3D image volume to a set of 2D JPEG images, the bounding box information will be lost. Likewise, there are user-defined parameters or attributes in Amira that cannot be encoded in most standard file formats. On the other hand a file reader often does not interpret all information provided by a specific file format.

3.2 Read Routines

As already mentioned in the previous section, a read routine is a C++ function that reads a disk file, interprets the data, creates an instance of an Amira data class, and fills that instance with the data read from the file.
In order to write a read routine, obviously two things are needed, namely a specification of the file format to be read as well as the information which of Amira's data classes is able to represent the data and how this class is used. More information about the standard Amira data classes is given in Chapter 5. The C++ interface of these classes is described in the online reference documentation.
A read routine may either be a static member function of a class or a global function. In addition to the function itself, an entry in the package resource file is needed. In this way Amira is informed about the existence of the read routine and about the type of files that can be handled by the reader.
In the following discussion the implementation of a user-defined read routine will be illustrated by two concrete examples, namely *a simple read routine for scalar fields* and *a read routine for surfaces and surface fields*. Some *more details about read routines* will be discussed subsequently.

3.2.1 A Reader for Scalar Fields

In this section we present a simple read routine designed for reading image volumes, i.e., 3D scalar fields, from a very simple file format, which we have invented for this example. The file format is called PPM3D (because it is similar to the ppm 2D image format). The PPM3D format will be an ASCII file format containing a header, three integer numbers specifying the size of the 3D image volume, and the pixel data as integer numbers in the range 0 to 255. An example file could look like this:

```
# PPM3D
4 4 3
43 44 213 9 23 234 3 3 3 44 213 9 23 234 36 63
44 213 9 23 234 35 3 5 44 213 9 23 234 31 13 12
44 213 9 23 234 35 3 5 44 213 9 23 234 31 13 12
```

The full source code of the read routine is contained in the demo package provided with Developer Option. In order to follow the example below, first create a local Amira directory using the *Development Wizard*. Be sure that the toggle *copy demo package* is activated, as described in Section 2.2. The read routine can then be found in the local Amira directory under `packages/mypackage/readppm3d.cpp`.

Let us first take a look at the commented source code of the reader. Some general remarks follow below.

```
////////////////////////////////////////////////////////////////////
//
// Sample read routine for the PPM3D file format
//
////////////////////////////////////////////////////////////////////

#include <hxcore/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/mypackageAPI.h>

MYPACKAGE_API int readppm3d(const char* filename)
{
    FILE* f = fopen(filename, "r"); // open the file

    if (!f) {
        theMsg->ioError(filename);
        return 0; // indicate error
    }
```

```
    // Skip header (first line). We could do some checking here:
    char buf[80];
    fgets(buf, 80, f);

    // Read size of volume:
    int dims[3];
    dims[0] = dims[1] = dims[2] = 0;
    fscanf(f, "%d %d %d", &dims[0], &dims[1], &dims[2]);

    // Do some consistency checking.
    if (dims[0]*dims[1]*dims[2] <= 0) {
        theMsg->printf("Error in file %s.", filename);
        fclose(f);
        return 0;
    }

    // Now create 3D image data object. The constructor takes
    // the dimensions and the primary data type. In this case
    // we create a field containing unsigned bytes (8 bit).
    HxUniformScalarField3* field =
        new HxUniformScalarField3(dims, McPrimType::mc_uint8);

    // The HxUniformScalarField3 stores its data in a member
    // variable called lattice. We know that the data is unsigned
    // 8 bit because we specified this in the constructor.
    unsigned char* data =
        (unsigned char*) field->lattice.dataPtr();

    // Now we must read dims[0]*dims[1]*dims[2] data values
    for (int i=0; i<dims[0]*dims[1]*dims[2]; i++) {
        int val=0;
        fscanf(f,"%d",&val);
        data[i] = (unsigned char) val;
    }

    // We are done with reading, close the file.
    fclose(f);

    // Register the data object to make it visible in the
    // Pool. The name for the new object is automatically
    // generated from the filename.
    HxData::registerData(field, filename);

    return 1; // Indicate success
}
```

The source file starts with some includes. First, the file *HxMessage.h* is included.

This header file provides the global pointer *theMsg* which allows us to print out text messages in the Amira console window. In our read routine we use *theMsg* to print out error messages if a read error occurred.
Next, the header file containing the declaration of the data class to be created is included, i.e., *HxUniformScalarField3.h*. As a general rule, every class in Amira is declared in a separate header file. The name of the header file is identical to the name of the C++ class.
Finally, the file *mypackageAPI.h* is included. This file provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. On Unix systems this macro is empty and can be omitted.
The read routine itself takes one argument, the name of the data file to be read. It should return 1 on success, or 0 if an error occurred and no data object could be created. The body of the read routine is rather straightforward. The file is opened for reading. The size of the image volume is read. A new data object of type *HxUniformScalarField3* is created and the rest of the data is written into the data object. Finally, the file is closed again and the data object is put into the Pool by calling `HxData::registerData`. In principle, all read routines look like this example. Of course, the type of data object being created and the way that this object is initialized may differ.
In order to make the new read routine known to Amira, an entry must be added to the package resource file, i.e., to the file `mypackage/share/resources/mypackage.rc`. In our case this entry looks as follows:

```
dataFile -name "PPM3D Demo Format"  \
    -header "PPM3D"                 \
    -load "readppm3d"               \
    -package "mypackage"
```

The `dataFile` command registers a new file format called *PPM3D Demo Format*. The option `-header` specifies a regular expression which is used for automatic file format detection. If the first 64 bytes of a file match this expression, the file will be automatically loaded using this read routine. Of course, some data formats do not have a unique file header. In this case, the format may also be detected from a standard file name extension. Such an extension may be specified using the `-ext` option of the `dataFile` command. Multiple extensions can be specified as a comma-separated list. The actual C++ name of the read routine is specified via `-load`. Finally, the package containing the read routine must be specified using the `-package` option.

If you have compiled the example in the mypackage demo package, you can try to load the demo file `mypackage/data/test.ppm3d`. As you will see, the file browser automatically detects the file format and displays *PPM3D Demo Format* in its file list.

3.2.2 A Reader for Surfaces and Surface Fields

Now that you know what a read routine looks like in principle, let us consider a more complex example. In this section we discuss a read routine which creates more than one data object. In particular, we want to read a triangular surface mesh from a file. In addition to the surface description, the file may also contain data values for each vertex of the surface. Data defined on a surface mesh are represented by separate classes in Amira. Therefore, the reader must first create a data object representing the surface only. Then appropriate data objects must be created for each surface field.
Again, the file format is quite simple and has been invented for the purpose of this example. We call it the *Trimesh* format. It is a simple ASCII format without any header. The first line contains the number of points and the number of triangles. Then the x-, y-, and z-coordinates of the points are listed. This section is followed by triangle specifications consisting of three point indices for each triangle, point count starts at one. The next section is for vertex data, starting with a line that contains an arbitrary number of integers. Each integer indicates that there is a data field with a certain number of variables defined on the surface's vertices, e.g., 1 for a scalar field or 3 for a vector field. The data values for each vertex follow in separate lines. Here is a small example containing a single scalar surface field:

```
4 2
0.0 0.0 0.0
1.0 0.0 0.0
0.0 1.0 0.0
1.0 1.0 0.0
1 2 4
1 4 3
1
0.0
0.0
1.0
1.0
```

You can find the full source code of the reader in the local Amira directory under `packages/mypackage/readtrimesh.cpp`. Remember that the demo package must have been copied into the local Amira directory before compiling. For details, refer to Section 2.2. Let us now look at the complete read routine before discussing the details:

```
////////////////////////////////////////////////////////////////////
//
// Read routine for the Trimesh file format
//
////////////////////////////////////////////////////////////////////

#include <McStringTokenizer.h>
#include <hxcore/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <hxsurface/HxSurfaceField.h>
#include <mypackage/mypackageAPI.h>

MYPACKAGE_API int readtrimesh(const char* filename)
{
    FILE* fp = fopen(filename, "r");

    if (!fp) {
        theMsg->ioError(filename);
        return 0;
    }

    // 1. Read the surface itself

    char buffer[256];
    fgets(buffer,256,fp); // read first line

    int i, j, k, nPoints=0, nTriangles=0;
    // Get number of points and triangles
    sscanf(buffer, "%d %d", &nPoints, &nTriangles);

    if (nPoints<0 || nTriangles<0) {
        theMsg->printf("Illegal number of points or triangles.");
        fclose(fp);
        return 0;
    }

    HxSurface* surface = new HxSurface; // create new surface
    surface->addMaterial("Inside",0); // add some materials
    surface->addMaterial("Outside",1);

    HxSurface::Patch* patch = new HxSurface::Patch;
    surface->patches.append(patch); // add patch to surface
    patch->innerRegion = 0;
    patch->outerRegion = 1;

    surface->points.resize(nPoints);
    surface->triangles.resize(nTriangles);
```

```
for (i=0; i<nPoints; i++) { // read point coordinates
    McVec3f& p = surface->points[i];
    fgets(buffer,256,fp);
    sscanf(buffer, "%g %g %g", &p[0], &p[1], &p[2]);
}

for (i=0; i<nTriangles; i++) { // read triangles
    int idx[3];
    fgets(buffer,256,fp);
    sscanf(buffer, "%d %d %d", &idx[0], &idx[1], &idx[2]);

    Surface::Triangle& tri = surface->triangles[i];
    tri.points[0] = idx[0]-1; // indices should start at zero
    tri.points[1] = idx[1]-1;
    tri.points[2] = idx[2]-1;
    tri.patch = 0;
}

// Add all triangles to the patch
patch->triangles.resize(nTriangles);
for (i=0; i<nTriangles; i++)
    patch->triangles[i] = i;

// Add surface to Pool
HxData::registerData(surface,filename);

// 2. Check if file also contains data fields

fgets(buffer,256,fp);
McStringTokenizer tk(buffer);
McDArray<HxSurfaceField*> fields;

while (tk.hasMoreTokens()) { // are there any numbers here ?
    int n = atoi(tk.nextToken());
    // Create field with desired number of components
    HxSurfaceField* field = HxSurfaceField::create(surface,
        HxSurfaceField::OnNodes, n);
    fields.append( field );
}

if (fields.size()) {
    // Read data values for all fields
    for (i=0; i<nPoints; i++) {
        fgets(buffer,256,fp);
        tk = buffer;
        for (j=0; j<fields.size(); j++) {
```

```
                int n = fields[j]->nDataVar();
                float* v = &fields[j]->dataPtr()[i*n];
                for (k=0; k<n; k++)
                    v[k] = atof(tk.nextToken());
            }
        }

        // Add all fields to Pool
        for (i=0; i<fields.size(); i++) {
            HxData::registerData(fields[i], NULL);
            fields[i]->composeLabel(surface->getName(),"data");
        }
    }

    fclose(fp); // close file and return ok
    return 1;
}
```

The first part of the read routine is very similar to the PPM3D reader outlined in the previous section. Required header files are included, the file is opened, the number of points and triangles are read, and a consistency check is performed.

Then an Amira surface object of type *HxSurface* is created. The class *HxSurface* has been devised to represent an arbitrary set of triangles. The triangles are organized into *patches*. A patch can be thought of as the boundary between two volumetric regions, an "inner" and an "outer" region. Therefore, for each patch an inner region and an outer region should be defined. In our case, all triangles will be inserted into a single patch. After this patch has been created and initialized, the number of points and triangles is set, i.e., the dynamic arrays *points* and *triangles* are resized appropriately.

Next, the point coordinates and the triangles are read. Each triangle is defined by the three points it consists of. The point indices start at one in the file but should start at zero in the *HxSurface* class. Therefore all indices are decremented by one. Once all triangles have been read, they are inserted into the patch we have created before. The surface is now fully initialized and can be added to the Pool by calling *HxData::registerData*.

The second part of the read routine is reading the data values. First, we check how many data fields are defined and how many data variables each field has. In order to parse this information, we use the utility class *McStringTokenizer*. This class returns blank-separated parts of a string one after the other. For more information about this and other utility classes refer to the online reference documentation of Developer Option.

For each group of data variables, a corresponding surface field is created. The

fields are temporarily stored in the dynamic array *fields*. Instead of directly calling the constructor of the class *HxSurfaceField*, the static method *HxSurfaceField::create* is used. This method checks the number of data variables and automatically creates an instance of a subclass such as *HxSurfaceScalarField* or *HxSurfaceVectorField*, if this is possible. In principle, surface fields may store data on a per-node or a per-triangle basis. Here we are dealing with vertex data, so we specify the encoding to be *HxSurfaceField::OnNodes* in *HxSurfaceField::create*.
Finally, the data values are read into the surface fields created before. Afterwards, all the fields are added to the Pool by calling *HxData::registerData* again. In order to define a useful name for the surface fields, we call the method *composeLabel*. This method takes a reference name, in this case the name of the surface, and replaces the suffix by some other string, in this case "data". Amira automatically modifies the name so that it is unique. Therefore we can perform the same replacement for all surface fields.
Like any other read routine, our *Trimesh* reader must be registered in the package resource file before it can be used. This is done by the following statement in `mypackage/share/resources/mypackage.rc`:

```
dataFile -name "Trimesh Demo Format"    \
    -ext "trimesh,tm"                   \
    -load "readtrimesh"                 \
    -package "mypackage"
```

Most of the options of the `dataFile` command have already been explained in the previous section. However, in contrast to the PPM3D format, the *Trimesh* format cannot be identified by its file header. Therefore, we use the `-ext` option to tell Amira that all files with file name extensions *trimesh* or *tm* should be opened using the *Trimesh* reader.

3.2.3 More About Read Routines

The basic structure of a read routine should be clear from the examples presented in the previous two sections. Nevertheless, there are a few more things that might be of interest in some situations. These will be discussed in the following.

Reading Multiple Images At Once

The Amira file browser allows you to select multiple files at once. Usually, all these files are opened one after the other by first determining the file format and then calling the appropriate read routine. However, in some cases the data of a single Amira data object are distributed among multiple files. The most prominent example is 3D images where every slice is stored in a separate 2D image file. In order to be able to create a full 3D image, the file names of all the individual 2D images must be available to a read routine. To facilitate this, read routines in Amira can have two different signatures. Besides the ordinary form

```
int myreader(const char* filename);
```

read routines can also be defined as follows:

```
int myreader(int n, const char** filenames);
```

In both cases exactly the same `dataFile` command can be used in the package resource file. Amira automatically detects whether a read routine takes a single file name as an argument or multiple ones. In the latter case, the read routine is called with the names of all files selected in the file browser, provided all these files have the same file format (if multiple files with different formats are selected, the read routine for each format is called with the matching files only). You can create the template of a multiple files read routine by selecting the toggle *create reader for multiple files* in the Development Wizard (see Section 2.7).

The Load Command

The current state of the Amira network with all its data objects and modules can be stored in a script file. When executed, the script should restore the network again. Of course, this is a difficult task especially if data objects have been modified since they have been loaded from files. However, even if this is not the case, Amira must know how to reload the data later on.
For this purpose a special parameter called *LoadCmd* should be defined for the data object. This parameter should contain a Tcl command sequence which restores the data object on execution. Usually, the load command is simply set to `load <filename>` when calling `HxData::registerData`. However, this approach fails if the format of the file cannot be detected automatically, or if multiple data objects are created from a single file, e.g., as in our *Trimesh* example.
In such cases the load command should be set manually. In case of the *Trimesh* reader, this could be done by adding the following lines of code at the very end of the routine just before the method's returning point:

```
McString loadCmd;
loadCmd.printf("set TMPIO [load -trimesh %s]\n"
               "lindex $TMPIO 0", filename);
surface->setLoadCmd(loadCmd,1);

for (int i=0; i<fields.size(); i++) {
    loadCmd.printf("lindex $TMPIO %d", i+1);
    fields[i]->setLoadCmd(loadCmd,1);
}
```

This code requires some explanation. The file is loaded and all data objects are created when the first line of the load command is executed. Note that we specified the `-trimesh` option as an argument of `load`. This ensures that the *Trimesh* reader will always be used. The format of the file to be loaded will not be determined automatically. The Tcl command `load` returns a list with the names of all data objects which have been created. This list is stored in the variable TMPIO. Later the names of the individual objects can be obtained by extracting the corresponding elements from this list. This is done using the Tcl command `lindex`.

Using Dialog Boxes in a Read Routine

In some cases a file cannot be read successfully unless certain parameters are interactively specified by the user. Usually this means that a special-purpose dialog must be popped up within the read routine. This is done, for example, when raw data are read in Amira. In order to write your own dialogs, you must use Qt, a platform-independent toolkit for designing graphical user interfaces.
If you don't want to use Qt, you may consider implementing your read routine within an ordinary module. Although this somewhat breaks Amira's data import concept, it will work too, of course. You then can utilize ordinary ports to let the user specify required import parameters.

3.3 Write Routines

Like read routines, write routines in Amira are C++ functions, either global ones or static member functions of an arbitrary class. In the following discussion we present write routines for the same two formats for which reader codes have been explained in the previous section. First, *a writer for scalar fields* will be discussed, then *a writer for surfaces and surface fields.*

3.3.1 A Writer for Scalar Fields

In this section we explain how to implement a routine for writing 3D images, i.e., instances of the class *HxUniformScalarField3*, to a file using the PPM3D format introduced in Section 3.2.1. The writer is even simpler than the reader. Again, the source code is contained in the demo package of Developer Option. Once you have created a local Amira directory using the *Development Wizard* and copied the demo package into that directory, you will find the write routine in the local Amira directory under `packages/mypackage/writeppm3d.cpp`. Here it is:

```
/////////////////////////////////////////////////////////////////
//
// Sample write routine for the PPM3D file format
//
/////////////////////////////////////////////////////////////////

#include <hxcore/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/mypackageAPI.h>

MYPACKAGE_API
int writeppm3d(HxUniformScalarField3* field, const char* filename)
{
    // For the moment we only want to support byte data
    if (field->primType() != McPrimType::mc_uint8) {
        theMsg->printf("This format only supports byte data.");
        return 0; // indicate error
    }

    FILE* f = fopen(filename, "w"); // open the file

    if (!f) {
        theMsg->ioError(filename);
        return 0; // indicate error
    }

    // Write header:
    fprintf(f, "# PPM3D\n");

    // Write fields dimensions:
    const int* dims = field->lattice.dims();
    fprintf(f, "%d %d %d\n", dims[0], dims[1], dims[2]);

    // Write dims[0]*dims[1]*dims[2] data values:
    unsigned char* data =
        (unsigned char*) field->lattice.dataPtr();
```

```
    for (int i=0; i<dims[0]*dims[1]*dims[2]; i++) {
        fprintf(f, "%d ", data[i]);
        if (i%20 == 19) // do some formatting
            fprintf(f,"\n");
    }

    // Close the file.
    fclose(f);

    return 1; // indicate success
}
```

At the beginning, the same header files are included as in the reader. *HxMessage.h* provides the global pointer *theMsg* which allows us to print out text messages in the Amira console window. *HxUniformScalarField3.h* contains the declaration of the data class to be written to the file. Finally, *mypackageAPI.h* provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. On Unix systems, this macro is empty and can be omitted.

The signature of a write routine differs from that of a read routine. It takes two arguments, namely a pointer to the data object to be written to a file, as well as the name of the file. Before a write routine is called, Amira always checks if the specified file already exists. If this is the case, the user is asked if the existing file should be overwritten. Therefore, such a check need not to be coded again in each write routine. Like a read routine, a write routine should return 1 on success, or 0 if an error occurred and the data object could not be saved.

The body of the write routine is almost self-explanatory. At the beginning, a check is made whether the 3D image really consists of byte data. In general, the type of data values of such an image can be 8-bit bytes, 16-bit shorts, 32-bit integers, floats, or doubles. If the image does contain bytes, a file is opened and the image contents are written into it. However, note that the data object also contains information which cannot be stored using our simple PPM3D file format. First of all, this applies to the bounding box of the image volume, i.e., the position of the center of the first and the last voxel in world coordinates. Also, all parameters of the object (defined in the member variable *parameters* of type *HxParamBundle*) will be lost if the image is written into a PPM3D file and read again.

Like a read routine, a write routine must be registered in the package resource file, i.e., in `mypackage/share/resources/mypackage.rc`. This is done by the following statement:

```
dataFile -name "PPM3D Demo Format"    \
    -save "writeppm3d"                \
    -type "HxUniformScalarField3"     \
    -package "mypackage"
```

The option `-save` specifies the name of the write routine. The option `-type` specifies the C++ class name of the data objects which can be saved using this format. Note that an export format may be registered for multiple C++ objects of different type. In this case multiple `-type` options should be specified. However, for each type there must be a separate write routine with a different signature (polymorphism). For example, if we additionally want to register the PPM3D format for objects of type *HxStackedScalarField3*, we must additionally implement the following routine:

```
int writeppm3d(HxStackedScalarField3* field, const char* fname);
```

Besides the standard data classes, there are so-called *interface classes* that may be specified with the `-type` option. For example, in this way it is possible to implement a generic writer for n-component regular 3D fields. Such data is encapsulated by the interface *HxLattice3*. For more information about interfaces, refer to Chapter 5.
At this point you may try to compile and execute the write routine by following the instructions given in Section 1.5 (Compiling and Debugging).

3.3.2 A Writer for Surfaces and Surface Fields

For the sake of completeness, a writer for the *Trimesh* format introduced in Section 3.2.2 is described in this section. Remember that the *Trimesh* format is suitable for storing a triangular mesh as well as an arbitrary number of data values defined on the vertices of the surface. In Amira, surfaces and data fields defined on surfaces are represented by different objects. This also has some implications when designing a write routine.
In our example we actually implement two different write routines, one for the surface and one for the surface field. If the user selects the surface and exports it using the `Trimesh` writer, the surface mesh as well as all attached data fields will be written to file. On the other hand, if the user selects a particular surface field, the corresponding surface and just the selected field will be written.
The source code of the writer can be found in the local Amira directory under `packages/mypackage/writetrimesh.cpp`. Remember that the demo

package must be copied into the local Amira directory before compiling. For details refer to Section 2.2. Again, let us start by looking at the code:

```
///////////////////////////////////////////////////////////////////
//
// Write routine for the Trimesh file format
//
///////////////////////////////////////////////////////////////////

#include <hxcore/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <hxsurface/HxSurfaceField.h>
#include <mypackage/mypackageAPI.h>

static
int writetrimesh(HxSurface* surface,
    McDArray<HxSurfaceField*> fields, const char* filename)
{
    FILE *f = fopen(filename, "w");

    if (!f) {
        theMsg->ioError(filename);
        return 0;
    }

    int i,j,k;
    McDArray<McVec3f>& points = surface->points;
    McDArray<Surface::Triangle>& triangles = surface->triangles;

    // Write number of points and number of triangles
    fprintf(f, "%d %d\n", points.size(), triangles.size());

    // Write point coordinates
    for (i=0; i<points.size(); i++) {
        McVec3f& v = points[i];
        fprintf(f, "%g %g %g\n", v[0], v[1], v[2]);
    }

    // Write point indices of all triangles
    for (i=0; i<triangles.size(); i++) {
        int* idx = triangles[i].points;
        fprintf(f, "%d %d %d\n", idx[0]+1, idx[1]+1, idx[2]+1);
    }

    // If there are data fields write them out too.
    if (fields.size()) {
        for (j=0; j<fields.size(); j++)
            fprintf(f, "%d ", fields[j]->nDataVar());
        fprintf(f, "\n");
```

```
        for (i=0; i<points.size(); i++) {
            for (j=0; j<fields.size(); j++) {
                int n = fields[j]->nDataVar();
                float* v = &fields[j]->dataPtr()[i*n];
                for (k=0; k<n; k++)
                    fprintf(f, "%g ", v[k]);
            }
            fprintf(f, "\n");
        }
    }

    fclose(f); // done
    return 1;
}

MYPACKAGE_API
int writetrimesh(HxSurface* surface, const char* filename)
{
    // Temporary array of surface data fields
    McDArray<HxSurfaceField*> fields;

    // Check if there are data fields attached to surface
    for (int i=0; i<surface->downStreamConnections.size(); i++) {
        HxSurfaceField* field =
            (HxSurfaceField*) surface->downStreamConnections[i];
        if (field->isOfType(HxSurfaceField::getClassTypeId()) &&
            field->getEncoding() == HxSurfaceField::OnNodes)
            fields.append(field);
    }

    // Write surface and all attached data fields
    return writetrimesh(surface, fields, filename);
}

MYPACKAGE_API
int writetrimesh(HxSurfaceField* field, const char* filename)
{
    // Check if data is defined on nodes
    if (field->getEncoding() != HxSurfaceField::OnNodes) {
        theMsg->printf("Data must be defined on nodes.");
        return 0;
    }

    // Store pointer to field in dynamic array
    McDArray<HxSurfaceField*> fields;
    fields.append(field);
```

```
    // Write surface and this data field
    return writetrimesh(field->surface(), fields, filename);
}
```

In the upper part of the code, first a static utility method is defined which takes three arguments: a pointer to a surface, a dynamic array of pointers to surface fields, and a file name. This is the function that actually writes the data to a file. Once you have understood the *Trimesh* reader presented in Section 3.2.2, it should be no problem to follow the writer code too.

In the lower part of the code, two write routines mentioned above are defined, one for surfaces and the other one for surface fields. Since these routines are to be exported for external use, we need to apply the package macro `MYPACKAGE_API`, at least on Windows.

Let us now look more closely at the surface writer. This routine first collects all surface fields attached to the surface in a dynamic array. This is done by scanning `surface->downStreamConnections` which provides a list of all objects attached to the surface. The class type of each object is checked using the method `isOfType`. This sort of dynamic type-checking is the same as in Open Inventor. If a surface field has been found and if it contains data defined on its nodes, it is appended to the temporary array `fields`. The surface itself, as well as the collected fields, are then written to file by calling the utility method defined in the upper part of the writer code.

The second write routine, the one adapted to surface fields, is simpler. Here a dynamic array of fields is used too, but this array is filled with data representing the original surface field only. Once this has been done, the same utility method can be called as in the first case.

Although actually two write routines have been defined, only one entry in the package resource file is required. This entry looks as follows (see `mypackage/share/resources/mypackage.rc`):

```
dataFile -name "Trimesh Demo Format"    \
    -ext "trimesh"                      \
    -type "HxSurface"                   \
    -type "HxSurfaceField"              \
    -save "writetrimesh"                \
    -package "mypackage"
```

In order to compile and execute the write, please follow the instructions given in Section 1.5 (Compiling and Debugging).

3.4 The AmiraMesh API

Besides many standard file formats, Amira also provides its own native format called AmiraMesh. The AmiraMesh file format is very flexible. It can be used to save many different data objects including image data, finite-element grids, and solution data defined on such grids. Among other features it supports ASCII or binary data encoding, data compression, and storage of arbitrary parameters. The format itself is described in more detail in the reference section of the user's guide. In this section we want to discuss how to save custom data objects in AmiraMesh format. For this purpose a special C++ utility class called `AmiraMesh` is provided. Using this class, reading and writing AmiraMesh files becomes very easy.

Below we will first provide an *overview* of the AmiraMesh API. After that, we present two simple examples. In the first one we show how colormaps are *written* in AmiraMesh format. In the second one we show how such colormaps are *read back* again.

3.4.1 Overview

The AmiraMesh API consists of a single C++ class. This class is called `AmiraMesh` as is the file format itself. It is defined in the header file `include/amiramesh/AmiraMesh.h` located in the Amira root directory. The class is designed to completely represent the information stored in an AmiraMesh file in memory. When reading a file first an instance of an `AmiraMesh` class is created. This instance can then be interpreted and the data contained in it can be copied into a matching Amira data object. Likewise, when writing a file, first an instance of an `AmiraMesh` class is created and initialized with all required information. Then this instance is written to file simply by calling a member method.
If you look at the header file or at the `AmiraMesh` class documentation, you will notice that there are four public member variables called `parameters`, `locationList`, `dataList`, and `fieldList`. These variables completely store the information contained in a file. The first variable is of type `HxParamBundle`. Like in an Amira data object, it is used to store an arbitrary hierarchy of parameters. The other three member variables are dynamic arrays of pointers to locally defined classes. The most important local classes are `Location` and `Data`, which are stored in `locationList` and `dataList`, respectively.

A `Location` defines the name of a single- or multi-dimensional array. It does not store any data by itself. This is done by a `Data` class. Every `Data` class must refer to some `Location`. For example, when writing a tetrahedral grid, we may define two different one-dimensional locations, one called *Nodes* and the other one called *Tetrahedra*. On the nodes we define a `Data` instance for storing the x-, y-, and z-coordinates of the nodes. Likewise, on the tetrahedra we define a `Data` instance for storing the indices of the four points of a tetrahedron.
As stated in the `AmiraMesh` class documentation, the *Data* class can take a pointer to some already existing block of memory. In this way it is prevented that all data must be copied before it is written to file. In order to write compressed data, the member method `setCodec` has to be called. Currently, two different compression schemes are supported. The first one, called *HxByteRLE*, implements simple run-length encoding on a per-byte basis. The second one, called *HxZip*, uses a more sophisticated compression technique provided by the external *zlib* library. In any case, the data will be automatically uncompressed when reading an AmiraMesh file.
It should be pointed out that the AmiraMesh file format itself merely provides a method for storing arbitrary data organized in single- or multi-dimensional arrays in a file. It does not specify anything about the semantics of the data. Therefore, when reading an AmiraMesh file it is not clear what kind of data object should be created from it. To facilitate file I/O of custom data objects, the actual contents of an AmiraMesh file are indicated by a special parameter called *ContentType*. For each such type, a special read routine is registered. Like an ordinary read routine, an AmiraMesh reader is a global function or a static member method of a C++ class. It has the following signature:

```
int readMyAmiraMesh(AmiraMesh* m, const char* filename);
```

This method is called whenever the *ContentType* parameter matches the the one the read method is registered for. The reader should create an Amira data object from the contents of the AmiraMesh class. The filename can be used to define the name of the resulting data object. In order to register an AmiraMesh read routine, a statement similar to the following one must be put into the package resource file:

```
amiramesh -ContentType "MyType" \
    -load "readMyAmiraMesh" \
    -package "mypackage"
```

3.4.2 Writing an AmiraMesh File

As a concrete example, in this section we want to show how a colormap is written in AmiraMesh format. In particular, we consider colormaps of type `HxColormap256`, consisting of N discrete RGBA tuples. Like most other write methods, the AmiraMesh writer is a global C++ function. Let us first look at the code before discussing the details.

```
HXCOLOR_API
int writeAmiraMesh(HxColormap256* map, const char* filename)
{
    float minmax[2];
    minmax[0] = map->minCoord();
    minmax[1] = map->maxCoord();
    int size = map->getLength();

    AmiraMesh m;
    m.parameters = map->parameters;
    m.parameters.set("MinMax", 2, minmax);
    m.parameters.set("ContentType", "Colormap");

    AmiraMesh::Location* loc =
        new AmiraMesh::Location("Lattice", 1, &size);
    m.insert(loc);

    AmiraMesh::Data* data = new AmiraMesh::Data("Data", loc,
        McPrimType::mc_float, 4, (void*) map->getDataPtr());
    m.insert(data);

    if ( !m.write(filename,1) ) {
        theMsg->ioError(filename);
        return 0;
    }

    setLoadCmd(filename);
    return 1;
}
```

In the first part of the routine a variable `m` of type `AmiraMesh` is defined. The parameters of the colormap are copied into `m`. In addition, two more parameters are set. The first one, called *MinMax*, describes the coordinate range of the colormap. The second one indicates the content type of the AmiraMesh file. This parameter ensures that the colormap can be read back again by a matching AmiraMesh read routine (see Section 3.4.3).
Before the RGBA data values can be stored, a `Location` of the right size must

be created and inserted into the `AmiraMesh` class. Afterwards, an instance of a `Data` class is created and inserted. The constructor of the `Data` class takes a pointer to the `Location` as an argument. Moreover, a pointer to the RGBA data values is specified. Each RGBA tuple consists of four numbers of type float.

3.4.3 Reading an AmiraMesh File

In the previous section we presented a simple AmiraMesh write routine for colormaps. We now want to read back such files again. For this reason we define a static AmiraMesh read function in class `HxColormap256`. Of course, a global C++ function could be used as well. The read function is registered in the package resource file `hxcolor.rc` in the following way:

```
amiramesh -ContentType "Colormap" \
    -load "HxColormap256::readAmiraMesh" \
    -package "hxcolor"
```

This statement indicates that the static member method `readAmiraMesh` of the class `HxColormap256` defined in package `hxcolor` should be called if the AmiraMesh file contains a parameter *ContentType* equal to `Colormap`. The source code of the read routine looks as follows:

```
int HxColormap256::readAmiraMesh(AmiraMesh* m,
    const char* filename)
{
    for (int i=0; i<m->dataList.size(); i++) {
        AmiraMesh::Data* data = m->dataList[i];

        if (data->location()->nDim() != 1)
            continue;

        if (data->dim()<3 || data->dim()>4)
            continue;

        if (data->primType() != McPrimType::mc_uint8 &&
            data->primType() != McPrimType::mc_float)
            continue;

        int dim = data->dim();
        int size = data->location()->dims()[0];

        HxColormap256* colormap = new HxColormap256(size);
        colormap->parameters = m->parameters;
```

```
        switch (data->primType()) {
        case McPrimType::mc_uint8: {
            unsigned char* src =
                (unsigned char*) data->dataPtr();
            for (int k=0; k<size; k++, src+=dim) {
                float a = (dim>3) ? (src[3])/255.0 : 1;
                colormap->setRGBA(k, src[0]/255., src[1]/255.,
                    src[2]/255., a);
            } } break;

        case McPrimType::mc_float: {
            float* src = (float*) data->dataPtr();
            for (int k=0; k<size; k++, src+=dim) {
                float a = (dim>3) ? src[3] : 1;
                colormap->setRGBA(k, src[0], src[1], src[2], a);
            } } break;
        }

        float minmax[2] = { 0,1 };
        m->parameters.findReal("MinMax", 2, minmax);
        colormap->setMinMax(minmax[0], minmax[1]);

        HxData::registerData(colormap, filename);
        return 1;
    }

    return 0;
}
```

Compared to the write routine, the read routine is a little bit more complex since some consistency checks are performed. First, the member `dataList` of the `AmiraMesh` structure is searched for a one-dimensional array containing vectors of three or four elements of type byte or float. This array should contain the RGB or RGBA values of the colormap. If a matching `Data` structure is found, a new instance of type `HxColormap256` is created. The parameters are copied from the `AmiraMesh` class into the new colormap. Afterwards, the actual color values are copied. Although the write routine only exports RGBA tuples of type float, the read routine also supports byte data. For this reason two different cases are distinguished in a switch statement. If the file only contains 3-component data, the opacity value of each colormap entry is set to 1. Finally, the coordinate range of the colormap is set by evaluating the 2-component parameter *MinMax*, and the new colormap is added to the Pool by calling `HxData::registerData`.

4 Writing Modules

Besides the data classes, modules are the core of Amira. They contain the actual algorithms for visualization and data processing. Modules are instances of C++ classes derived from the common base class `HxModule`.
There are two major groups of modules: *compute modules* and *display modules*. The first group usually performs some sort of operation on input data, creates some resulting data object, and deposits the latter in the Pool. In contrast, display modules usually directly visualize their input data. In this chapter both types of modules will be covered in separate sections. For each case a concrete example will be presented and discussed in detail.
In addition, we also discuss the Amira*Plot API* in this chapter. This API makes it possible to create simple line plots or bar charts within a module.

4.1 A Compute Module

As already mentioned *compute modules* usually take one or more input data objects and calculate a new resulting data object from these. The resulting data object is deposited in the Pool. Compute modules are represented by red icons in the Pool. They are derived from the base class `HxCompModule`.
In order to learn how to implement a new compute module, we will take a look at a concrete example. In particular, we want to write a compute module which performs a threshold operation on a 3D image, i.e., on an input object of type *HxUniformScalarField3*. The module produces another 3D image as output. In the resulting image, all voxels with a value below a user-specified minimum value or above a maximum value should be set to zero.
For easier understanding we start with a very simple and limited version of the module. Then we iteratively improve the code. In particular, we proceed in three steps:

- *Version 1*: merely scans the input image, does not yet produce a result
- *Version 2*: creates an output object as result, uses the progress bar

- *Version 3*: adds an *Apply* button, overwrites the existing result if possible

You can find the source code of all three versions in the demo package provided with Developer Option, i.e., under `packages/mypackage` in the local Amira directory. For each version there are two files: a header file called `MyComputeThresholdN.h` and a source code file called `MyComputeThresholdN.cpp` (where `N` is either 1, 2, or 3). Since the names are different, you can compile and execute all three versions in parallel.

Afterwards we describe how to implement an *ITK image to image filter compute module*.

In order to create a new local Amira directory, please follow the instructions given in Section 2.2. In order to compile the demo package, please refer to Section 1.5 (Compiling and Debugging).

4.1.1 Version 1: Skeleton of a Compute Module

The first version of our module does not yet produce any output. It simply scans the input image and prints the number of voxels above and below the threshold. Like most other modules, our compute module consists of a header file containing the class declaration as well as a source file containing the actual code (or the class definition). Let us look at the header file `MyComputeThreshold1.h` first:

```
///////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
///////////////////////////////////////////////////////////////
#ifndef MY_COMPUTE_THRESHOLD_H
#define MY_COMPUTE_THRESHOLD_H

#include <hxcore/HxCompModule.h>
#include <hxcore/HxPortFloatTextN.h>
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyComputeThreshold1 : public HxCompModule
{
    // This macro is required for all modules and data objects
    HX_HEADER(MyComputeThreshold1);

  public:
    // Every module must have a default constructor.
    MyComputeThreshold1();
```

```
    // This virtual method will be called when the port changes.
    virtual void compute();

    // A port providing float text input fields.
    HxPortFloatTextN portRange;
};

#endif
```

As usual in C++ code, the file starts with a define statement that prevents the contents of the file from being included multiple times. Then three header files are included. `HxCompModule.h` contains the definition of the base class of our compute module. The next file, `HxPortFloatTextN.h`, contains the definition of a *port* we want to use in our class.

A port represents an input parameter of a module. In our case we use a port of type `HxPortFloatTextN`. This port provides one or more text fields where the user can enter floating point numbers. The required text fields and labels are created automatically within the port constructor. As a programmer you simply put some ports into your module, specifying their types and labels, and do not have to bother creating a user interface for it.

Following `HxPortFloatTextN.h`, the package header file `mypackageAPI.h` is included. This file provides import and export storage-class specifiers for Windows systems. These are encoded in the macro `MYPACKAGE_API`. A class declared without this macro will not be accessible from outside the DLL it is defined in. On Unix systems the macro is empty and can be omitted.

In the rest of the header file nothing more is done than deriving a new class from `HxCompModule` and defining two member functions, namely the constructor and an overloaded virtual method called `compute`. The `compute` method is called when the module has been created and whenever a change of state occurs on one of the module's input data objects or ports. In fact, a connection to an input data object is also established by a port, as we shall see later on. In this example we just declare one port in our class, specifically an instance of type `HxPortFloatTextN`.

The corresponding source file looks like this:

```
///////////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
```

```
////////////////////////////////////////////////////////////////

#include <hxcore/HxMessage.h>
#include <hxfield/HxUniformScalarField3.h>
#include <mypackage/MyComputeThreshold1.h>

HX_INIT_CLASS(MyComputeThreshold1,HxCompModule) // required macro

MyComputeThreshold1::MyComputeThreshold1() :
    HxCompModule(HxUniformScalarField3::getClassTypeId()),
    portRange(this,"range",2) // we want to have two float fields
{
}

void MyComputeThreshold1::compute()
{
    // Access the input data object. The member portData, which
    // is of type HxConnection, is inherited from HxModule.
    HxUniformScalarField3* field =
        (HxUniformScalarField3*) portData.source();

    // Check whether the input port is connected
    if (!field) return;

    // Get the input parameters from the user interface:
    float minValue = portRange.getValue(0);
    float maxValue = portRange.getValue(1);

    // Access size of data volume:
    const int* dims = field->lattice.dims();

    // Now loop through the whole field and count the pixels.
    int belowCnt=0, aboveCnt=0;
    for (int k=0; k<dims[2]; k++) {
        for (int j=0; j<dims[1]; j++) {
            for (int i=0; i<dims[0]; i++) {
                // This function returns the value at the specific
                // grid node. It implicitly casts the result
                // to float if necessary.
                float value = field->evalReg(i,j,k);
                if (value<minValue)
                    belowCnt++;
                else if (value>maxValue)
                    aboveCnt++;
            }
        }
    }
```

```
    // Finally print the result.
    theMsg->printf("%d voxels < %g, %d voxels > %g\n",
        belowCnt, minValue, aboveCnt, maxValue);
}
```

Following the include statements and the obligatory `HX_INIT_CLASS` macro, the constructor is defined. The usual C++ syntax must be used in order to call the constructors of the base class and the class members. The constructor of the base class `HxCompModule` takes the class type of the input data object to which this module can be connected. Amira uses a special run-time type information system that is independent of the rtti feature provided by the newer ANSI C++ compilers.

The second method we have to implement is the `compute` method. We first retrieve a pointer to our input data object through a member called `portData`. This port is inherited from the base class `HxModule`, i.e., every module has this member. The port is of type `HxConnection` and it is represented as a blue line in the user interface (if connected). The rest of the compute method is rather straightforward. The way the actual data are accessed and how the computation is performed, of course, is highly specific to the input data class and the task the module performs. In this case we simply loop over all voxels of the input image and count the number of voxels below the minimum value and above the maximum value. In order to access a voxel's value, we use the *evalReg* method. This method is provided by any scalar field with regular coordinates, i.e., by any instance of class `HxRegScalarField3`. Regardless of the primitive data type of the field, the result will always be cast to float.
Once you have compiled the `mypackage` demo package, you can load the file `lobus.am` from Amira 's `data/tutorials` directory and attach the module to it. Try to type in different threshold values, or use different input data sets. Instructions for compiling local packages are provided in Section 1.5 (Compiling and Debugging).

4.1.2 Version 2: Creating a Result Object

Now that we have a first working version of the module, we can add more functionality. First, we want to create a real output data object. Then we further want to improve the module by using Amira's progress bar and by providing better default values for the range port. The header file of our module will not be affected by all these changes. We merely need to add some code in the source file

`MyComputeThreshold2.cpp`.
Let us start with the output data object. In the compute method just before the for-loop, we insert the following statements:

```
// Create output with same primitive data type as input:
HxUniformScalarField3* output =
    new HxUniformScalarField3(dims, field->primType());

// Output shall have same bounding box as input:
output->coords()->setBoundingBox(field->bbox());
```

This creates a new instance of type `HxUniformScalarField3` with the same dimensions and the same primitive data type as the input data object. Since the output has the same bounding box, i.e., the same voxel size as the input, we copy the bounding box. Note that this approach will only work for fields with uniform coordinates. For other regular coordinate types such as stacked or curvilinear coordinates, we refer to Section 5.2.
After the output object has been created, its voxel values are not yet initialized. This is done in the inner part of the nested for-loops. The method `set`, used for this purpose, automatically performs a cast from float to the primitive data type of the output field. In summary, the inner part of the for-loop now looks as follows:

```
float value = field->evalReg(i,j,k);
float newValue = 0;

if (value<minValue)
    belowCnt++;
else if (value>maxValue)
    aboveCnt++;
else newValue = value;

output->set(i,j,k,newValue);
```

Creating a new data object using the `new` operator will not automatically make it appear in the Pool. Instead, we must explicitly register it. In a compute module this can be done by calling the method `setResult`:

```
setResult(output); // register result
```

This method adds a data object to the Pool if it is not already present there. In addition, it connects the object's *master* port to the compute module itself. Like any other connection, this link will be represented by a blue line in the Pool. The master port of a data object may be connected to a compute module or to an editor. Such a master connection indicates that the data object is controlled by an 'upstream' component, i.e., that its contents may be overridden by the object it is connected to.

Now that we have created an output object, let us address the progress bar. Although for the test data set `lobus.am` our threshold operation does not take very long, it is good practice to indicate that the application is busy when computations are performed that could take long time on large input data. Even better is to show a progress bar, which is not difficult. Before the time-consuming part of the compute routine, i.e., before the nested for-loops, we add the following line:

```
// Turn the application into busy state,
// don't activate Stop button.
theWorkArea->startWorkingNoStop("Computing threshold");
```

We use the global instance `theWorkArea` of class `HxWorkArea` here. The corresponding header file must be included at the beginning of the source file. The method turns the application into the 'busy' state and displays a working message in the status line. As opposed to the method `startWorking`, this variant does not activate the stop button. See Section 8.2 for details. When the computation is done, we must call

```
theWorkArea->stopWorking(); // stop progress bar
```

in order to switch off the 'busy' state again. Inside the nested for-loops we update the progress bar just before a new 2D slice is processed. This is done by the following line of code:

```
    // Set progress bar, the argument ranges between 0 and 1.
    theWorkArea->setProgressValue((float)(k+1)/dims[2]);
```

The value of `(float)(k+1)/dims[2]` progressively increases from zero to one during computation. Note that you should not call `setProgressValue` in the inner of the three loops. Each call involves an update of the graphical user interface and therefore is relatively expensive. It is perfectly okay to update the progress bar several hundred times during a computation, but not several hundred thousand times.

Another slight improvement we have incorporated into the second version of our compute module concerns the `range` port. In the constructor we have set new initial values for the minimum and maximum fields. While both values are 0 by default, we now set them to 30 and 200, respectively:

```
// Set default value for the range port:
portRange.setValue(0,30);  // min value is 30
portRange.setValue(1,200); // max value is 200
```

You may now test this second version of the compute module by loading the test data set `lobus.am` from Amira's `data/tutorials` directory. Attach the `ComputeThreshold2` module to it. To better appreciate the progress bar, try to resample the input data, for example to 512x512x100, and connect the compute module to the resampled data set. However, be sure that you have enough main memory installed on your system.

4.1.3 Version 3: Reusing the Result Object

Testing the first two versions of our module, we saw that the module's compute method is triggered automatically when the module is created and whenever the range port is changed. Each time a new result output data object is created. This quickly fills up the computer's main memory as well as Amira's graphical user interface. Therefore, we now change this behavior: A new result object is to be created only the first time. Whenever the range port is changed afterwards, the existing result object should be overridden. In order to achieve this, we modify the middle part of the compute method in the following way:

```
// Check if there is a result which we can reuse.
HxUniformScalarField3* output =
    (HxUniformScalarField3*) getResult();

// Check for proper type.
if (output && !output->isOfType(
        HxUniformScalarField3::getClassTypeId() ))
    output = 0;

// Check if size and primType still match the current input:
if (output) {
    const int* outdims = output->lattice.dims();
    if (dims[0]!=outdims[0] ||dims[1]!=outdims[1] ||
        dims[2]!=outdims[2] ||
        field->primType() != output->primType())
        output=0;
}

// If necessary, create a new result data set.
if (!output) {
    output = new HxUniformScalarField3(dims,
```

```
            field->primType());
        output->composeLabel(field->getName(),"masked");
    }
```

The `getResult` method checks whether there is a data set whose master port is connected to the compute module. This typically is the object set by a previous call to `setResult`. However, it also may be any other object. Therefore, a run-time type check must be performed by calling the `isOfType` member method of the output object. If the output object is not of type `HxUniformScalarField3`, the variable `output` will be set to null. Then a check is made whether the output object has the same dimensions and the same primitive data type as the input object. If this test fails, `output` will also be set to null. At the end, a new result object will only be created if no result exists already or if the existing result does not match the input. It is possible to interactively try different range values without creating a bunch of new results.
However, when one of the numbers of the range port is changed, computation starts immediately. Sometimes this may be desired, but in this case we prefer to add an *Apply* button as present in many other compute modules. The user must explicitly push this button in order to start computation. In order to use the *Apply* button, the following line of code must be added in the public section of the module's header file:

```
    // Start computation when this button is clicked.
    HxPortDoIt portDoIt;
```

Of course, the corresponding include file `Amira/HxPortDoIt.h` must be included as well. As for the other port, we must initialize `portDoIt` in the constructor of our module in the source file:

```
MyComputeThreshold3::MyComputeThreshold3() :
    HxCompModule(HxUniformScalarField3::getClassTypeId()),
    portRange(this,"range",2), // we want to have two float fields
    portDoIt(this,"action")
{
    ...

    // Set text of doIt button
    portDoIt.setLabel(0,"DoIt");
}
```

To achieve the desired behavior we finally change our compute method so that it immediately returns unless the *Apply* button was pressed. This can be done by adding the following piece of code at the beginning of the compute method:

```
    // Check whether doIt button was hit
    if (!portDoIt.wasHit()) return;
```

With these changes, the module is already quite usable. Try to attach the final version of the module to some data set, press *Apply*, change the range and press *Apply* again. Attach an *OrthoSlice* module to the result while experimenting with the range (use the histogram mapping in the *OrthoSlice* in order to see small changes). Try to detach the connection between the result and the module and press *Apply* again.

Note: Since Amira 4.0, the `HxPortDoIt` port is not (by default) visible in the control panel of its associated module. Rather, the fact that a module has an `HxPortDoIt` activates (makes green) the *Apply* button at the bottom of the Properties Area. To request display of the DoIt port in the module control panel, check the *Show "DoIt" buttons* box in the *Layout* tab of the *Edit/Preferences* dialog.

Finally, some remarks on performance. Although it is probably not critical in this simple example, performance typically becomes an issue in real-world applications. In the inner-most loop, calling the methods `field-> evalReg` and `output-> set` is convenient but rather expensive. For example, if the input consists of bytes like in `lobus.am`, these methods involve a cast from `unsigned char` to `float` and back to `unsigned char`.

The performance can be improved by writing code which explicitly handles a particular primitive data type. A pointer to the actual data values of a *HxUniformScalarField3* can be obtained by calling `field-> lattice.dataPtr()`. The value returned by this method is of type `void*`. It must be explicitly cast to the data type the field actually belongs to. The voxel values itself are arranged without any padding. This means that the index of voxel (i, j, k) is given by `(k*dims[1]+j)*dims[0]+i`, where `dims[0]` and `dims[1]` denote the number of voxels in the x and y directions, respectively.

4.1.4 Implement an ITK image to image filter

At this point basic understanding of Amira's compute module architecture is assumed.

This example will show how to write an Amira compute module implementing an ITK image to image filter. In a step by step manner we will show how to

- use Amira's ports to specify parameters needed by ITK filters,
- import an Amira image data object into the ITK filter pipeline and export the ITK result image data object, respectively,

- monitor the progress of ITK image filters within Amira's global progress bar,
- update the ITK filter pipeline by the Amira compute module.

In our example an ITK mean image filter with a variable kernel size will be implemented.
You will find the source code within the ITK demo package provided with the Developer Option, i.e. under `packages/myitkpackage` in the local Amira directory. There will be two files: a header file called `MyITKFilter.h` and a source code file called `MyITKFilter.cpp`.
Let us look at the header file `MyITKFilter.h` first.

```
/////////////////////////////////////////////////
//
// Example of an ITK compute module
//
/////////////////////////////////////////////////

#ifndef MYITKFILTER_H
#define MYITKFILTER_H

#include <hxcore/HxCompModule.h>
#include <hxcore/HxPortIntSlider.h>
#include <hxcore/HxPortDoIt.h>

// storage-class specification
#include <myitkpackage/myitkpackageAPI.h>

class MYITKPACKAGE_API MyITKFilter : public HxCompModule
    HX_HEADER(MyITKFilter);

  public:

    MyITKFilter();
    ~MyITKFilter();

    HxPortIntSlider   portKernelSize;
    HxPortDoIt        portDoIt;

    virtual void compute();
};

#endif // MYITKFILTER_H
```

As usual in C++ code, the file starts with a define statement that prevents the contents of the file from being included multiple times. Then four header files are

included. `HxCompModule.h` contains the definition of the base class of our compute module. The next two files, `HxPortIntSlider.h` and `HxPortDoIt.h`, contain the definitions of two *ports* we want to use in our class.
In our example we use the `HxPortIntSlider` to specify the kernel size of the filter and the `HxPortDoIt` port in order to apply the computation.
The `myitkpackageAPI.h` header file is included to provide import export storage-class specifiers for Windows systems. These are encoded in the macro `MYITKPACKAGE_API`. A class declared without this macro will not be accessible from outside the DLL it is defined in. On Unix systems the macro is empty and can be omitted.
In the rest of the header file nothing more is done than deriving a new class from `HxCompModule` and defining three member functions, namely the constructor, destructor and an overloaded virtual method called `compute`. The `compute` method is called when the module has been created and whenever a change of state occurs on one of the module's input data objects or ports.
Let's look at some code snippets of the source code file `MyITKFilter.cpp` in order to clarify important parts of the implementation.
In our example the following headers from Amira's `hxitk` package are included, providing convenience functionality for wrapping ITK image data objects and monitoring the progress of ITK process objects.
Note that ITK headers have to be included without specifying a path in front of the header file name.

```
...
#include <hxitk/HxItkImageImporter.h>
#include <hxitk/HxItkImageExporter.h>
#include <hxitk/HxItkProgressObserver.h>

#include "itkMeanImageFilter.h"
...
```

As usual the module's compute method will be triggered automatically if the module is created and whenever the kernel size port or the data respectively have changed. However our compute method should return immediately unless the *Apply* button has been pressed. This will be done by the following piece of code at the beginning of the compute method:

```
    ...
    // Check whether doIt button was hit
    if (!portDoIt.wasHit()) return;
    ...
```

The `getResult` method checks whether there is a data set whose master port is connected to the compute module. This typically is the object set by a previous call to `setResult`. However, it also may be any other object. Therefore, a run-time type check will be performed by casting the result returned by `getResult` via a `mcinterface_cast`. If the result returned by `getResult` cannot be casted to the type of a `HxUniformScalarField3`, the variable `resultField` will be set to null. Then a check is made whether the output object has the same dimensions and the same primitive data type as the input object. If this test fails, `resultField` will also be set to null. Note, that only if the variable `resultField` is null a new output object of type `HxUniformScalarField3` will be created later on.

```
void MyITKFilter::compute()
{
    ...
    // Access the input data object. The member portData
    // (which is of type HxConnection) is inherited from
    // the base class HxModule.
    HxUniformScalarField3* inField =
        hxconnection_cast<HxUniformScalarField3>( portData );

    // Check whether the input port is connected
    if (!inField) return;

    // Access size of data volume:
    const int* dims = inField->lattice.dims();

    // Check if there is a result which we can reuse.
    McHandle<HxUniformScalarField3> resultField =
        mcinterface_cast<HxUniformScalarField3>( getResult() );

    // Check for proper type.
    if (resultField &&
           !resultField->isOfType(
           HxUniformScalarField3::getClassTypeId() )
       )
        resultField = 0;

    // Check if size and primType still match the current input:
    if (resultField) {
        const int* outdims = resultField->lattice.dims();
        if (dims[0]!=outdims[0] ||
            dims[1]!=outdims[1] ||
            dims[2]!=outdims[2] ||
            resultField->primType() != resultField->primType())
```

```
        resultField=0;
    }
    ...
```

Due to the fact, that ITK is a template library we have to translate the implicitly encoded type information of Amira's data objects into template code. This is achieved by instantiating different versions of a function template `meanImageFilter` where the actual ITK filtering is implemented.

```
    ...
    switch (inField->lattice.primType()) {
            case McPrimType::mc_uint8:
                meanImageFilter<unsigned char>(inField,
                resultField, kernelSize);
                break;
            case McPrimType::mc_uint16:
                meanImageFilter<unsigned short>(inField,
                resultField, kernelSize);
                break;
            case McPrimType::mc_int16:
                meanImageFilter<short>(inField,
                resultField, kernelSize);
            ....
    }
    ...
```

The following steps implement the ITK image to image filter pipeline within the `meanImageFilter` function template.

- Create the ITK filter pipeline by defining certain ITK typedefs of input and output images and filters with respect to the template type `Type`. In our example we create a single ITK process object e.g. a mean image filter.

  ```
  ...
  typedef itk::Image< Type, 3 > ImageType;
  typedef itk::MeanImageFilter<ImageType,
      ImageType > FilterType;

  FilterType::Pointer filter = FilterType::New();
  ...
  ```

- Wrap the Amira input data object of type `HxUniformScalarField3` into an ITK image object with the help of the `HxItkImageImporter` template class. The result of `HxItkImageImporter::getOutput()` has to be passed as input image to the first process object of the filtering pipeline. Note that the `HxItkImageImporter` only maps the data of the `HxUniformScalarField3` into the ITK image data object. Meaning that no additional memory will be allocated, but the ITK image data object will use the memory allocated by the Amira data object.

```
...
HxItkImageImporter<Type> importer(inField);
filter->SetInput(importer.GetOutput());
...
```

- Provide an Amira data object of type `HxUniformScalarField3` where the result image of the ITK filter pipeline should be written. Therefore the ouput image of the last ITK process object of the filter pipeline has to be wrapped by the `HxItkImageExporter` template class. If an existing `resultField` has been passed to the constructor the data of the `HxUniformScalarField3` will be mapped into the ITK image data object. Again, the ITK output image will not allocate memory, but use the data allocated by the Amira data object. If no valid pointer of type `HxUniformScalarField3` has been provided via the constructor ITK will allocate the memory. Later on, a new Amira data object of type `HxUniformScalarField3` will be created by `HxItkImageExporter::getResult()`, which afterward takes over the buffer allocated by the ITK image (see below).

```
...
HxItkImageExporter<ImageType> exporter(
    filter->GetOutput(),
    resultField );
...
```

- Observe the ITK process object's progress and visualize it within Amira's global progressbar. Therefore an instance of `HxItkProgressObserver` needs to be instantiated. In order to monitor the progress of an ITK process object it has to be registered via the `HxItkProgressObserver::startObservingProcessObject()` member function.

```
...
/// Display filter progress within Amira's progressbar
HxItkProgressObserver progress;
progress.startObservingProcessObject(filter);
progress.setProgressMessage(
```

```
        McString("Applying mean filter...") );
    ...
```

- Update the ITK filter pipeline and get the result. If a null pointer has been passed to the constructor of `HxItkImageExporter`, `HxItkImageExporter::getResult()` will create a new Amira data object, which takes over the buffer allocated by the ITK image.

  In order to avoid memory leaks `HxItkImageExporter` holds a handle (see `McHandle`) on the newly created Amira data object of type `HxUniformScalarField3`. Thus a variable of type `McHandle<HxUniformScalarField3>` has to be used in order to store the result and increase the reference count of the Amira data object. Otherwise the newly created result data object will be deleted within the exporter's destructor.

  If an existing result field of type `HxUniformScalarField3` has been passed to the constructor of `HxItkImageExporter`, calling `HxItkImageExporter::getResult()` isn't necessary because the pointer to the reused Amira data object won't change.

```
    ...
    // Execute the filter
    filter->Update();
    resultField = exporter.getResult();
```

4.2 A Display Module

Our next example is a module which displays some geometry in Amira's 3D viewer. The module takes a surface model as input and draws a little cube at every vertex that belongs to *n* triangles, where *n* is a user-adjustable parameter.
From the previous section we already know the basic idea: We derive a new class from the base class `HxModule`. Since this time our module does not produce a new data set we directly use `HxModule` as base class instead of `HxCompModule`. As input the module should accept data of class `HxSurface`. We need one additional port allowing the user to specify the parameter *n*. As in the previous section we develop different versions of our module, thereby introducing new concepts step by step:

- *Version 1*:
  creates an Open Inventor scene graph and displays it in the viewer

- *Version 2*:
  adds a colormap port, provides a parse method for Tcl commands
- *Version 3*:
  implements a new display mode, dynamically shows or hides a port

You can find the source code of all three versions of the module in the demo package provided with Developer Option, i.e., under `packages/mypackage` in the local Amira directory. For each version there are two files, a header file called `MyDisplayVerticesN.h` and a source code file called `MyDisplayVerticesN.cpp` (where `N` is either 1, 2, or 3). Since the names are different you can compile and execute all three version in parallel.
In order to create a new local Amira directory, please follow the instructions given in Section 2.2. In order to compile the demo package, please refer to Section 1.5 (Compiling and Debugging).

4.2.1 Version 1: Displaying Geometry

The first version of our module, called *MyDisplayVertices1*, merely detects the vertices of interest and displays them using little cubes. In order to understand the code, we first need to look more closely at the class `HxSurface`. As we can see in the reference documentation, a surface essentially contains an array of 3D points and an array of triangles. Each triangle has three indices pointing into the list of points. In order to count the triangles per vertex, we simply walk through the list of triangles and increment a counter for each vertex.
Once we have detected all interesting vertices, we are going to display them using small cubes. This is done by creating an Open Inventor scene graph. If you want to learn more about Open Inventor, you probably should look at *The Inventor Mentor*, an excellent book about Open Inventor published by Addison-Wesley. In brief, an Open Inventor scene graph is a tree-like structure of C++ objects which describes a 3D scene. Our scene is quite simple. It consists of one *separator node* containing several cubes, i.e., instances of class `SoCube`. Since an `SoCube` is always located at the origin, we put an additional node of type `SoTranslation` right before each `SoCube`. We adjust the size of the cubes so that each side is 0.01 times the length of the diagonal of the bounding box of the input surface.
After this short overview we now look at the header file of the module. It is called `MyDisplayVertices1.h`:

```
////////////////////////////////////////////////////////////////////
//
```

```
// Example of a display module
//
/////////////////////////////////////////////////////////////////
#ifndef MY_DISPLAY_VERTICES_H
#define MY_DISPLAY_VERTICES_H

#include <McHandle.h> // smart pointer template class
#include <hxcore/HxModule.h>
#include <hxcore/HxPortIntSlider.h>
#include <mypackage/mypackageAPI.h>

#include <Inventor/nodes/SoSeparator.h>

class MYPACKAGE_API MyDisplayVertices1 : public HxModule
{
    HX_HEADER(MyDisplayVertices1);

  public:
    // Constructor.
    MyDisplayVertices1();

    // Destructor.
    ~MyDisplayVertices1();

    // Input parameter.
    HxPortIntSlider portNumTriangles;

    // This is called when an input port changes.
    virtual void compute();

  protected:
    McHandle<SoSeparator> scene;
};

#endif
```

The header file can be understood quite easily. First some other header files are included. Then the new module is declared as a child class of `HxModule`. As usual, the macros `MYPACKAGE_API` and `HX_HEADER` are obligatory. Our module implements a default constructor, a destructor, and a compute method. In addition, it has a port of type `HxPortIntSlider` which allows the user to specify the number of triangles of the vertices to be displayed.
A pointer to the actual Open Inventor scene is stored in the member variable `scene` of type `McHandle<SoSeparator>`. A `McHandle` is a so-called *smart pointer*. It can be used like an ordinary C pointer. However, each time a

value is assigned to it, the reference counter of the referenced object is automatically increased or decreased. This is done by calling the `ref` or `unref` method of the object. If the reference counter becomes zero or less, the object is deleted automatically. We recommend using smart pointers instead of C pointers because they are safer.
The actual implementation of the module is contained in the file `MyComputeThreshold1.cpp`. This file looks as follows:

```
/////////////////////////////////////////////////////////////////
//
// Example of a compute module (version 1)
//
/////////////////////////////////////////////////////////////////

#include <hxcore/HxMessage.h>
#include <hxsurface/HxSurface.h>
#include <mypackage/MyDisplayVertices1.h>

#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoTranslation.h>

HX_INIT_CLASS(MyDisplayVertices1,HxModule)

MyDisplayVertices1::MyDisplayVertices1() :
    HxModule(HxSurface::getClassTypeId()),
    portNumTriangles(this,"numTriangles")
{
    portNumTriangles.setMinMax(1,12);
    portNumTriangles.setValue(6);
    scene = new SoSeparator;
}

MyDisplayVertices1::~MyDisplayVertices1()
{
    hideGeom(scene);
}

void MyDisplayVertices1::compute()
{
    int i;

    // Access input object (portData is inherited from HxModule):
    HxSurface* surface = (HxSurface*) portData.source();

    if (!surface) { // Check if input object is available
        hideGeom(scene);
```

```
        return;
    }

    // Get value from input port, query size of surface:
    int numTriPerVertex = portNumTriangles.getValue();
    int nVertices = surface->points.size();
    int nTriangles = surface->triangles.size();

    // We need a triangle counter for every vertex:
    McDArray<unsigned short> triCount(nVertices);
    triCount.fill(0);

    // Loop over all triangles and increase vertex counters:
    for (i=0; i<nTriangles; i++)
        for (int j=0; j<3; j++)
            triCount[surface->triangles[i].points[j]]++;

    // Now create the scene graph...
    // First remove all previous childs:
    scene->removeAllChildren();

    // Cube size should be 1% of the bounding box diagonal:
    float size = surface->getBoundingBoxSize().length() * 0.01;

    // Pointer to coordinates cast from McVec3f to SbVec3f.
    SbVec3f* p = (SbVec3f*) surface->points.dataPtr();

    SbVec3f q(0,0,0); // position of last point
    int count = 0; // vertex counter

    for (i=0; i<nVertices; i++) {
        if (triCount[i] == numTriPerVertex) {
            SoTranslation* trans = new SoTranslation;
            trans->translation.setValue(p[i]-q);

            SoCube* cube = new SoCube;
            cube->width = cube->height = cube->depth = size;

            scene->addChild(trans);
            scene->addChild(cube);

            count++;
            q=p[i];
        }
    }

    theMsg->printf("Found %d vertices belonging to %d triangles",
```

```
        count, numTriPerVertex);

    showGeom(scene); // finally show scene in viewer
}
```

A lot of things are happening here. Let us point out some of these in more detail now. The constructor initializes the base class with the type returned by `HxSurface::getClassTypeId`. This ensures that the module can only be attached to data objects of type `HxSurface`. The constructor also initializes the member variable `portNumTriangles`. The range of the slider is set from 1 to 12. The initial value is set to 6. Finally, a new Open Inventor separator nodes is created and stored in `scene`.
The destructor contains only one call, `hideGeom(scene)`. This causes the Open Inventor scene to be removed from all viewers (provided it is visible). The scene itself is deleted automatically when the destructor of `McHandle` is called.
The actual computation is performed in the `compute` method. The method returns immediately if no input surface is present. If an input surface exists, the numbers of triangles per point are counted. For this purpose a dynamic array `triCount` is defined. The array provides a counter for each vertex. Initially it is filled with zeros. The counters are increased in a loop over the vertices of all triangles.
In the second part of the compute method the Open Inventor scene graph is created. First, all previous children of `scene` are removed. Then the length of the diagonal of the input surface is determined. The size of the cubes will be set proportional to this length. For convenience the pointer to the coordinates of the surface is stored in a local variable `p`. Actually the coordinates are of type `McVec3f`. However, this class is fully compatible with the Open Inventor vector class `SbVec3f`. Therefore the pointer to the coordinates can be cast as shown in the code.
After everything has been set up, every element of the array `triCount` is checked in a for-loop. If the value of an element matches the selected number of triangles per vertex, two new Inventor nodes of type `SoTranslation` and `SoCube` are created, initialized, and inserted into `scene`. Since the `SoTranslation` also affects all subsequent translation nodes we must remember the position of the last point in `q` and subtract this position from the one of the current point. Alternatively, we could have encapsulated the `SoTranslation` and the `SoCube` in an additional `SoSeparator` node. However, this would have resulted in a more complex scene graph. At the very end of the compute method, the new scene graph is made visible in the viewer by calling `showGeom`. This method automatically checks if a node has already been visible. Therefore it may be called multiple

times with the same argument.
The module is registered in the usual way in the package resource file, i.e., in `mypackage/share/resource/mypackage.rc`. Once you have compiled the demo package, you may test the module by loading the surface `mypackage/data/test.surf` located in the local Amira directory.

4.2.2 Version 2: Adding Color and a Parse Method

In this section we want to add two more features to our module. First, we want to use a colormap port which allows us to specify the color of the cubes. Second, we want to add a parse method which allows us to specify additional Tcl commands for the module.
A colormap port is used to establish a connection to a colormap, i.e., to a class of type `HxColormap`. It is derived from `HxConnection` but, in contrast to the base class, it provides a graphical user interface showing the contents of the colormap and letting the user change its coordinate range. If no colormap is connected to the port, a default color is displayed. The default color can be edited by the user by double-clicking the color bar.
In order to provide our module with a colormap port, we must insert the following line into the module's header file:

```
HxPortColormap portColormap;
```

Of course, we must also include the header file of the class `HxPortColormap`. This file is located in package `hxcolor`. Note that the order in which ports are displayed on the screen depends on the order in which the ports are declared in the header file. If we declare `portColormap` before `portNumTriangles`, the colormap port will be displayed before the integer slider.
In the compute method of our module we add the following piece of code just after the previous children of the scene graph have been removed:

```
SoMaterial* material = new SoMaterial;
material->diffuseColor =
    portColormap.getColor(numTriPerVertex);
scene->addChild(material);
```

With these lines we insert a material node right before all the translation and cube nodes into the separator. The material node causes the cubes to be displayed in a certain color. We call the `getColor` method of the colormap port in order to determine this color. If the port is not connected to a colormap, this method simply returns the default color. However, if it is connected, the color is taken from the colormap. As an argument we specify `numTriPerVertex`, the number of triangles of the selected vertices. Depending on the value of `portNumTriangles`, the cubes therefore will be displayed in different colors. Of course, this requires that the range of the colormap extend from something like 1 to 10 or 12.

Besides the colormap port, we also want to add a Tcl command interface to our module. This is done by overloading the virtual method `parse` of `HxModule`. We therefore insert the following line into the module's class declaration:

```
virtual int parse(Tcl_Interp* t, int argc, char **argv);
```

In a parse method special commands can be defined which allow us to control the module in a more sophisticated way. A typical application is to set special parameters which should not be represented by a separate port in the user interface. As an example, we want to provide a method which allows us to change the size of the cubes. In the initial version of the module the cubes were adjusted so that each side was 0.01 times the length of the diagonal of the bounding box of the input surface. The value of the scale factor shall now be stored in the member variable `scale`. In order to set and get this variable, two Tcl commands `setScale` and `getScale` shall be provided. The implementation of the parse method looks as follows:

```
int
MyDisplayVertices2::parse(Tcl_Interp* t, int argc, char **argv)
{
    if (argc < 2) return TCL_OK;
    char *cmd = argv[1];

    if (CMD("setScale")) {
        ASSERTARG(3);
        scale = atof(argv[2]);
        fire(); // ensures that cubes will be updated immediately
    }
    else if (CMD("getScale")) {
        Tcl_VaSetResult(t, "%g", scale);
    }
    else return HxModule::parse(t,argc,argv);

    return TCL_OK;
}
```

Commands are defined in a sequence of if-else statements. For each command, the macro `CMD` should be used. At the end of the if-else sequence the parse method of the base class should be called. Note that after a command is issued, the compute method of the module will not be called automatically by default. This is in contrast to interactive changes of ports. However, we may explicitly call `fire` in a command like shown above. In this case the size of the cubes then will be adjusted immediately. You may test the parse method by loading the file `mypackage/data/test.surf`, attaching `DisplayVertices2` to it, and then typing something like `DisplayVertices2 setScale 0.03` into the Amira console window.

4.2.3 Version 3: Adding an Update Method

Besides a compute method, modules may also define a *update method*. This method is called just before the compute method and also whenever a module is selected. In the update method, the user-interface of the module can be configured, i.e., ports can be shown or hidden dynamically if this is required, the sensitivity of ports can be adjusted, or the number of entries of an option menu can be modified dynamically.

In order to illustrate how an update method might work, we implement an alternate display mode in our module. In this mode all vertices of a surface should be displayed, not only the ones with a certain number of neighboring triangles. In this second mode the slider `portNumTriangles` is not meaningful anymore. We therefore hide it by defining an appropriate update method. The following lines are added in the header file `MyDisplayVertices3.h`:

```
    // Mode: 0=selected vertices, 1=all vertices
    HxPortRadioBox portMode;

    // Shows or hides required ports.
    virtual void update();
```

The new radio box port lets the user switch between the two display modes. Like the compute method, the update method takes no arguments and also has no return value.

If you look into the source code file `MyDisplayVertices3.cpp` you will notice that the radio box port is initialized in the constructor of the module and that the text labels are set properly. The update method itself is quite simple:

```
void MyDisplayVertices3::update()
{
    if (portMode.getValue()==0)
```

```
        portNumTriangles.show();
    else portNumTriangles.hide();
}
```

The slider `portNumTriangles` is shown or hidden depending on the value of the radio box port. Note that before the update method is called, all ports are marked to be shown. Therefore you must hide them every time `update` is called. For example, the `show` and `hide` calls should not be encapsulated by an if statement which checks if some input port is new.
In order to support the new all-vertices display style, we slightly modify the way the Open Inventor scene graph is created. Instead of a single `SoMaterial` node, we insert a new one whenever the color of a cube needs to be changed, i.e., whenever the number of triangles of a vertex differs from the previous one. The new for-loop looks as follows:

```
    int lastNumTriPerVertex = -1;
    int allVertices = portMode.getValue();

    for (i=0; i<nVertices; i++) {
        if (allVertices || triCount[i]==numTriPerVertex) {

            if (triCount[i]!=lastNumTriPerVertex) {
                SoMaterial* material = new SoMaterial;
                material->diffuseColor =
                    portColormap.getColor(triCount[i]);
                scene->addChild(material);
                lastNumTriPerVertex = triCount[i];
            }

            SoTranslation* trans = new SoTranslation;
            trans->translation.setValue(p[i]-q);

            SoCube* cube = new SoCube;
            cube->width = cube->height = cube->depth = size;

            scene->addChild(trans);
            scene->addChild(cube);

            count++;
            q=p[i];
        }
    }
```

Again, you can test the module by loading the file `mypackage/data/test.surf` and attaching `DisplayVertices3` to

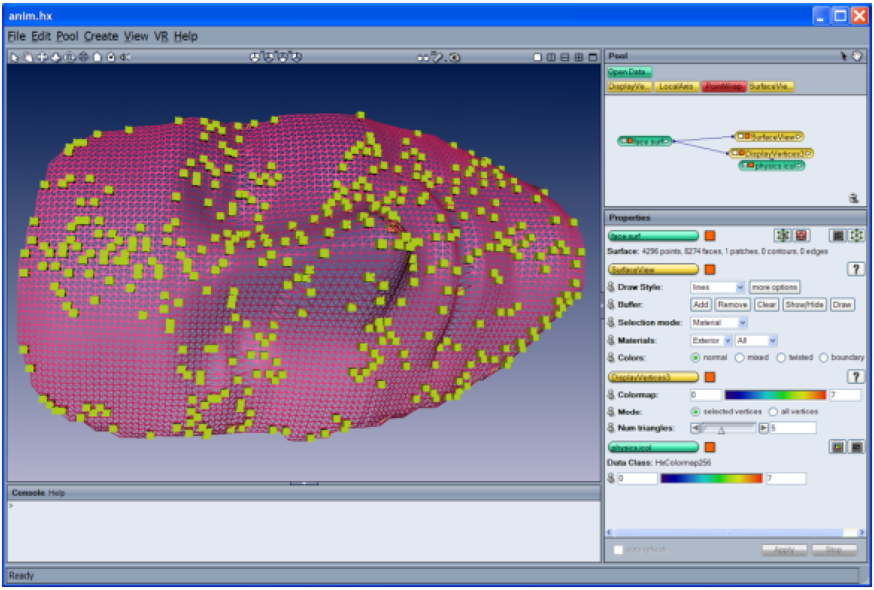

Figure 4.1: The demo module *DisplayVertices3* displays little cubes at the vertices of a surface. The cubes are colored according to the number of neighboring triangles.

it. If you connect the *physics.icol* colormap to the colormap port, adjust the colormap range to 1...9, and select the all-vertices display style, you should get an image similar to the one shown in Figure 4.1.

4.3 A Module With Plot Output

In some cases you may want to show a simple 2D plot in an Amira module, for example a histogram or some bar chart. To facilitate this task Amira provides a special-purpose *Plot API* which can be used in any Amira object, regardless of whether it is a compute module or a display module.

The class `PzEasyPlot` provides the necessary methods to open a plot window and to draw in that window. In the following, we illustrate how to use this class, again by means of an example. In particular, we are going to write a module which plots the number of voxels per slice for all materials defined in a label field. A label field usually represents the results of an image segmentation operation. For each voxel there is a label indicating which material the voxel belongs to. In a separate section *further features of the Plot API* will be described.

4.3.1 A Simple Plot Example

In this section we show how to plot some simple curves using the class `PzEasyPlot`. As mentioned above, the curves represent the number of voxels per slice for the materials of a label field. For this purpose we define a new module called `MyPlotAreaPerSlice`.
Like the other examples, this module is contained in the Amira demo package. In order to check out the demo package, you must create a local Amira directory as described in Section 2.2. In order to compile the demo package, please refer to Section 1.5 (Compiling and Debugging).
Let us first look at the header file `MyPlotAreaPerSlice.h`:

```
/////////////////////////////////////////////////////////////////
//
// Example of a plot module (header file)
//
/////////////////////////////////////////////////////////////////
#ifndef MY_PLOT_AREA_PER_SLICE_H
#define MY_PLOT_AREA_PER_SLICE_H

#include <hxcore/HxModule.h>
#include <hxcore/HxPortButtonList.h>
#include <hxplot/PzEasyPlot.h> // simple plot window
#include <mypackage/mypackageAPI.h>

class MYPACKAGE_API MyPlotAreaPerSlice : public HxModule
{
    HX_HEADER(MyPlotAreaPerSlice);

  public:
    // Constructor.
    MyPlotAreaPerSlice();

    // Shows the plot window.
    HxPortButtonList portAction;

    // Performs the actual computation.
    virtual void compute();
```

```
  protected:
    McHandle<PzEasyPlot> plot;
};

#endif
```

The class declaration is very simple. The module is derived directly from `HxModule`. It provides a constructor, a compute method, and a port of type `HxPortButtonList`. In fact, we will only use a single push button in order to let the user pop up the plot window. The plot window class `PzEasyPlot` itself is referenced by a smart pointer, i.e., by a variable of type `McHandle<PzEasyPlot>`. We have already used smart pointers in Section 4.2.1, for details see there.
Now let us take a look at the source file `MyPlotAreaPerSlice.cpp`:

```
/////////////////////////////////////////////////////////////////
//
// Example of a plot module (source code)
//
/////////////////////////////////////////////////////////////////

#include <hxcore/HxWorkArea.h>
#include <hxfield/HxLabelLattice3.h>
#include <mypackage/MyPlotAreaPerSlice.h>

HX_INIT_CLASS(MyPlotAreaPerSlice,HxModule)

MyPlotAreaPerSlice::MyPlotAreaPerSlice() :
    HxModule(HxLabelLattice3::getClassTypeId()),
    portAction(this,"action",1)
{
    portAction.setLabel(0,"Show Plot");
    plot = new PzEasyPlot("Area per slice");
    plot->autoUpdate(0);
}

void MyPlotAreaPerSlice::compute()
{
    HxLabelLattice3* lattice = (HxLabelLattice3*)
        portData.source(HxLabelLattice3::getClassTypeId());

    // Check if valid input is available.
    if (!lattice) {
        plot->hide();
        return;
```

```
    }

    // Return if plot window is invisible and show button
    // wasn't hit
    if (!plot->isVisible() && !portAction.isNew())
        return;

    theWorkArea->busy(); // activate busy cursor

    int i,k,n;
    const int* dims = lattice->dims();
    unsigned char* data = lattice->getLabels();
    int nMaterials = lattice->materials()->nBundles();

    // One counter per material and slice
    McDArray< McDArray<float> > count(nMaterials);

    for (n=0; n<nMaterials; n++) {
        count[n].resize(dims[2]);
        count[n].fill(0);
    }

    // Count number of voxels per material and slice
    for (k=0; k<dims[2]; k++) {
        for (i=0; i<dims[1]*dims[0]; i++) {
            int label = data[k*dims[0]*dims[1]+i];
            if (label<nMaterials)
                count[label][k]++;
        }
    }

    plot->remData(); // remove old curves

    for (n=0; n<nMaterials; n++) // add new curves
        plot->putData(lattice->materials()->bundle(n)->name(),
            dims[2], count[n].dataPtr());

    plot->update(); // refresh display
    plot->show(); // show or raise plot window

    theWorkArea->notBusy(); // deactivate busy cursor
}
```

In the constructor the base class `HxModule` is initialized with the class type ID of the class `HxLabelLattice3`. This class is not a data class derived from `HxData` but a so-called *interface*. Interfaces are used to provide a common API for objects not directly related by inheritance. In our case,

MyPlotAreaPerSlice can be connected to any data object providing a HxLabelLattice3 interface. This might be a HxUniformLabelField3 but also a HxStackedLabelField3 or something else.
Also in the constructor, a new plot window of type PzEasyPlot is created and stored in plot. Then the method plot->autoUpdate(0) is called. This means that we must explicitly call the update method of PzEasyPlot after the contents of the plot window are changed. Auto-update should be disabled when more than one curve is being changed at once.
As usual, the actual work is performed by the compute method. First, we retrieve a pointer to the label lattice. Since we want to use an interface instead of a data object itself, we must specify the class type ID of the interface as an argument of the source method of portData. Otherwise we would get a pointer to the object providing the interface, but we can't be sure about the type of this object.
The method returns if no label lattice is present or if the plot window is not visible and the show button has not been pressed. Otherwise, the contents of the plot window are recomputed from scratch. For this purpose a dynamic array of arrays called count is defined. The array provides a counter for each material and for each slice of the label lattice. Initially all counters are set to zero. Afterwards, they are incremented while the voxels are traversed in a nested for-loop.
The actual initialization of the plot window happens subsequently. First, old curves are removed by calling plot->remData. Then, for each material, a new curve is added by calling plot->putData. Afterwards, plot->update is called. If we had not disabled 'auto update' in the constructor, the plot window would have been updated automatically in each call of putData. The putData method creates a curve with the given name and sets the values. If a curve of the given name exists, the old values are overridden. The method returns a pointer to the curve which in turn can be used to set attributes for the curve individually (see below). Finally, the plot window is popped up and the 'busy' cursor we have activated before is switched off again.
To test the module, first compile the demo package. For instructions, see Section 1.5 (Compiling and Debugging). Then load the file data/tutorials/lobus.labels.am from the Amira root directory. Attach PlotAreaPerSlice to it and press the show button. You then should get a result like that shown in Figure 4.2.

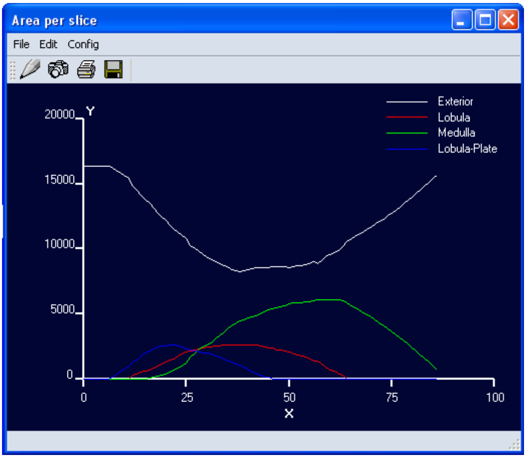

Figure 4.2: Plot produced by sample module *PlotAreaPerSlice*.

4.3.2 Additional Features of the Plot API

The 'pointer to curve' objects returned by the `putData` call can be used to access the curve directly, i.e., to manipulate its attributes. The most important attributes of curve objects are:

- Color, represented by a RGB values between 0 and 1. Can be set by calling: `curve->setAttr("color", r, g, b);`
- Line width, represented by an integer number. Can be set by calling: `curve->setAttr("linewidth", linewidth);`
- Line type, represented by an integer number. Available line types are 0=no line, 1=line, 2=dashed, 3=dash-dotted, and 4=dotted. Can be set by calling: `curve->setAttr("linetype", type);`
- Curve type, represented by an integer number. Available curve types are 0=line curve, 1=histogram, 2=marked line, 3=marker. Can be set by calling: `curve->setAttr("curvetype", type);`

For each attribute corresponding `getAttr` methods are available. In order to access the axis of the 'easy plot' window, you must call

```
PzAxis* axis = plot->getTheAxis();
```

Don't forget to include the corresponding header file `PzAxis.h`.
The color, line width, and line type attributes of the curves apply to axes as well. Besides this, there are some more methods to change the appearance of axes:

```
// Set the range of the axes
float xmin = 0.0, xmax = 1.0;
float ymin = 0.0, ymax = 1.0;
axis->setMinMax(xmin, xmax, ymin, ymax);

// Set the label of an axis
axis->setLabel(0, "X Axis");
axis->setLabel(1, "Y Axis");
```

If you are not satisfied with the size of the plot window and you don't want to change it using the mouse every time, just call `setSize` right after creating the plot window:

```
plot->setSize(width, height);
```

As you would expect, the methods `getMinMax`, `getLabel` and `getSize` are also available with the same parameter list as their `set` counterparts.
Finally, it is also possible to have a legend or a grid in the plot. In this case more arguments must be specified in the constructor of `PzEasyPlot`:

```
int withLegend = 1;
int withGrid = 0;
plot = new PzEasyPlot("Area per slice",
   withLegend, withGrid);
```

Like the axis, the legend and the grid are internally represented by separate objects of type `PzLegend` and `PzGrid`. You can access these objects by calling the methods `getTheLegend` and `getTheGrid`. Details about the member methods of these objects are listed in the class reference documentation.

5 Data Classes

This chapter provides an overview of the structure of Amira data classes. Important classes are discussed in more detail. In particular, the following topics will be covered:

- *an introduction to data classes*, including the hierarchy of data classes
- *data on regular grids*, e.g., 3D images with uniform or stacked coords
- *tetrahedral grids*, including data fields defined on such grids
- *hexahedral grids*, including data fields defined on such grids
- *other issues related to data classes*, including transparent data access

5.1 Introduction

A profound knowledge of the Amira data objects is essential to developers. Data objects occur as input of write routines and almost all modules, and as output of read routines and compute modules. In the previous chapters we already encountered several examples of Amira data objects such as 3D image data (represented by the class `HxUniformScalarField3`), triangular surfaces (represented by the class `HxSurface`), or colormaps (represented by the class `HxColormap`). Like modules, data objects are instances of C++ classes. All data objects are derived from the common base class `HxData`. Data objects are represented by green icons in Amira's Pool.
In the following let us first present an overview of the *hierarchy of data classes*. Afterwards, we will discuss some of the *general concepts* behind it.

5.1.1 The Hierarchy of Data Classes

The hierarchy of Amira data classes roughly looks as follows (derived classes are indented, auxiliary base classes are ignored):

`HxData` *base class of all data objects*

`HxSpreadSheet` *spreadsheet containing an arbitrary number of rows and columns*

`HxColormap` *base class of colormaps*
`HxColormap256` *colormap consisting of discrete RGBA tuples*
`HxCameraPath` *base class of camera paths*
`HxKeyframeCameraPath` *camera path based on interpolated keyframes*
`HxSpatialData` *data objects embedded in 3D space*
`HxIvData` *encapsulates an Open Inventor scene graph*
`HxField3` *base class representing fields in 3D space*
`HxScalarField3` *scalar field (1 component)*
`HxRegScalarField3` *scalar field with regular coordinates*
`HxUniformScalarField3` *scalar field with uniform coordinates*
`HxUniformLabelField3` *material labels with uniform coordinates*
`HxStackedScalarField3` *scalar field with stacked coordinates*
`HxStackedLabelField3` *material labels with stacked coordinates*
`HxAnnaScalarField3` *scalar field defined by an analytic expression*
`HxTetraScalarField3` *scalar field defined on a tetrahedral grid*
`HxHexaScalarField3` *scalar field defined on a hexahedral grid*
`HxVectorField3` *vector field (3 components)*
`HxRegVectorField3` *vector field with regular coordinates*
`HxUniformVectorField3` *vector field with uniform coordinates*
`HxEdgeElemVectorField3` *vector field defined by Whitney elements*
`HxAnnaVectorField3` *vector field defined by an analytic expression*
`HxTetraVectorField3` *vector field defined on a tetrahedral grid*
`HxHexaVectorField3` *vector field defined on a hexahedral grid*
`HxComplexScalarField3` *complex-valued scalar field (2 components)*
`HxRegComplexScalarField3` *complex scalar field with regular coordinates*
`HxUniformComplexScalarField3` *complex scalar field w/ uniform coords*
`HxTetraComplexScalarField3` *complex scalar field defined on a tetra grid*
`HxHexaComplexScalarField3` *complex scalar field defined on a hexa grid*
`HxComplexVectorField3` *complex-valued vector field (6 components)*
`HxRegComplexVectorField3` *complex vector field with regular coordinates*
`HxUniformComplexVectorField3` *complex vector field w/ uniform coords*
`HxEdgeElemComplexVectorField3` *complex vector field w/ Whitney elements*
`HxTetraComplexVectorField3` *complex field defined on a tetrahedral grid*
`HxHexaComplexVectorField3` *complex field defined on a hexahedral grid*
`HxColorField3` *color field consisting of RGBA-tuples*
`HxRegColorField3` *color field with regular coordinates*
`HxUniformColorField3` *color field with uniform coordinates*
`HxRegField3` *other n-component field with regular coordinates*

`HxTetraField3` *other n-component field defined on a tetrahedral grid*
`HxHexaField3` *other n-component field defined on a hexahedral grid*
`HxVertexSet` *data objects providing a set of discrete vertices*
`HxSurface` *represents a triangular surface*
`HxTetraGrid` *represents a tetrahedral grid*
`HxHexaGrid` *represents a hexahedral grid*
`HxLineSet` *represents a set of line segments with vertex data*
`HxLandmarkSet` *represents one or multiple sets of corresponding landmarks*
`HxCluster` *represents a set of vertices with associated data values*
`HxSurfaceField` *base class for fields defined on triangular surfaces*
`HxSurfaceScalarField` *scalar field defined on a surface (1 component)*
`HxSurfaceVectorField` *vector field defined on a surface (3 components)*
`HxSurfaceComplexScalarField` *complex scalar surface field (2 components)*
`HxSurfaceComplexVectorField` *complex vector surface field (6 components)*
`HxSurfaceField` *other n-component field defined on a surface*

Note that you can find an in-depth description of every class in the online reference documentation. This description has been generated automatically from the commented Amira header files by a tool called DOC++. You may view it by pointing an external web browser such as Internet Explorer or Netscape Navigator to the file `share/doc++/index.html` contained in the Amira root directory. The reference documentation not only covers data objects but all classes provided with Developer Option. As you already know, these classes are arranged in packages. For example, all data classes derived from `HxField3` are located in package `hxfield`, and all classes related to triangular surfaces are located in package `hxsurface`.

5.1.2 Remarks About the Class Hierarchy

All data classes are derived from the base class `HxData`. This class in turn is derived from `HxObject`, the base class of all objects that can be put into the Amira Pool. The class `HxData` adds support for reading and writing data objects, and it provides the variable `parameters` of type `HxParamBundle`. This variable can be used to annotate a data object by an arbitrary number of nested parameters. The parameters of any data object can be edited interactively using the parameter editor described in the User's Guide.
We observe that the majority of data classes are derived from `HxSpatialData`. This is the base class of all data objects which are embedded in 3D space as op-

posed for example to colormaps. `HxSpatialData` adds support for user-defined affine transformations, i.e., translations, rotations, and scaling. For details refer to Section 5.5.2. It also provides the virtual method `getBoundingBox` which is redefined by all derived classes. Two important child classes of `HxSpatialData` are `HxField3` and `HxVertexSet`.

`HxVertexSet` is the base class of all data objects that are defined on an unstructured set of vertices in 3D space, like surfaces or tetrahedral grids. The class provides methods to apply a user-defined affine transformation to all vertices of the object, or modify the point coordinates in some other way.

`HxField3` is the base class of data fields defined on a 3D-domain, like 3D scalar fields or 3D vector fields. `HxField3` defines an efficient procedural interface to evaluate the field at an arbitrary 3D point within the domain, independent of whether the latter is a regular grid, a tetrahedral grid, or something else. The procedural interface is described in more detail in Section 5.5.1.

Looking at the inheritance hierarchy again, we observe that a high level distinction is made between fields returning a different number of data values. For example, all 3D scalar fields are derived from a common base class `HxScalarField3`, and all 3D vector fields are derived from a common base class `HxVectorField3`. The reason for this structure is that many modules depend on the data dimensionality of a field only, not on the internal representation of the data. For example, a module for visualizing a flow field by means of particle tracing can be written to accept any object of type `HxVectorField3` as input. It then automatically operates on all derived vector fields, regardless of the type of grid they are defined on.

On the other hand, it is often useful to treat the number of data variables of a field as a dynamic quantity and to distinguish between the type of grid a field is defined on. For example, we may wish to have a common base class of fields defined on a regular grid and derived classes for regular scalar or vector fields. Since this structure and the one sketched above are very hard to incorporate into a common class graph, even if multiple inheritance were used, another concept has been chosen in Amira, namely *interfaces*. Interfaces were first introduced by the Java programming language. They allow the programmer to take advantage of common properties of classes that are not related by inheritance.

In Amira interfaces can be implemented as class members, or as additional base classes. In the first case a data class *contains* an interface class, while in the second case it is *derived* from `HxInterface`. Important interface classes are `HxLattice3`, `HxTetraData`, and `HxHexaData`, which are members of fields defined on regular, tetrahedral, and hexahedral grids, respectively. Another example is HxLabelLattice3, which is a member of HxUniformLabelField3, as well as HxStackedLabelField3. In Section 4.3.1 we have already presented an example of how to use this interface in order to write a module which operates on any label field, regardless of the actual coordinate type.

5.2 Data on Regular Grids

Fields defined on a regular grid occur in many different applications. For example, 3D image volumes fall into this category. The term 'regular' means that the nodes of the grid are arranged as a regular 3D array, i.e., every node can be addressed by an index triple (i,j,k). A regular field can be characterized by three major properties: the coordinate type, the number of data components, and the primitive component data type (for example short or float).
In the *class hierarchy* a major distinction is made between the number of data components of a field. For example, there is a class HxRegScalarField3 representing (one-component) scalar fields defined on a regular grid. This class is derived from the general base class HxScalarField3. Similar classes exist for (three-component) vector fields, complex scalar field, and complex vector fields defined on regular grids. Fields not falling into one of these categories, i.e., fields defined on regular grids with a different number of data components, are represented by the class HxRegField3 which is directly derived from HxField3. Moreover, there are separate subclasses for the most relevant combinations of the number of data components and the coordinates type, like HxStackedScalarField3 or HxUniformVectorField3. All regular data classes provide a member variable lattice of type HxLattice3. This variable is an *interface*. It can be used to access data fields with a different number of components in a transparent way.
Below we first discuss the *lattice interface* in more detail. We then present an overview of all supported *coordinate types*. Afterwards, two more types of data fields defined on regular coordinates are discussed, namely *label fields* and *color fields*.
Note that all these fields can be evaluated without regard to the actual coordinate type or the primitive data type by means of Amira's procedural interface for 3D fields (see Section 5.5.1).

5.2.1 The Lattice Interface

The actual data of any regular 3D field is stored in a member variable `lattice` of type `HxLattice3`. This variable essentially represents a dynamic 3D array of n-component vectors. The number of vector components as well as the primitive data type are subject to change, i.e., a data object of type `HxLattice3` can be re-initialized to hold a different number of components of different primitive data type. However, a lattice contained in an object of type `HxRegScalarField3` always consists of 1-component vectors, while a lattice contained in an object of type `HxRegVectorField3` always consists of 3-component vectors. In addition, the coordinates of the field are stored in a separate coordinate object that is also referenced by the lattice.

Accessing the Data

To learn what kind of methods are provided by the lattice class, please refer to the online reference documentation or directly inspect the header file `HxLattice3.h` located in package `hxfield`. At this point, we just present a short example which shows how the dimensionality of the lattice, the number of data components, and the primitive data type can be queried. The primitive data type is encoded by the class `McPrimType` defined in package `mclib`. In particular, the following six data types are supported by Amira:

- `McPrimType::mc_uint8` (8-bit unsigned bytes)
- `McPrimType::mc_int16` (16-bit signed shorts)
- `McPrimType::mc_uint16` (16-bit unsigned shorts)
- `McPrimType::mc_int32` (32-bit signed integers)
- `McPrimType::mc_float` (32-bit floats)
- `McPrimType::mc_double` (64-bit doubles)

Regardless of the actual type of the lattice data values, the pointer to the data array is returned as `void*`. The return value must be explicitly cast to a pointer of the correct type. This is illustrated in the following example where we compute the maximum value of all data components of a lattice. Note that the data values are stored one after another without any padding. The first index runs fastest.

```
HxLattice3& lattice = field->lattice;
const int* dims = lattice.dims();
int nDataVar = lattice.nDataVar();
```

```
    switch (lattice.primType()) {
    case McPrimType::mc_uint8: {
        unsigned char* data = (unsigned char*) lattice.dataPtr();
        unsigned char max = data[0];
        for (int k=0; k<dims[2]; k++)
            for (int j=0; j<dims[1]; j++)
                for (int i=0; i<dims[0]; i++)
                    for (int n=0; n<nDataVar; n++) {
                        int idx =
                            nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                        if (data[idx]>max)
                            max = data[idx];
                    }
        theMsg->printf("Max value is %d", max);
        } break;

    case McPrimType::mc_int16: {
        short* data = (short*) lattice.dataPtr();
        short max = data[0];
        for (int k=0; k<dims[2]; k++)
            for (int j=0; j<dims[1]; j++)
                for (int i=0; i<dims[0]; i++)
                    for (int n=0; n<nDataVar; n++) {
                        int idx =
                            nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                        if (data[idx]>max)
                            max = data[idx];
                    }
        theMsg->printf("Max value is %d", max);
        } break;

    ...

    }
```

As a tip, note that the processing of different primitive data types can often be simplified by defining appropriate template functions locally. In the case of our example, such a template function may look like this:

```
template<class T>
void getmax(T* data, const int* dims, int nDataVar)
{
    T max = data[0];
    for (int k=0; k<dims[2]; k++)
        for (int j=0; j<dims[1]; j++)
            for (int i=0; i<dims[0]; i++)
                for (int n=0; n<nDataVar; n++) {
```

```
                    int idx =
                        nDataVar*((k*dims[1]+j)*dims[0]+i)+n;
                    if (data[idx]>max)
                        max = data[idx];
                }
    theMsg->printf("Max value is %d", max);
}
```

Using this template function, the above switch statement looks as follows:

```
    switch (lattice.primType()) {
    case McPrimType::mc_uint8:
        getmax((unsigned char*)lattice.dataPtr(),dims,nDataVar);
        break;
    case McPrimType::mc_int16:
        getmax((short*)lattice.dataPtr(),dims,nDataVar);
        break;

    ...

    }
```

Though less efficient, another possibility for handling different primitive data types is to use one of the methods `eval`, `set`, `getData`, or `putData`. These methods always involve a cast to `float` if the primitive data type of the field requires it.

Accessing the Lattice Interface

Imagine you want to write a module which operates on any kind of regular field, i.e., on objects of type `HxRegScalarField3`, `HxRegVectorField3`, and so on. One way to achieve this would be to configure the input port of the module so that it can be connected to all possible regular field input objects. This can be done by calling the method `portData.addType()` in the module's constructor multiple times with the required class type IDs. In addition, all input types must be listed in the package resource file. This can be done by specifying a blank-separated list of types as the argument of the `-primary` option of the `module` command. In the compute method of the module, the actual type of the input must be queried, then the input pointer must be cast to the required type before a pointer to the lattice member of the object can be stored.
Of course, this approach is very tedious. A much simpler approach is to make use of the fact that the lattice member of a regular field is an interface. Instead of the name of a real data class, the class type ID of `HxLattice3` may be used

to specify to what kind of input object a module may be connected to. In fact, if this is done, any data object providing the lattice interface will be considered as a valid input. In order to access the lattice interface of the input object, the following statement must be used in the module's compute method (also check Section 4.3.1 for an example of how to deal with interfaces):

```
HxLattice3* lattice = (HxLattice3*)
    portData.source(HxLattice3::getClassTypeId());
```

Creating a Field From an Existing Lattice

When working with lattices, we may want to deposit a new lattice in the Pool, for example as the result of a compute module. However, since `HxLattice3` is not an Amira data class, this is not possible. Instead we must create a suitable field object which the lattice is a member of. For this purpose the class `HxLattice3` provides a static method `create` which creates a regular field and puts an existing lattice into it. If the lattice contains one data component, a scalar field will be created; if it contains three components, a vector field will be created, and so on. The resulting field may then be used as the result of a compute module. Note that the lattice must not be deleted once it has been put into a field object. The concept is illustrated by the following example:

```
HxLattice3* lattice = new HxLattice3(dims, nDataVar,
    primType, otherLattice->coords()->duplicate());

...

HxField3* field = HxLattice3::create(lattice);
theObjectPool->addObject(field);
```

5.2.2 Regular Coordinate Types

Currently four different coordinate types are supported for regular fields, namely uniform coordinates, stacked coordinates, rectilinear coordinates, and curvilinear coordinates. The coordinate types are distinguished by way of the enumeration data type `HxCoordType`. The coordinates themselves are stored in a separate utility class of type `HxCoord3` which is referenced by the lattice member of a regular field. For each coordinate type there is a corresponding subclass of `HxCoord3`.
As already mentioned in the introduction, for some important cases there are special subclasses of a regular field dedicated to a particular coordinate type. Exam-

ples are HxStackedScalarField3 (derived from HxRegScalarField3) or HxUniformVectorField3) (derived from HxRegVectorField3). If such special classes do not exist, the regular base class should be used instead. In this case the coordinate type must be checked dynamically and the pointer to the coordinate object has to be down-cast explicitly before it can be used. This is illustrated in the following example:

```
HxCoord3* coord = field->lattice.coords();

if (coord->coordType() == c_rectilinear) {
    HxRectilinearCoord3* rectcoord =
        (HxRectilinearCoord3*) coord;
    ...
}
```

Uniform Coordinates

Uniform coordinates are the simplest form of regular coordinates. All grid cells are axis-aligned and of equal size. In order to compute the position of a particular grid node, it is sufficient to know the number of cells in each direction as well as the bounding box of the grid.
Uniform coordinates are represented by the class HxUniformCoord3. This class provides a method bbox which returns a pointer to an array of six floats describing the bounding box of the grid. The six numbers represent the minimum x-value, the maximum x-value, the minimum y-value, the maximum y-value, the minimum z-value, and the maximum z-value in that order. Note that the values refer to grid nodes, i.e., to the corner of a grid cell or to the center of a voxel. In order to compute the width of a voxel, you should use code like this:

```
const int* dims = uniformcoords->dims();
const float* bbox = uniformcoords->bbox();
float width = (dims[0]>1) ? (bbox[1]-bbox[0])/(dims[0]-1):0;
```

Stacked Coordinates

Stacked coordinates are used to describe a stack of uniform 2D slices with variable slice distance. They are represented by the class HxStackedCoord3. This class provides a method bboxXY which returns a pointer to an array of four floats describing the bounding box of a 2D slice. In addition, the method coordZ returns a pointer to an array containing the z-coordinate of each 2D slice.

Rectilinear Coordinates

Same as for uniform or stacked coordinates, in the case of rectilinear coordinates the grid cells are aligned to the axes, but the grid spacing may vary from cell to cell in each direction. Rectilinear coordinates are represented by the class `HxRectilinearCoord3`. This class provides three methods, `coordX`, `coordY`, and `coordZ`, returning pointers to the arrays of x-, y-, and z-coordinates, respectively.

Curvilinear Coordinates

In the case of curvilinear coordinates, the position of each grid node is stored explicitly as a 3D vector of floats. A single grid cell need not to be axis-aligned anymore. An example of a 2D curvilinear grid is shown in Figure 5.1.
Curvilinear coordinates are represented by the class `HxCurvilinearCoord3`. This class provides a method `pos` which can be used to query the position of a grid node indicated by an index triple (i,j,k). Alternatively, a pointer to the coordinate values may be obtained by calling the method `coords`. The coordinate vectors are stored one after another without padding and with index i running fastest. Here is an example:

```
const int* dims = curvilinearcoords->dims();
const float* coords = curvilinearcoords->coords();

// Position of grid node (i,j,k)
float x = coords[3*((k*dims[1]+j)*dims[0]+i)];
float y = coords[3*((k*dims[1]+j)*dims[0]+i)+1];
float z = coords[3*((k*dims[1]+j)*dims[0]+i)+2];
```

5.2.3 Label Fields and the Label Lattice Interface

Label fields are used to store the results of an image segmentation process. Essentially, at each voxel a number is stored indicating which material the voxel belongs to. Consequently, label fields can be considered scalar fields. In fact, currently there are two different types of label fields, one for uniform coordinates (represented by class `HxUniformLabelField3` derived from `HxUniformScalarField3`) and one for stacked coordinates (represented by class `HxStackedLabelField3` derived from class `HxStackedScalarField3`). Since the two types are not derived from a common base class, a special-purpose interface called `HxLabelLattice3` is pro-

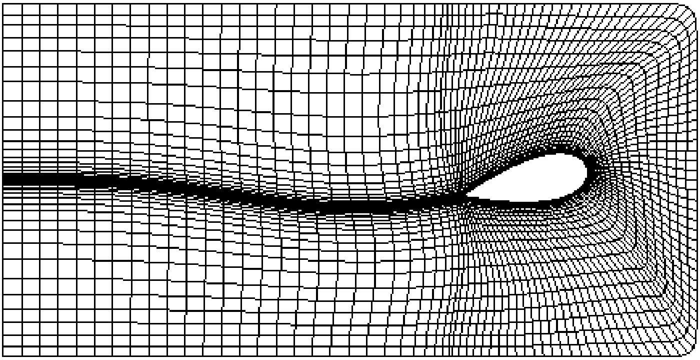

Figure 5.1: Example of a 2D grid with curvilinear coordinates.

vided. In fact, this interface is in turn derived from `HxLattice3`. It replaces the standard lattice variable of ordinary regular fields (see Section 5.2.1).

The primitive data type of a label field is always `McPrimType::mc_uint8`, i.e., bytes. In addition to the standard lattice interface, the label lattice interface also provides access to the label field's materials. Materials are stored in a special parameter subdirectory of the underlying data object. While discussing the plot API, we already encountered an example of how to interpret the materials of a label field (see Section 4.3.1). Note that whenever a new label is introduced, a new entry should also be put into the material list. Existing materials are marked so that they can not be removed from the material list (this would corrupt the labeling). In order to remove obsolete materials, call the method `removeEmptyMaterials` of `HxLabelLattice3`.

In addition to the labels, special weights can be stored in a label lattice. These weights are used to achieve sub-voxel accuracy when reconstructing 3D surfaces from the segmentation results. A pointer to the weights can be obtained by calling `getWeights` or `getWeights2` of the label lattice. For more details about `HxLabelLattice3`, please refer to the online class documentation.

5.2.4 Color Fields

Color fields are yet another type of regular fields. They consist of 4-component RGBA-byte-tuples and are represented by the class `HxRegColorField3` derived from `HxColorField3`. The latter class is closely related to `HxScalarField3` or `HxVectorField3`, see the overview on data class inheritance presented in Section 5.1.1. For color fields with uniform coordinates there is a special subclass `HxUniformColorField3`. Like any other regular fields, color fields provide a member `lattice` which can be used to access the data in a transparent way.

5.3 Unstructured Tetrahedral Data

Another important type of data refers to fields defined on unstructured tetrahedral grids. Such grids are often used in finite element simulations (FEM). In Amira, tetrahedral grids and data fields defined on such grids are implemented by two different classes or groups of classes and are also distinguished in the user interface by different icons. The reason is that by separating grid and data there is no need for replicating the grid in case many fields are defined on the same grid, a case that occurs frequently in practice.
In the following two sections, we introduce the *grid class* `HxTetraGrid` before discussing the corresponding *field classes* and the interface `HxTetraData`.

5.3.1 Tetrahedral Grids

Tetrahedral grids in Amira are implemented by the class `HxTetraGrid` and its base class `TetraGrid`. Looking at the reference documentation of `TetraGrid` we observe that a tetrahedral grid essentially consists of a number of dynamic arrays such as `points`, `tetras`, or `materialIds`.

- The `points` array is a list of all 3D points contained in the grid. A single point is stored as an element of type `McVec3f`. This class has the same layout as the Open Inventor class `SbVec3f`. Thus, a pointer to `McVec3f` can be cast to a pointer to `SbVec3f` and vice versa.
- The `tetras` array describes the actual tetrahedra. For each tetrahedron the indices of the four points and the indices of the four triangles it consists of are stored. The numbering of the points and triangles is shown in Figure 5.2. In particular, the fourth point is located above of the triangle defined by the first three points. Triangle number *i* is located opposite to point number *i*.
- The `materialIds` array contains 8-bit labels that assign a 'material' iden-

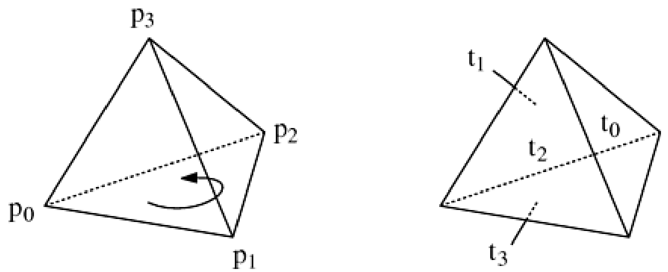

Figure 5.2: Numbering of points in a tetrahedron with positive volume (left). Numbering of the corresponding triangles (right).

tifier to every tetrahedron. For example, this is used in tetrahedral grids generated from segmented image data to distinguish between different image segments corresponding to different material components of physical objects represented by the (3D) image data Like in the case of label fields or surfaces, the set of possible material values is stored in the parameter list of the grid data object.

The three arrays, `points`, `tetras`, and `materialIds`, must be provided by the 'user'. The triangles of the grid are stored in an additional array called `triangles`. This array can be constructed automatically by calling the member method `createTriangles2`. This method computes the triangles from scratch and sets the triangle indices of all tetrahedra defined in `tetras`.
The `triangles` array also provides a way for accessing neighboring tetrahedra. Among other information (see reference documentation) stored for each triangle, the indices of the two tetrahedra it belongs to are available. In the case of boundary triangles, one of these indices is -1. Therefore, in order to get the index of a neighboring tetrahedron you can use the following code:

```
// Find tetra adjacent to tetra n at face 0:
int triangle = grid->tetras[n].triangles[0];
otherTetra = grid->triangles[triangle].tetras[0];
if (otherTetra == n)
    otherTetra = grid->triangles[triangle].tetras[1];
if (otherTetra == -1) {
    // No neighboring tetra, boundary face
    ...
}
```

Note that it is possible to define a grid with duplicated vertices, i.e. with vertices having exactly the same coordinates. This is useful to represent discontinuous data fields. The method *createTriangles2* checks for such duplicated nodes and correctly creates a single triangle between two geometrically adjacent tetrahedra, even if these tetrahedra refer to duplicated points.
Optionally the edges of a grid can be computed in addition to its points triangles, and tetrahedra by calling *createEdges*. The edges are stored in an array called `edges` and another array `edgesPerTetra` is used in order to store the indices of the six edges of a tetrahedron.
Moreover, the class `TetraGrid` provides additional optional arrays, for example to store a dynamic list of the indices of all tetrahedra adjacent to a particular point (`tetrasPerPoint`). This and other information is primarily used for internal purposes, for example to facilitate editing and smoothing of tetrahedral grids.

5.3.2 Data Defined on Tetrahedral Grids

In most applications, you will not only have to deal with a single tetrahedral grid, but also with data fields defined on it, for example scalar fields (e.g. temperature) or vector fields (e.g. flow velocity). Amira provides special classes for these data modalities, namely `HxTetraScalarField3`, `HxTetraVectorField3`, `HxTetraComplexScalarField3`, `HxTetraComplexVectorField3`, and `HxTetraField3` (see class hierarchy in Section 5.1.1).
Like in the case of regular data fields, the actual information is stored in a special member variable called *data*, which is of type `HxTetraData`. Like the corresponding member type `HxLattice3` for regular data, `HxTetraData` is an `interface`, i.e., derived from `HxInterface`. It provides transparent access to data fields defined on tetrahedral grids regardless of the actual number of data components of the field. In order to access that interface without knowing the actual type of input object within a module, you may use the following statement:

```
HxTetraData* data = (HxTetraData*)
    portData.source(HxTetraData::getClassTypeId());
if (!data) return;
```

Data on tetrahedral grids must always be of type `float`. The data values may be stored in three different ways, indicated by the encoding type as defined in `HxTetraData`:

- `PER_TETRA`: One data vector is stored for each tetrahedron. The data are assumed to be constant inside the tetrahedron.

- PER_VERTEX: One data vector is stored for each vertex of the grid. The data are interpolated linearly inside a tetrahedron.
- PER_TETRA_VERTEX: Four separate data vectors are stored for each tetrahedron. The data are also interpolated linearly.

This last encoding scheme is useful for modeling discontinuous fields. In order to evaluate a field at an arbitrary location in a transparent way, Amira's procedural data interface should be used. This interface is described in Section 5.5.1.
Like HxLattice3, the class HxTetraData provides a static method create which can be used to create a matching data field, e.g., an object of type HxTetraScalarField3, from an existing instance of HxTetraData. The HxTetraData object will not be copied but will be directly put into the field object. Therefore it may not be deleted afterwards. Also see Section 5.2.1.

5.4 Unstructured Hexahedral Data

In an unstructured hexahedral grid the grid cells are defined explicitly by specifying all the points in the cell. This is in contrast to regular hexahedral grids where the grid cells are arranged in a regular 3D array and thus are defined implicitly. The implementation of hexahedral grids is very similar to tetrahedral grids as described in the previous section. There are separate classes for the grid itself and for data fields defined on a hexahedral grid.
In the following two sections we introduce the *grid class* HxHexaGrid before discussing the corresponding *field classes* and the interface HxHexaData.

5.4.1 Hexahedral Grids

Hexahedral grids in Amira are implemented by the class HxHexaGrid and its base class HexaGrid. Looking at the reference documentation of HexaGrid we observe that a hexahedral grid essentially consists of a number of dynamic arrays such as points, hexas, or materialIds.

- The points array is a list of all 3D points contained in the grid. A single point is stored as an element of type McVec3f. This class has the same layout as the Open Inventor class SbVec3f. Thus, a pointer to McVec3f can be cast to a pointer to SbVec3f and vice versa.
- The hexas array describes the actual hexahedra. For each hexahedron the indices of the eight points and the indices of the six faces it consists of are

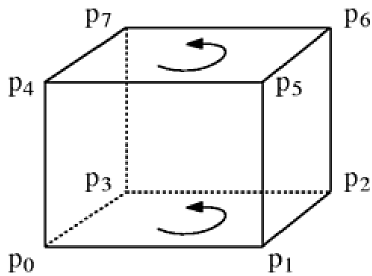

Figure 5.3: Numbering of points in a hexadron with positive volume.

stored. The numbering of the points is shown in Figure 5.3. Degenerate cells such as prisms or tetrahedra may be defined by choosing the same index for neighboring points.

- The `materialIds` array contains 8-bit labels, which assign a material identifier to every hexahedron. Like the case of label fields or surfaces, the set of possible material identifiers is stored in the parameter list of the grid data object.

The three arrays, `points`, `hexas`, and `materialIds`, must be provided by the 'user'. The faces of the grid are stored in an additional array called `faces`. This array can be constructed automatically by calling the member method `createFaces`. This method computes the faces from scratch and sets the face indices of all hexahedra defined in `hexas`.
Note that, in contrast to tetrahedral grids, in a hexahedral grid degenerate cells are allowed, i.e., cells where neighboring corners in a cell coincide. In this way grids with mixed cell types can be defined. The faces of a hexahedron are stored in a small dynamic array called `faces`. For a degenerate cell this array contains less then six faces.
Also note that, although non-conformal grids are allowed, i.e., grids with hanging nodes on edges and faces, currently the method *createFaces* does not detect the connectivity between neighboring hexahedra sharing less than four points. Thus, faces between such cells are considered to be external cells.

5.4.2 Data Defined on Hexahedral Grids

In most applications, you will not only have to deal with a single hexahedral grid, but also with data fields defined on it, for example scalar fields (e.g. temperature) or vector fields (e.g., flow velocity). Amira provides special classes for these data modalities, namely HxHexaScalarField3, HxHexaVectorField3, HxHexaComplexScalarField3, HxHexaComplexVectorField3, and HxHexaField3 (see class hierarchy in Section 5.1.1).
As for fields defined on tetrahedral grids, the actual information is stored in a special member variable called *data*, which is of type HxHexaData. HxHexaData is a so-called interface, i.e. derived from HxInterface. The *data* variable provides transparent access to data fields defined on hexahedral grids regardless of the actual number of data components the field has. In order to access the interface without knowing the actual type of input object within a module, you may use the following statement:

```
HxHexaData* data = (HxHexaData*)
    portData.source(HxHexaData::getClassTypeId());
if (!data) return;
```

Data on hexahedral grids must always be of type float. The data values may be stored in three different ways, indicated by the encoding type defined in HxHexaData:

- PER_HEXA: One data vector is stored for each hexahedron. The data are assumed to be constant inside the hexahedron.
- PER_VERTEX: One data vector is stored for each vertex of the grid. The data are interpolated trilinearly inside a hexahedron.
- PER_HEXA_VERTEX: Eight separate data vectors are stored for each hexahedron. The data are also interpolated trilinearly.

The last encoding scheme is useful for modeling discontinuous fields. In order to evaluate a field at an arbitrary location in a transparent way, Amira's procedural data interface should be used. This interface is described in Section 5.5.1.
Like HxLattice3, the class HxHexaData provides a static method create which can be used to create a matching data field, e.g., an object of type HxHexaScalarField3, from an existing instance of HxHexaData. The HxHexaData object will not be copied but will be directly put into the field object. Therefore it may not be deleted afterwards. Also see Section 5.2.1.

5.5 Other Issues Related to Data Classes

In this section the following topics will be covered:

- *Amira's procedural interface for evaluating 3D fields*
- *coordinate systems and transformations of spatial data objects*
- *defining parameters and materials in data objects*

5.5.1 Procedural Interface for 3D Fields

The internal representation of a data field very much depends on whether the field is defined on a regular, tetrahedral, or hexahedral grid. There are even data types such as `HxAnnaScalarField3` or `HxAnnaVectorField3` for fields that are defined by an analytical mathematical expression. To allow for writing a module which operates on any scalar field without having to bother about the particular data representation, a transparent interface is needed. One could think of a function like

```
float value = field->evaluate(x,y,z);
```

For the sake of efficiency, a slightly different interface is used in Amira. Evaluating a field defined on tetrahedral grid at an arbitrary location usually involves a global search to detect the tetrahedron which contains that point. The situation is similar for other grid types. In most algorithms, however, the field is typically evaluated at points not far from each other, e.g., when integrating a field line. To take advantage of this fact, the concept of an abstract `Location` class has been introduced. A `Location` describes a point in 3D space. Depending on the underlying grid, `Location` may keep track of additional information such as the current grid cell number. The `Location` class provides two different search strategies, a global one and a local one. In this way performance can be improved significantly. Here is an example of how to use a `Location` class:

```
float pos[3];
float value;
...
HxLocation3* location = field->createLocation();
if (location->set(pos))
    field->eval(location, &value);
...
if (location->move(pos))
    field->eval(location, &value);
```

```
    ...
    delete location;
```

First a location is created by calling the virtual method `createLocation` of the field to be evaluated. The two methods, `location->set(pos)` and `location->move(pos)`, both take an array of three floats as argument, which describe a point in 3D space. The `set` method always performs a global search in order to locate the point. In contrast, `move` first tries to locate the new point using a local search strategy starting from the previous position. You should call `move` when the new position differs only slightly from the previous one. Both `set` and `move` may return 0 in order to indicate that the requested point could not be located, i.e., that it is not contained in any grid cell.
In order to locate the field at a particular location, `field->eval( location, &value)` is called. The result is written to the variable pointed to by the second argument. Internally the `eval` method does two things. First it interpolates the field values, for example, using the values at the corners of the cell the current point is contained in. Secondly, it converts the result to a float value if the field is represented internally by a different primitive data type.

5.5.2 Transformations of Spatial Data Objects

In Amira, all data objects which are embedded in 3D space are derived from the class `HxSpatialData` defined in the subdirectory `kernel/Amira` (see class hierarchy in Section 5.1.1). On the one hand, this class provides a virtual method `getBoundingBox` which derived classes should redefine. On the other hand, it allows the user to transform the data object using an arbitrary geometric transformation. The transformation is stored in an Open Inventor *SoTransform* node. This node is applied automatically to any display module attached to a transformed data object.
In total there are three different coordinate systems:

- The *world coordinate system* is the system the camera of the 3D viewer is defined in.
- The *table coordinate system* is usually the same as the world coordinate system. However, it might be different if special modules displaying, for example, the geometry of a radiotherapy device is used. These modules should call the method `HxBase::useWorldCoords` with a non-zero argument in their constructor. Later they may then call the method `HxController::setWorldToTableTransform` of the global ob-

ject `theController`. In this way they can cause all other objects to be transformed simultaneously.

- Finally, the *local coordinate* system is defined by the transformation node stored for objects of type `HxSpatialData`. This transformation can be modified interactively using the transformation editor. Transformations can be shared between multiple data objects using the method `HxBase::setControllingData`. Typically, all display modules attached to a data object will share its transformation matrix, so that the geometry generated by these modules is transformed automatically when the data itself is transformed.

The transformation node of a spatial data object may be accessed using the `SoTransform* HxSpatialData::getTransform()` method, which may return a NULL pointer when the data object is not transformed.
Often it is easier to use `HxSpatialData::getTransform(SbMatrix& matrix)` instead, which returns the current transformation matrix or the identity matrix when there is no transformation. This matrix is to be applied by multiplying it to a vector from the right-hand side. It transforms vectors from the local coordinate system to the table or world coordinate system.
If you want to transform table or world coordinates to local coordinates, use `HxSpatialData::getInverseTransform( SbMatrix& matrix)`. For example, consider the following code which transforms the lower left front corner of object A into the local coordinate system of a second object B:

```
float bbox[6];
SbVec3f originWorld,originB;
SbMatrix matrixA, inverseMatrixB;

// Get origin in local coordinates of A
fieldA->getBoundingBox(bbox);
SbVec3f origin(bbox[0],bbox[1],bbox[2]);

// Transform origin to world coordinates:
fieldA->getTransform(matrixA);
matrixA.multVecMatrix(origin,originWorld);

// Transform origin from world coords to local coords of B
fieldB->getInverseTransform(inverseMatrixB);
inverseMatrixB.multVecMatrix(originWorld,originB);
```

Instead of this two-step approach, the two matrices could also be combined:

```
SbMatrix allInOne = matrixA;
```

```
allInOne.multRight(inverseMatrixB);

allInOne.multVecMatrix(origin,originB);
```

Note that the same result is obtained in the following way:

```
SbMatrix allInOne = inverseMatrixB;
allInOne.multLeft(matrixA);

allInOne.multVecMatrix(origin,originB);
```

Since the transformation could contain a translational part, special attention should be paid when directional vectors are transformed. In this case the method `HxSpatialData::getTransformNoTranslation( SbMatrix& matrix)` should be used.

5.5.3 Parameters and Materials

For every data object an arbitrary number of attributes or parameters can be defined. The parameters are stored in a member variable `parameters` of type `HxParamBundle`. The header file of the class `HxParamBundle` is located in the subdirectory `kernel/amiramesh`.
`HxParamBundle` is derived from the base class `HxParamBase`. Another class derived from `HxParamBase` is `HxParameter`. This class is used to actually store a parameter value. A parameter value may be a string or an n-component vector of any primitive data type supported in Amira (byte, short, int, float, or double). The bundle class `HxParamBundle` may hold an arbitrary number of `HxParamBase` objects, i.e., parameters or other bundles. In this way parameters may be ordered hierarchically.
Many data objects such as label fields, surfaces, or unstructured finite element grids make use of the concept of a `material` list. Material parameters are stored in a special sub-bundle of the object's parameter bundle called *Materials*. In order to access all material parameters of such an object, the following code may be used:

```
HxParamBundle* materials = field->parameters.materials();
int nMaterials = materials->nBundles();

for (int i=0; i<nMaterials; i++) {
    HxParamBundle* material = materials->bundle(i);
    const char* name = material->name();
    theMsg->printf("Material[%d] = %s\n", name);
}
```

The class `HxParamBundle` provides several methods for looking up parameter values. All these *find*-methods return 0 if the requested parameter could not be found. For example, in order to retrieve the value of a one-component floating point parameter called *Transparency*, the following code may be used:

```
float transparency = 0;
if (!material->findReal("Transparency",transparency))
    theMsg->printf("Transparency not defined, using default");
```

In order to add a new parameter or to overwrite the value of an existing one, you may use one of several different *set*-methods, for example:

```
material->set("Transparency",transparency);
```

Many modules check whether a color is associated to a particular 'material' in the material list of a data object. If this is not the case, the color or some other value is looked up in the global material database Amira provides. This database is represented by the class `HxMatDatabase` defined in `kernel/Amira`. It can be accessed via the global pointer `theDatabase`. Like an ordinary data object, the database has a member variable `parameters` of type `HxParamBundle` in order to store parameters and materials. In addition, it provides some convenience methods, for example `getColor(const char* name)`, which returns the color of a material, defining a new one if the material is not yet contained in the database.

6 Documentation of Modules in Developer Option

Developer Option allows the user to write the documentation for his or her own modules and integrate it into the user's guide. For this, in the package directory a subdirectory named `doc` need to be created, e.g.,
`AMIRA_LOCAL/src/mypackage/doc`
The documentation must be written in Amira's native documentation style. The syntax is borrowed from the LaTeX text processing language. See the reference guide in Chapter 7 for a complete list of commands.
Documentation files can easily be created by the `createDocFile` command. To create a documentation template for `MyModule`, type
`MyModule createDocFile`
in the Amira console. This will generate a template for the documentation file as well as snapshots of all ports. The file `MyModule.doc` already provides the skeleton for the module description and includes the port snapshots.
The command `createPortSnaps` only creates the snapshots of the module ports. This is useful when the ports have changed and their snapshots must be updated in the user's guide.

6.1 The Documentation File

Here, the basic elements of a documentation file are presented.

```
\begin{hxmodule2}{MyModule}{Short description to appear
in the table of modules}
This command indicates the beginning of a description file. MyModule
is the module name.

\begin{hxdescription}
This block contains a general module description.
```

```
All beginning blocks must have an end.
\end{hxdescription}

\begin{hxconnections}
\hxlabel{MyModule_data}
This command sets a label such that this
connection can be referenced in the documentation.
\hxconnection{Data}{[required]}\\
 Here the required master connection is described.

\end{hxconnections}

\begin{hxports}
The module ports are listed here.
\end{hxports}

Anywhere in the documentation a label can be referenced:
\link{MyModule_data}{Text for reference}


\end{hxmodule2}
```

This is an `hxmodule` description. Others are also available:

```
\begin{fileformat2}{name}{short description to appear
in the table of file formats}
\begin{data2}{name}{short description to appear
in the table of data types}
```

The file always must be closed by the corresponding `end` command.
More commands Other formats allow one to format and structure the documentation:

- ```
  \begin{itemize}
  \item This is an enumeration,
  and each item starts with the key word \item
  \end{itemize}
  ```
- `{\bf This will be set in bold face}`
- `{\it This will be set in italics}`
- `{\tt This will be set in Courier}`

Formulas can be included by means of the text processor LaTeX. They must be written in the usual LaTeX syntax, e.g., `$a \cdot \sqrt{b}$`. Doc2html uses MathJaX to display them, so there is no user action required to enable formula rendering.
  ```

6.2 Generating the Documentation

All documentation files must be converted to HTML files and copied into the user's guide. For this purpose, the program `doc2html` is provided. It is called automatically when local packages are compiled. If you want to create documentation files without compiling the local packages, run this program from a command shell with the following option:

```
doc2html -a
```

This gathers all the .doc files that can be found in local packages' /doc subfolders, converts these documentation files and writes the resulting HTML files to the appropriate places in the `AMIRA_LOCAL/share/doc/mymodule` directory.
Call `doc2html` without options to get a complete list of options.

7 Documentation Markup Language Reference Guide

This reference guide lists all commands that can be used when documenting Amira resources. These are a subset of common LaTeX commands as well as several Amira-specific concepts.
For a first overview on Amira's Documentation Markup Language, please read the introduction in Chapter 6.
To get detailed information, please continue here:

- *Document Structure*
- *Text Markup and Font Styles*
- *Images and Tables*
- *Math Environments*
- *Meta-Level Commands*
- *CSS Attributes*

7.1 Document Structure

The Amira documentation is split up into .doc and .tex files. Both of them contain plain text (the .doc extension is not related to Microsoft Word in this case).
The .doc files usually contain documentation for a single resource (module, port, file format, data type, editor, or demo). This information is later automatically added to the alphabetic reference guide by doc2html. Therefore, these .doc files must include a *header* that tells doc2html what kind of resource they describe. They may also be used for other general purposes if the header is omitted. In this case the files are also auto-detected but nothing is added to the alphabetic reference guide.
The .tex files are not detected automatically by doc2html. They should not have any header, and must be *included* manually inside one of the .doc/.tex files that *references* them.
The purpose of this construction is that it should be possible to add new documentation resources without having to include them from another existing file. Any .doc file that is located in a package subdirectory is automatically found and translated by doc2html, and is immediately available in the Amira Help Browser. This is especially useful when the user wishes to document his own resources, and to integrate them into the official Amira documentation: The official .doc/.tex files are not delivered to customers so there would be no possibility to include the new files anywhere.

7.1.1 Document Header: hxmodule, etc.

A .doc file document header must specify the package name using \hxpackname{PACKAGENAME}. It must also mention what *kind of an object* this file documents: a module, an editor, a component, a file format, a demo, an extension, or data. The header may contain a label name, which can later be used as the target of a link. Thus, the document structure is:

LaTeX	XHTML	Info
\hxpackname {PACKAGENAME}	-no output-	Replace PACKAGENAME by the Amira Option this file belongs to, usually amira or e.g., amiraMol. No output is rendered. The package name is needed for the alphabetic reference guide.
\begin{OBJECTTYPE} {PARAMETER1}{...}...	Headline	Describes the object that is documented here. See Section 7.1.1.1 for a list of object types.
\label{LABELNAME} *or* \hxlabel{LABELNAME}	<a name = LABELNAME/>	Label name. It will later be the target for linking to this file. Using \label or \hxlabel makes no difference in *this* case.
DOCUMENT CONTENT	Content	The document content is parsed here.
\end{OBJECTTYPE}	-no output-	Does not output anything, and is not needed for doc2html - but it must be there when building the PDF docs.

The three header commands may occur in arbitrary order. They should be the first lines of the document, and must not contain anything but whitespace between them.

7.1.1.1 Object Types

The following object types can be documented in .doc files. One of the commands is deprecated, and some of them are no longer supported in doc2html.

Object type	parameter1	parameter2	Status	Info
~~\begin{hxcomponent}~~	~~{componentName}~~	~~-none-~~	available (deprecated)	Describe a component
\begin{hxcomponent2}	{component}	{component Description}	available	Describe a component
\begin{hxdemo}	{demoName}	-none-	available	Describe a demo
\begin{hxextension}	{extensionName}	-none-	available	Describe an extension
~~\begin{hxmodule}~~	~~{moduleName}~~	~~-none-~~	not available	No longer supported
\begin{hxmodule2}	{moduleName}	{module Description}	available	Describe a module
~~\begin{hxeditor}~~	~~{editorName}~~	~~-none-~~	not available	No longer supported
\begin{hxeditor2}	{editorName}	{editor Description}	available	Describe an editor
~~\begin{hxfileformat}~~	~~{fileformatName}~~	~~-none-~~	not available	No longer supported
\begin{hxfileformat2}	{fileformatName}	{fileformat Description}	available	Describe a file format
~~\begin{hxdata}~~	~~{dataName}~~	~~-none-~~	not available	No longer supported
\begin{hxdata2}	{dataName}	{data Description}	available	Describe data
\begin{glslshader}	{shaderName}	-none-	available	Describe a GLSL shader
\begin{hxzibtutproj}	{projectName}	{project Description}	available	Describe a ZIB project tutorial
\begin{hxzibtutpub}	{publicationName}	{publication Description}	available	Describe a ZIB publication tutorial

7.1.2 Table of Contents

Note: The following commands should be used only for the official Amira documentation. When used while documenting custom Amira resources in the `AMIRA_LOCAL` directory, they may work unpredictably (or not at all).

LaTeX	XHTML	Info
\chapterblock {CHAPTERNAME}	 CHAPTERNAME 	Used only in usersguide/ index.doc to start a new chapter.
\tableofcontents	Table of contents	Used only in usersguide/toc.doc to render the table of contents.
\AmiraBaseExtensions	List of Amira Options	Used only in usersguide/toc.doc to list all available Amira Options.
\hxpublication	<! HxHelpHint: "publication based module" !>	Generate an HTML comment to note a related reference that need not be displayed.

7.1.3 Sections

7.1.3.1 LaTeX Style Chapters and Sections

Chapters, sections, and subsections will be visible in the table of contents.

LaTeX	XHTML	Info
\chapter{TEXT}	 TEXT 	Start a new chapter.
\section{TEXT}	 x TEXT 	Start a new section. It gets a new auto-generated section number.
\subsection{TEXT}	 x.y TEXT 	Start a new subsection. It gets a new auto-generated subsection number.
\subsubsection{TEXT}	 x.y.z TEXT 	Start a new subsubsection. It gets a new auto-generated subsubsection number.
\section*{TEXT}	 TEXT 	Start a new section.
\subsection*{TEXT}	 TEXT 	Start a new subsection.
\subsubsection*{TEXT}	 TEXT 	Start a new subsubsection.

7.1.3.2 {hxports}, {hxcommands} etc. Sections

These hx sections should be used to document properties of the current object - e.g., ports should be described inside an hxports section. It is not allowed to use hx commands like \hxport outside an {hxports} environment. See the next section for *usage and translation details.* The following hx sections are available:

LaTeX	XHTML	Info
\begin{hxcommands} TEXT \end{hxcommands}	<div class= "hxcommands"> <h3> Commands: </h3>\n TEXT </div>	Describe the Tcl commands of the object. Each command is started by the \hxcommand{COMM_NAME} command. The following TEXT should describe the command further.
\begin{hxdescription} TEXT \end{hxdescription}	<div class= "hxdescription"> <h3> Description: </h3>\n TEXT </div>	TEXT should describe the functionality of the object.
\begin{hxconnections} TEXT \end{hxconnections}	<div class= "hxconnections"> <h3> Connections: </h3>\n TEXT </div>	Describe the connections of the object. Each connection is started by the \hxconnection{CONN_NAME} command. The following TEXT should describe the connection further.
\begin{hxports} TEXT \end{hxports}	<h3> <div class ="hxports"> Ports: </h3>\n TEXT </div>	Describe the ports of the object. Each port is started by the \hxport{PORT} command. The following TEXT should describe the port further.
\begin{hxsection} {SECTIONNAME} TEXT \end{hxsection}	<h3> <div class= "hxsection"> SECTIONNAME: </h3>\n TEXT </div>	Start a section that looks like the ones above, but has a custom title.

7.1.3.3 \hxport, \hxcommand etc. Usage

Commands like \hxport can only be used inside an {hxports} environment. If it is used outside, a parse warning is generated. If it is used in the wrong environment, like \hxport inside {hxconnections}, no warning is given and the \hxport command is automatically treated as if it was an \hxconnection command. In fact, translation of these \hxport etc. commands only depends on the section type they are in, so we just call them *\hxitem* in this section. The syntax of an \hxitem is as follows. Commands in brackets [ ] are optional. The asterisk (*) indicates an arbitrary number of repetitions of this command:

```
\begin {SECTIONTYPE}

\hxitem {HEADLINE} [ {ADDENDUM} ]
[ \hximage {IMAGE1} ]*
TEXT

\end {SECTIONTYPE}
```

Translation depends on the section type, the addendum, and the presence of port snapshots.

- The addendum is used to indicate if an hxconnection is required or optional, so ADDENDUM may be "required", "optional", "unused", or empty (in this case braces can be omitted). Any LaTeX commands other than simple text are removed from the addendum to ensure a standardized markup via CSS. If we are currently inside an {hxconnections} environment, the following <div> tags will be generated: "hxconnection_required", "hxconnection_optional", or "hxconnection_unused". If we are not inside {hxconnections}, the addendum has no influence on the tags but is displayed only as text. See *below* for a usage example.
- When describing a port, port snapshots may be added. Any \hximage found between the \hxitem and the following text is considered to be a snapshot image and therefore put into <div class="portsnapshot"> tags. Note that this does not apply for \hximagescaled, which will never be detected as portsnapshot (would not make sense to resize snapshots).

LaTeX	XHTML
\begin{hxconnections} \hxitem{Data}{required} Description goes here \end{hxconnections}	<div class="hxconnections"> <div class="hxconnection_required"> Data required <p> Description goes here </p> </div> </div>
\begin{hxports} \hxitem{Portname} \hximage{snapshot1} \hximage{snapshot2} Description goes here \end{hxports}	<div class="hxports"> <div class="hxport"> Portname <div class="portsnapshots"> <img ....snapshot1...> <img ....snapshot2...> </div> <p> Description goes here </p> </div> </div>

Occasionally one may wish to end an \hxport, insert a headline, and then start the next \hxport. But without further precautions the headline would still be included in the <div> tags of the previous port, which would not look nice. The solution is to put an \endhxitem at the position where the last port should end.
This example illustrates the problem:

LaTeX	XHTML
\begin{hxports} *Headline1* *\hxitem*{Portname1} text1 *Headline2* *\hxitem*{Portname2} text2 \end{hxports}	<div class="hxports"> *Headline1* <div class="hxport"> Portname1 <p> text1 *Headline2* </p> </div> <div class="hxport"> Portname2 <p> text1 </p> </div> </div>

This is the recommended solution:

LaTeX	XHTML
\begin{hxports} *Headline1* *hxitem*{Portname1} text1 **\endhxitem** *Headline2* *hxitem*{Portname2} text2 \end{hxports}	<div class="hxports"> *Headline1* <div class="hxport"> Portname1 <p> text1 </p> </div> *Headline2* <div class="hxport"> Portname2 <p> text1 </p> </div> </div>

7.1.4 Links & Labels

Doc2html internally stores a list of labels that can be used as link targets, i.e., every <a name=... /> tag in all files can be referenced from any other file.

- To define a custom label anywhere in the text, use \hxlabel. This command prints out the <a name=... /> tag and registers the new label in the internal label list.
- The \label command yields no output, but merely registers the last <a name=... /> tag that was outputted by another function to the internal list. The only functions that generate this output themselves are \chapter, \section, \subsection, \subsubsection, and \figure. When linking to a \label, the chapter or figure number is automatically added to the linking text. Therefore, if an \hxlabel is used instead of a \label, links to this label will be displayed without the chapter number. For usage examples, see below.
- A link to a label can be set by several commands, which all take a link target and the text to be displayed as link. The commands are \link and \hxref (which do exactly the same), and the ref commands: \ref, \tref, \reft, and \treft. When building the PDF docs, the ref commands display the number of the linked chapter together with the given text, while the other commands simply display the target name in italic letters without a number. To link to a chapter, use \tref{Chapter}{LABELNAME}.
- An external URL can be linked using \URL. External links may not work in the Amira online help, but do work if the docs are displayed in a web browser.

7.1.4.1 Labels

LaTeX	XHTML	Info
\label{NAME}	-no output-	Add a label to the internal collection of labels. This command can only be called when a new chapter/section/.. begins, or inside a figure. The NAME is then stored as internal reference.
\hxlabel{NAME}		Define a custom label. It is automatically added to the internal collection of labels.

If an \hxlabel is combined with e.g., a \begin{hxsection} or similar section types, the order of the commands is not important:

```
\begin{hxsection}{Defining a break}
\hxlabel{DemoMaker_break}
text
\end{hxsection}
```

is the same as

```
\hxlabel{DemoMaker_break}
\begin{hxsection}{Defining a break}
text
\end{hxsection}
```

but the second variant should be preferred for performance reasons.

7.1.4.2 Links

LaTeX	XHTML	Info
\link{TARGET}{TEXT} *or* \hxref{TARGET}{TEXT}	 TEXT 	Link to a target label, or to a file. If the label does not exist, the target is assumed to be a file, and the .html extension is then added automatically. \hxref and \link are the same. Example: \link{HxColormap} {Colormap}
\ref{TARGET}{TEXT}	 TEXT 	Link to a target label.
\tref{TEXT}{TARGET}	 TEXT␣	e.g., \tref{Section} {SecCmdLine} (puts a space behind the text)
\reft{TARGET}{TEXT}	 ␣TEXT	e.g., \reft{Section} {SecCmdLine} (puts a space before the text)
\treft{TEXT_1} {TARGET} {TEXT_2}	 TEXT_1␣TEXT_2	e.g., \treft{Section1} {Headlinetext} {SecCmdLine} (puts a space between both texts)

7.1.4.3 URLs

LaTeX	XHTML	Info
\URL{TARGET}	 TARGET	Link to the external URL. Using mailto: is also allowed. \URL{example.com} \URL{mailto:abc@ example.com}
\URL[TEXT]{TARGET}	 TEXT	Link to the external URL using an alternative link description. Using mailto: is also allowed. \URL[linkname] {example.com} \URL[mail]{mailto:abc@ example.com}

7.2 Text Styles

7.2.1 Text Formatting

7.2.1.1 Size

LaTeX	XHTML
{\tiny TEXT}	TEXT
{\scriptsize TEXT}	TEXT
{\footnotesize TEXT}	TEXT
{\small TEXT}	TEXT
{\large TEXT}	TEXT
{\Large TEXT}	TEXT
{\LARGE TEXT}	TEXT
{\huge TEXT}	TEXT
{\Huge TEXT}	TEXT
{\HUGE TEXT}	TEXT

Commands like {\scriptsize TEXT} may also be combined with text markup commands, e.g., {\em \sf \scriptsize TEXT}.

7.2.1.2 Markup (bold, italic etc.)

LaTeX	XHTML	Info
\textbf{TEXT}	TEXT	**bold**
\textit{TEXT}	<i>TEXT</i>	*italic*
\texttt{TEXT}	<tt>TEXT</tt>	`typewriter`
{\bf TEXT}	TEXT	**bold**
{\it TEXT}	<i>TEXT</i>	*italic*
{\tt TEXT}	<tt>TEXT</tt>	`typewriter`
{\em TEXT}	TEXT	*emphasis* (usually italic)
{\sf TEXT}	 TEXT	different font for Product Names, e.g., Amira

Commands like {\em TEXT} may also be combined, even together with text size commands, e.g., {\em \sf \scriptsize TEXT}.

7.2.1.3 Alignment

LaTeX	XHTML
\begin{center} TEXT \end{center}	TEXT
\begin{flushleft} TEXT \end{flushleft}	TEXT
\begin{flushright} TEXT \end{flushright}	TEXT

7.2.2 Special Text-producing Commands

7.2.2.1 Product-specific Abbreviations

All product names may be abbreviated.

LaTeX	XHTML	Info
\AmiraVR	 Virtual Reality Option	different font
\ProductNamePlain	Amira	same font as usual
\ProductNameTT	<tt>Amira</tt>	in typewriter font

The following product names are available. Note that they are case-sensitive: \amira \AmiraDemo \AmiraDev \AmiraDicomReader \AmiraFEI \AmiraLD \AmiraMesh \AmiraMol \AmiraNeuro \AmiraQuant \AmiraSkel \AmiraVLD \AmiraVR \ProductMajorVersion \ProductName \ProductNamePlain \ProductNameTT \ProductVersion \ResolveRT \ResolveRTSKEL \VRmodule

7.2.2.2 Environment Variables

Every command that starts with \hx... is checked to see if it is an environment variable. To display the name `AMIRA_LOCAL`, the command \hxlocal should be used.

LaTeX	XHTML	Info
\hxENVVAR	<tt> ENVVAR </tt>	if ENVVAR is a known variable
\hxOTHER	\hxOTHER	syntax warning: unknown command

The following environment variables are used by Amira:
\hxburoot → AMIRABU_ROOT
\hxcheckfreeram → AMIRA_CHECK_FREERAM
\hxcomdesynctimeout → AMIRA_COM_DESYNC_TIMEOUT
\hxcomlog → AMIRA_COM_LOG
\hxcomresynctimeout → AMIRA_COM_RESYNC_TIMEOUT
\hxdata → AMIRA_DATA
\hxdatadir → AMIRA_DATADIR
\hxdebug → AMIRA_DEBUG
\hxdebugfuncfilter → AMIRA_DEBUG_FUNC_FILTER

\hxdebuglevel → AMIRA_DEBUG_LEVEL
\hxdebugpackagefilter → AMIRA_DEBUG_PACKAGE_FILTER
\hxdeconvnumthreads → AMIRA_DECONV_NUM_THREADS
\hxdefaultvisual → AMIRA_DEFAULTVISUAL
\hxdemos → AMIRA_DEMOS
\hxdicomjpegbase → AMIRA_DICOM_JPEG_BASE
\hxexecarch → AMIRA_EXEC_ARCH
\hxfakebutton → AMIRA_FAKEBUTTON
\hxforcefluentdialog → AMIRA_FORCE_FLUENT_DIALOG
\hxforcenoborder → AMIRA_FORCE_NOBORDER
\hxforcevolumeviz → AMIRA_FORCE_VOLUMEVIZ
\hxginocog → AMIRA_GI_NOCOG
\hxhostname → AMIRA_HOSTNAME
\hxinitfile → AMIRA_INIT_FILE
\hxjoblogfile → AMIRA_JOB_LOG_FILE
\hxlicensefile → AMIRA_LICENSE_FILE
\hxlocal → AMIRA_LOCAL
\hxmesa → AMIRA_MESA
\hxmeshdebug → AMIRAMESH_DEBUG
\hxmhtstereotaxis → AMIRA_MHT_STEREOTAXIS
\hxmolroot → AMIRAMOL_ROOT
\hxmultisample → AMIRA_MULTISAMPLE
\hxmultithread → AMIRA_MULTITHREAD
\hxnetworkdrivedir → AMIRA_NETWORKDRIVE_DIR
\hxnocontextsharing → AMIRA_NO_CONTEXT_SHARING
\hxnoindex → AMIRA_NO_INDEX
\hxnoindexthread → AMIRA_NO_INDEX_THREAD
\hxnolicensemessage → AMIRA_NO_LICENSE_MESSAGE
\hxnomultithread → AMIRA_NO_MULTITHREAD
\hxnooverlays → AMIRA_NO_OVERLAYS
\hxnospacemouse → AMIRA_NO_SPACEMOUSE
\hxnosplashscreen → AMIRA_NO_SPLASH_SCREEN
\hxnumrecentfiles → AMIRA_NUM_RECENTFILES
\hxoutofcoremanagement → AMIRA_OUTOFCORE_MANAGEMENT
\hxroot → AMIRA_ROOT
\hxshowglerrors → AMIRA_SHOW_GL_ERRORS
\hxsmallfont → AMIRA_SMALLFONT
\hxspacemouse → AMIRA_SPACEMOUSE

\hxstereoondefault → AMIRA_STEREO_ON_DEFAULT
\hxteamworkavatar → AMIRA_TEAMWORK_AVATAR
\hxtecplotusetimeseries → AMIRA_TECPLOT_USE_TIME_SERIES
\hxtexmem → AMIRA_TEXMEM
\hxtranslationfile → AMIRA_TRANSLATION_FILE
\hxuservideomemamount → AMIRA_USER_VIDEO_MEM_AMOUNT
\hxvrcaliblog → AMIRAVR_CALIB_LOG
\hxvrconfigspath → AMIRA_VR_CONFIGS_PATH
\hxxshm → AMIRA_XSHM
\hxziehmreconstruction → AMIRA_ZIEHM_RECONSTRUCTION

7.2.2.3 Special Characters

LaTeX	XHTML	Info
\{	{	
\}	}	
\$	$	
\tcl	Tcl	
\textregistered	®	®
\quad		outputs a single space
\qquad		outputs a single space
\hxrightarrow	→	→
\hxleftarrow	←	←
\textasciicircum	^	circumflex
\hxpipe	|	
\hxdegree	°	°
\AA	Å	Å
\^	^	
\-	-no output-	command has no effect
\’a	á	works also for e,i,o,u
\‘a	à	works also for e,i,o,u
\’ *or* ’	’	single quote
\" *or* " *or* "’ *or* "‘ *or* ’" *or* ‘" *or* \textquotedbl	"	double quote
\backslash	\	
\textbackslash	\	
\#	#	
\textvisiblespace	␣	visible space
_	_	underscore
<	<	
>	>	
\%	%	
~		non-breaking space
\zerolengthspace	‌	zero-length space
\~	˜	
\&	&	&

LaTeX	XHTML	Info
\textasciitilde	˜	~
\mu *or* \textmu *or* µ	µ	µ
\lambda	λ	λ
\sigma	σ	σ
[	[	
]	]	
{	-no output-	syntax error if no closing brace found
}	-no output-	syntax error if no opening brace found

7.2.2.4 Non-breaking Space for Text Indention

LaTeX	XHTML	Info
\hxbl		
\hxbbl		
\hxbbbl		
\hxbbbbl		
\hxbbbbbl		
\hxtenbl		
~~\indent~~		The command has no effect.
~~\noindent~~		The command has no effect.

7.2.2.5 Newlines

LaTeX	XHTML	Info
\newline	 	
\tabularnewline	 	
\\	 	
\smallskip	 	
\medskip	 	
\bigskip	 	
\newpage	 	
\vspace	 	optional parameters are ignored
\n	\n	no effect on output, only improves .html readability
\n\n		start new paragraph (\n\n means: an empty line)

7.2.3 Lists & Enumerations

7.2.3.1 List Types

LaTeX	XHTML	Info
\begin{enumerate} \item TEXT1 \item TEXT2 \end{enumerate}	 - TEXT1 - TEXT2 	Items have numbers
\begin{itemize} \item TEXT1 \item TEXT2 \end{itemize}	 - TEXT1 - TEXT2 	Items have symbols
\begin{description} \item[DESCR1] TEXT1 \item[DESCR2] TEXT2 \end{description}	<dl> <dt> DESCR1 </dt> <dd> TEXT1 </dd> <dt> DESCR1 </dt> <dd> TEXT1 </dd> </dl>	Items have custom description

Lists of different types may be nested inside each other. Using \item or \item[] in the wrong environment is not recommended, and gives a parse warning or error (but is rendered anyway).

7.2.3.2 Items

LaTeX	XHTML	Info
\item TEXT	TEXT	Simple item
\item{TEXT}	TEXT	Simple item
\item[TEXT_1] TEXT_2	<dt> TEXT_1 </dt> <dd> TEXT_2 </dd>	Item with special caption
\item[TEXT_1]{TEXT_2}	<dt> TEXT_1 </dt> <dd> TEXT_2 </dd>	Item with special caption

Items may contain \n\n to indicate that a new paragraph begins. In this case, the item is automatically enclosed in <p>...</p> tags.

7.2.4 FAQ Environment

There is an {faq} environment, which is used for usersguide/doc/faq.doc. Every new {faq} environment resets the section counter, so that one may first write down only the questions, and give the answers in the next {faq} environment, which automatically gets the same enumeration. Usage example:

```
\begin{faq}
\faqsection{HEADLINE_A}
\faqitem{\link{q1}{QUESTION_1}}
\faqitem{\link{q2}{QUESTION_2}}
\faqsection{HEADLINE_B}
\faqitem{\link{q3}{QUESTION_3}}
\end{faq}
[...]
\begin{faq}
\faqsection{HEADLINE_A}
\hxlabel{q1}
\faqitem{QUESTION_1} ANSWER_1
\hxlabel{q2}
\faqitem{QUESTION_2} ANSWER_2
\faqsection{HEADLINE_B}
\hxlabel{q3}
\faqitem{QUESTION_3} ANSWER_3
\end{faq}
```

LaTeX	XHTML	Info
\begin{faq} TEXT \end{faq}	TEXT	Environment that allows FAQ items
\faqsection{TEXT}	TEXT	Starts a new section of FAQ items
\faqitem{TEXT1} TEXT2	 TEXT1 TEXT2 	Inserts an FAQ item. TEXT1 is the question, TEXT2 is the answer. The answer may be set in braces, or may be omitted.

7.2.5 Quotes

LaTeX	XHTML	Info
\begin{quote} TEXT \end{quote}	<blockquote> TEXT </blockquote>	Cite some text

7.2.6 Verbatim

7.2.6.1 Verbatim Inline

LaTeX	XHTML	Info
\verb*TEXT*	<tt>TEXT</tt>	Verbatim text without starting a new paragraph

The * stands for an arbitrary character that indicates the beginning and end of the verbatim environment. The text between the * is copied directly to the output without parsing any LaTeX commands it may contain. HTML code inside the text is escaped.

7.2.6.2 Verbatim Paragraphs

LaTeX	XHTML	Info
\begin{verbatim} TEXT \end{verbatim}	<pre>TEXT</pre>	Verbatim text inside its own paragraph

The text is copied directly to the output without parsing any LaTeX commands it may contain. HTML code inside the text is escaped.

7.3 Displaying Images and Tables

Figures can be used only to *display images*. Enclosing a *table* in a figure has no effect and is not recommended.
If no caption and no references are needed for an image, it is not necessary to use a figure environment to display it. The \hximage and \hximagescaled commands can stand alone anywhere in the text.

7.3.1 Figures

A figure environment must contain the image to be displayed, given by an \hximage or \hximagescaled command. It may contain a \caption, a \label, and \begin{center} .. \end{center} tags.
If the image is included by \hximagescaled, and the scale factor is != 1, an additional HTML page is generated that contains the image in its original size and its caption. That page is displayed if the scaled image is clicked.
Images are searched only in the folder where the current .doc/.tex file is located.
Usage example:

```
\begin{figure}
\begin{center}
\hximagescaled{HxDemoGUI_Selection_Tab}{0.5\textwidth}
\caption{Selection tab with one selected demo}
\label{ImgSelectionTab}
\end{center}
\end{figure}
```

LaTeX	XHTML	Info
\begin{figure} ... \end{figure}	-no output-	The command indicates that the following \caption, \label, and \hximage belong together
\begin{floatingfigure} ... \end{floatingfigure}	-no output-	Same as figure, but implicitly centers image and caption

7.3.2 \hximage and \hximagescaled

LaTeX	XHTML	Info
\hximage {IMAGE_FILENAME}		If the image filename has no extension, .png is assumed.
\hximagescaled {IMAGE_FILENAME} [SCALEFACTOR]		The image is rescaled by the scale factor. See the next subsection for syntax details. Another HTML subpage is generated that contains the original size image and its caption; that page is displayed if the rescaled image is clicked.

7.3.2.1 The \hximagescaled Scale Factor

The scale factor may be expressed with or without the \textwidth parameter in LaTeX. Omitting the \textwidth in doc2html displays the image in its original size, because HTML does not know how to handle input like "2cm" or "4.7in". In detail, these cases may occur:

- The \textwidth parameter is not given,
  e.g., \hximagescaled{myimage.png}{4cm}
  →the image is displayed in its original size
- The \textwidth parameter is used:
  - a valid scale factor is given,
    e.g., \hximagescaled{myimage.png}{0.4\textwidth}
    →the image is displayed using width = scale * 750
  - no scale factor is given,
    e.g., \hximagescaled{myimage.png}{\textwidth}
    →the image is displayed using width = 750
  - the given scale factor is invalid,
    e.g., \hximagescaled{myimage.png}{1.2.3.4\textwidth}
    →the image is displayed in its original size

7.3.3 Tables

Tables are generated by using \begin{tabular}{TABHEAD} ... \end{tabular}. The TABHEAD determines the number of columns and their alignment, e.g., {llrc} gives four columns: the first two are left-aligned, the third is right-aligned, the last one is centered. The LaTeX character '|' to indicate lines between columns is ignored, so {llrc} is the same as {|l|l|r|c|}. Using a 'p' in the table header to indicate a fixed width column is interpreted as if it was an 'l' (the column width is always determined by doc2html and cannot be overridden).

The \begin{table}[] ... \end{table} commands to determine the location of the table are not available in doc2html (tables are always outputted where they appear in text - no need to move them e.g., to the top of the next page). If these commands are needed for the PDF doc, they must be enclosed in \ifdefined\PDFONLY tags.

Tables may be not nested inside each other.

Usage example:

```
\ifdefined \PDFONLY \begin{table}[htb] \fi
\begin{center}
\begin{tabular}{ll}
TEXT1 & TEXT2 \\
TEXT4 & TEXT3 \\
TEXT5 & TEXT6
\caption{CAPTIONTEXT \hxlabel{LABELNAME}}
\end{tabular}
\end{center}
\ifdefined \PDFONLY \end{table} \fi
```

Caption, center, and hxlabel commands are optional.

LaTeX	XHTML	Info
\begin{tabular} {TABULARHEAD}	<table border=0 cellspacing=4>\n	TABULARHEAD may contain only 'l','c','r','p'
TEXT1 & TEXT2 \\	<tr> <td align=...> TEXT1 </td> <td align=...> TEXT2 </td> </tr>	Alignment is read from TABULARHEAD
\end{tabular}	</table>	

Tables inside this reference guide are marked by <div class="d2htable"> so that they may be formatted differently by CSS. To get these tags, use the LaTeX environment {d2htables}.

LaTeX	XHTML	Info
\begin{d2htables}	-no output-	
TEXT \begin{tabular} {TABULARHEAD} TABLE \end{tabular} {TABULARHEAD} TEXT	TEXT <div class="d2htable"> <table ...> TABLE </table> </div> TEXT	
\end{d2htables}	-no output-	

7.4 Math

Doc2html uses MathJaX to render math. Every .html file automatically gets a special header entry that links to the MathJaX library. This means that LaTeX math environments can be copied "as is" from input .doc/.tex to output .html files. The MathJaX library must be located in the same path as the .html documentation subfolders, usually /share/doc/amira/mathjax. It need not be copied to AMIRA_LOCAL because doc2html changes the path accordingly when building the docs in AMIRA_LOCAL.
MathJaX can be downloaded from http://mathjax.org.

7.4.1 Inline

LaTeX	XHTML	Info
$ MATH_TEXT $	$$ MATH_TEXT $$	inline math

7.4.2 Equations

LaTeX	XHTML	Info
\[MATH_TEXT \]	\[MATH_TEXT \]	math paragraph
\begin{eqnarray} MATH_TEXT \end{eqnarray}	\[\begin{eqnarray} MATH_TEXT \end{eqnarray} \]	equation array
\begin{eqnarray*} MATH_TEXT \end{eqnarray*}	\[\begin{eqnarray*} MATH_TEXT \end{eqnarray*} \]	equation array (don't enumerate equations)

7.5 Meta-Level Commands

Note: Some of the following commands refer to internal features of the official Amira documentation. In particular, it is not possible to generate a PDF file from the documentation of the `AMIRA_LOCAL` resources. Therefore, no PDF-related commands should be used when documenting local resources, and the only important information in this section is probably on how to *include different input files*.

7.5.1 Conditions Using \ifdefined

LaTeX	XHTML	Info
\ifdefined \VARIABLE TEXT \fi	TEXT -no output-	if \VARIABLE is set no output if \VARIABLE is not set
\ifdefined \VARIABLE TEXT_1 \else TEXT_2 \fi	TEXT_1 TEXT_2	if \VARIABLE is set if \VARIABLE is not set

The TEXT is parsed normally; it may also contain any LaTeX commands. The following variables are available:

\VARIABLE	Effect
\Amira	True if building docs for Amira
\PDFONLY	True if rendering the PDF version, false if rendering HTML
\FEIONLY	True if building docs for FEI Edition

7.5.2 Differentiate between PDF and HTML Output

LaTeX	XHTML	Info
\begin{hxhtml} TEXT \end{hxhtml}	TEXT	The text is copied "as is" to the output
\begin{hxtex} TEXT \end{hxtex}	-no output-	No output is rendered

Use {hxhtml} to write HTML code directly to the output file. It does not appear in the PDF version. Use {hxtex} to write text that only appears in the PDF version, and is not rendered in the HTML files.
In contrast to "\ifdefined \PDFONLY..." (see above), the TEXT written into the hxhtml environment is not parsed by doc2html, it is written to the output file "as is".

7.5.3 Include Different Input Files

LaTeX	XHTML	Info
\input{FILENAME}	-starts new file-	No output rendered in the current file. The content of FILENAME is parsed, and written to a new output file named FILENAME.html.

FILENAME is usually a string like "../../hxtracking/doc/HxVR-trackemu.tex" (without quotes). All paths are relative to the "usersguide/doc" directory because usersguide/doc/index.doc is the root file, which includes all others. If the command is used for files in AMIRA_LOCAL, the relative path need not be adopted (doc2html treats AMIRA_LOCAL and AMIRA_ROOT together as if it was a single directory relative to the "usersguide/doc" directory).

7.5.4 Exclude Files

LaTeX	XHTML	Info
\hxpackname{exclude}	-no output-	No output rendered in this file. The complete file is discarded, and parsing continues with the next file in queue.

7.5.5 Comments

LaTeX	XHTML	Info
%.............	-no output-	LaTeX comment: the rest of the line after the % is ignored

7.6 CSS Attributes

This section lists the Amira-specific CSS attributes, which can be accessed when editing `share/doc/Amira.css`.

7.6.1 Headlines

CSS Attribute	LaTeX Command	XHTML
generic_head	\begin{hxmodule2} {MYMODULE} {DESCRIPTION} ... \end{hxmodule2}	<div class= "generic_head"> MYMODULE </div>.

7.6.2 Front Page (index.doc)

CSS Attribute	LaTeX Command	XHTML
head_menu	-no output-	*Text that is to appear on the top line of the* Amira *front page is contained in an* <ul id="head_menu"> ... *list.*
imageBarPad	-no output-	*Determine the vertical space between the front page top line and the images below it by setting* .imageBarPad {height: 30px;}.
imageBarSpace	-no output-	*Determine the vertical space between front page images by setting* .imageBarSpace {height: 7px;}.

7.6.3 hx Environments

7.6.3.1 Connections

CSS Attribute	LaTeX Command	XHTML
hxconnections hxconnection_required hxconnection_optional hxconnection_unused hxconnection_default connectiontype headline	\begin{hxconnections} \hxconnection {MY_CONNECTION} {[required]} ...CONNECTION... \end{hxconnections}	*Depending on the connection type:* <div class= "hxconnections"> <div class= "hxconnection _required"> <div class="headline"> MY_CONNECTION </div> <div class= "connectiontype> [required] </div> ...CONNECTION... </div> ... </div>

7.6.3.2 Ports

CSS Attribute	LaTeX Command	XHTML
hxports hxport portsnapshot headline	\begin{hxports} \hxport{MY_PORT} \hximage{MY_PORT_ SNAPSHOT} ...PORT... \end{hxports}	<div class="hxports"> <div class="hxport"> <div class="headline"> MY_PORT </div> <div class= "portsnapshot"> MY_PORT_SNAPSHOT </div> ...PORT... </div> ... </div>

7.6.3.3 Commands

CSS Attribute	LaTeX Command	XHTML
hxcommands hxcommand headline	\begin{hxcommands} \hxcommand {MY_COMMAND} ...COMMAND... \end{hxcommands}	<div class= "hxcommands"> <div class= "hxcommand"> <div class="headline"> MY_COMMAND </div> ...COMMAND... </div> ... </div>

7.6.3.4 Description

CSS Attribute	LaTeX Command	XHTML
hxdescription	\begin{hxdescription} ...DESCRIPTION... \end{hxdescription}	<div class= "hxdescription"> DESCRIPTION </div>.

7.6.3.5 Section

CSS Attribute	LaTeX Command	XHTML
hxsection headline	\begin{hxsection} {SECTION_NAME} ...TEXT... \end{hxsection}	<div class= "hxsection"> <div class="headline"> SECTION_NAME </div> ...TEXT... </div>

7.6.4 Tables

CSS Attribute	LaTeX Command	XHTML
d2htables	\begin{d2htables} TEXT \begin{tabular} {TABULARHEAD} TABLE \end{tabular} {TABULARHEAD} TEXT \end{d2htables}	TEXT <div class="d2htable"> <table ...> TABLE </table> </div> TEXT

8 Miscellaneous

This chapter covers a number of additional issues of interest for the Amira developer. In particular, the following topics are included:

- *Import of time-dependent data*, including the use of `HxPortTime`
- *Important global objects*, such as `theMsg` and `theWorkArea`
- *Save-network issues*, making save-network work for custom modules
- *Using version information*, write version dependent code
- *Troubleshooting*, providing a list of common errors and solutions

8.1 Time-Dependent Data And Animations

This section covers some more advanced topics of Developer Option, namely the handling of dynamic data sets and the implementation of animated compute tasks. Before reading the section you should at least know how to write ordinary IO routines and modules.

8.1.1 Time Series Control Modules

In general, the processing of time-dependent data sets is a challenging task in 3D visualization. Usually not all time steps of a dynamic data series can be loaded at once because of insufficient main memory. Even if all time steps would fit into memory it is usually not a good idea to load every time step as a separate object in Amira. This would result in a large number of icons in the Pool. The selection between different time steps would become difficult.
A better solution comprise special-purpose control modules. An example is the *time series control module* described in the user's guide. This module is created if a time series of data objects each stored in a separate file is imported via the *Load time series...* option of the main window's file menu. Instead of loading all time steps together the control module loads only one time step at once. The current

time step can be adjusted via a time slider. When a new time step is selected the data objects associated with the previous one are replaced.
If you want to support a file format where multiple time steps are stored in a common file, you can write a special time series control module for that format. For each format a special control module is needed because seeking for a particular time step inside the file of course is different for each format. For convenience, you may derive a control module for a new format from the class *HxDynamicDataControl* contained in the package *hxtime*. This base class provides a time slider and a virtual method `newTimeStep(int k)` which is called whenever a new new time step is to be loaded. In contrast to the standard time series control module in most other control modules data objects should be created only once. If a new time step is selected existing objects should be updated and reused instead of replacing them by new objects. In this way the burden of disconnecting and reconnecting down-stream objects is avoided.

8.1.2 The Class HxPortTime

In principal, an ordinary float slider (*HxPortFloatSlider*) can be used to adjust the time of a time series control module or of some other time-dependent data object. However, in many cases the special-purpose class *HxPortTime* defined in the package *hxtime* is more appropriate. This class can be used like an ordinary float slider but it provides many additional features. The most prominent one is the possibility to auto-animate the slider. In addition, *HxPortTime* can be connected to a global time object of type *HxTime*. In this way multiple time-dependent modules can be synchronized. In order to create a global time object, choose *Time* from the main window's *Create* menu.
Another feature of *HxPortTime* is that the class is also an interface, i.e., it is derived from *HxInterface* (compare Section 5.1.2). In this way it is possible to write modules which can be connected to any object containing an instance of *HxPortTime*. An example is the *DisplayTime* module. In order to access the time port of a source object the following C++ dynamic cast construct should be used:

```
HxPortTime* time = dynamic_cast<HxPortTime*>(
    portData.source(HxPortTime::getClassTypeid()));
```

In the previous section we discussed how time-dependent data could be imported using special-purpose control modules. Another alternative is to derive a time-dependent data object from an existing static one. An example of this is the class

MyDynamicColormap contained in the demo package of Developer Option. Looking at the header file `packages/mypackage/MyDynamicColormap.h` in the local Amira directory you notice that this class is essentially an ordinary colormap with an additional time port. Here is the class declaration:

```
class MYPACKAGE_API MyDynamicColormap : public HxColormap
{
    HX_HEADER(MyDynamicColormap);

  public:
    // Constructor.
    MyDynamicColormap();

    // This will be called when an input port changes.
    virtual void compute();

    // The time slider
    HxPortTime portTime;

    // Implements the colormap
    virtual void getRGBA1(float u, float rgba[4]) const;
};
```

The implementation of the dynamic colormap is very simple too (see the file `MyDynamicColormap.cpp`). First, in the constructor the time slider is initialized:

```
portTime.setMinMax(0,1);
portTime.setIncrement(0.1);
portTime.setDiscrete(0);
portTime.setRealTimeFactor(0.5*0.001);
```

The first line indicates that the slider should go from 0 to 1. The increment set in the next line defines by what amount the time value should be changed if the backward or the forward button of the slider is pressed. The next line unsets the discrete flag. If this flag is on, the slider value always would be an integer multiple of the increment. Finally, the so-called real-time factor is set. Setting this factor to a non-zero value implies that the slider is associated with physical time in animation mode. More precisely, the number of microseconds elapsed since the last animation update is multiplied with the real-time factor. Then the result is added to the current time value.

In order to see the module in action compile the demo package, start Amira (use the `-debug` option or the debug executable if you compiled in debug mode), and choose *DynamicColormap* from the main window's *Create* menu. Attach a *DisplayColormap* module to the colormap and change the value of the colormap's time slider. Animate the slider. The speed of the animation can be adjusted by resetting the value of the real-time factor using the Tcl command `DynamicColormap time setRealTimeFactor`.

8.1.3 Animation Via Time-Out Methods

In some cases you might want certain methods to be called in regular intervals without using a time port. There are several ways to do this. First, you could use the Open Inventor class `SbTimerSensor` or related classes. Another possibility would be to use the Qt class `QTimer.` However, both methods have the disadvantage that the application can get stuck if too many timer events are emitted at once. In same cases it could even be impossible to press the stop button or some other button for turning off user-defined animation. For this reason Amira provides its own way off registering time-out methods. The relevant methods are implemented by the class `HxController`. Suppose, you have written a module with a member method called `timeOut`. If you want this method to be called automatically once in a second, you can use the following statement:

```
    theController->addTimeOutMethod(
this,(HxTimeOutMethod)timeOut,1000);
```

In order to stop the animation again, use

```
    theController->removeTimeOutMethod(
this,(HxTimeOutMethod)timeOut);
```

Instead of using a member method of an Amira object class, you can also register an arbitrary static function using the method `addTimeOutFunction` of class `HxController`. The corresponding remove method is called `removeTimeOutFunction`. For more information, see the reference documentation of `HxController`.
The Developer Option demo package contains the module `MyAnimateColormap` which makes use of the above time-out mechanism. The source code of the module again is quite easy to understand. After compiling the demo package, you can attach to module under the name *DoAnimate* to an existing colormap. The colormap then is modified and copied. After pressing the animate toggle of the module the output colormap is shifted automatically at regular intervals. Note that in this example the `fire` method of the module is

used as time-out method. `fire` invokes the modules `compute` method and also updates all down-stream objects.

8.2 Important Global Objects

Beside the base classes of modules and data objects, there are some more classes in the Amira kernel that are important to the developer. Many of these classes have exactly one global instance. A short summary of these global objects is presented here. For details, please refer to the online reference documentation by looking at the file `share/programmersguide/index.html` in the Amira root directory.

HxMessage: This class corresponds to the Amira console window in the lower right part of the screen. There is only one global instance of this class, which can be accessed by `theMsg`. All text output should go to this object. Text can be printed using the function `theMsg->printf("...",...)`, which supports common C-style `printf` syntax. `HxMessage` also provides static methods for popping up error and warning messages or simple question dialogs.

HxObjectPool: This class maintains the list of all currently existing data objects and modules. In the graphical user interface the Pool is represented by the upper area in the main window containing the modules' and data objects' icons. There is only one global instance of this class, which can be accessed by the pointer `theObjectPool`.

HxWorkArea: This class displays the ports of selected objects in the Properties Area and provides the progress bar and busy-state functionality. Important functions are `startWorking`, `stopWorking`, `wasInterrupted` as well as `busy` and `notBusy`. There is only one global instance of this class, which can be accessed by the pointer `theWorkArea`.

HxFileDialog: This class represents the file browser used for loading and saving data. Normally the developer does not need to use this class since the standard I/O mechanism is completely implemented in the Amira kernel. However, for special purpose modules, a separate file browser might be useful. There is a global instance of this class, which can be accessed by the pointer `theFileDialog`.

HxResource: This class maintains the list of all registered file formats and modules as defined in the package resource files. It also provides information about the Amira root directory, the local Amira directory, the version number, and so on. Normally there is no need for the developer to use this class directly. There is no instance of this class, since all its members are static.

HxViewer: This class represents an Amira 3D viewer. There can be multiple instances which are accessed via the method `viewer` of the global object `theController`. Normally you will not not need to use this class. Instead, you should use the member functions `showGeom` and `hideGeom` which every module and data object provides in order to display geometry.
HxController: This class controls all 3D viewers and Open Inventor geometry. In order to access a viewer you may use the following statement:

```
HxViewer* v0 = theController->viewer(0,0);
```

The first argument indicates the ID number of the viewer to be retrieved. In total there may be up to 16 different viewers. The second argument specifies whether the viewer should be created or not if it does not already exist.
HxColorEditor: Amira's color editor. Used, for example, to define the background color of the viewer. In a standard module you should use a port such as `HxPortColorList` or `HxPortColorMap` instead of directly accessing the color editor. There is a global instance of this class, which can be accessed by the pointer `theColorEditor`.
HxHTML: A window used to display HTML files. This class is used for Amira's online help. The global instance used for displaying the online user's guide and the online programmer's guide can be accessed by the pointer `theBrowser`.
HxMatDatabase: This class represents Amira's global parameter and material database. For example, the database contains default colors for a number of biomedical tissue types such as fat, muscle, and bone. The database can be accessed by the global pointer `theDatabase`. Details about the material database are discussed in Section 5.5.3.

8.3 Save-Network Issues

This section describes the mechanism used in Amira to save networks. For most modules this is done transparently for the developer.
The menu command "Save Network" dumps a Tcl script that should reconstruct the current network. Essentially this is done by writing a `load ...` command for each data object, a `create ...` command for each module and `setValue ...` commands for each port of a module.
This suffices to reconstruct the network correctly if all information about a module's state is kept in the module's ports only. If this is not the case, e.g., if the developer uses extra member variables that are important for the modules current state, those values are not restored automatically. If you cannot avoid this, you

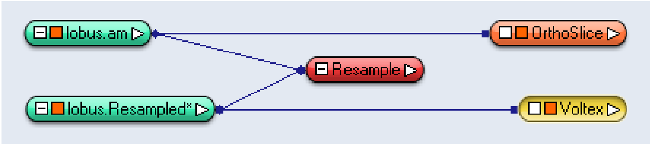

Figure 8.1: When loading this network, the *Resample* module recreates the *lobus.Resampled* data object on the fly.

must extend the 'Save Network' functionality of your module. In order to do so, you can override the virtual function `savePorts` so that it writes additional Tcl commands. For example, let us take a look at the `HxArbitraryCut` class, which is the base class e.g., for the `ObliqueSlice` module and which has to save its current slice orientation:

```
void HxArbitraryCut::savePorts(FILE* fp)
{
    HxModule::savePorts(fp);
    ...
    fprintf(fp, "%s setPlane %g %g %g %g %g %g %g %g %g\n",
        getName(),
        origin()[0], origin()[1], origin()[2],
        uVec()[0], uVec()[1], uVec()[2],
        vVec()[0], vVec()[1], vVec()[2]);
}
```

Note that this method requires that `HxArbitraryCut` or some of its parent classes implement the Tcl command `setPlane`. Hints about implementing new Tcl commands are given in Section 4.2.2.
Some remarks about how to generate the load command for data objects are given in Section 3.2.
There is a special optimization for data objects created by computational modules. Amira automatically determines whether data objects which are created by other modules are not yet saved and asks the user to do so if necessary. However, in some cases, this may not be desired, since saving the data object consumes disk space and regenerating can sometimes be nearly as fast as loading the object from disk. As an example, consider the network in figure 8.1. In this case the resample module can automatically recompute the `lobus.Resampled` data object when the network is loaded. In order

to determine whether a compute module is able to do so, the module must implement the function `int HxResample::canCreateData(HxData* data, McString& createCmd)`. This function is called whenever a network containing newly created data objects is saved and these objects have not yet been saved but are still connected to a compute module. The method should return 1 if the compute module is able to recreate that particular data object. In this case the corresponding Tcl command should be stored in the string `createCmd`. When executed, the Tcl command should return the name of the newly created data object.

Determining whether a compute module can create a data object may be tricky. Typically, it must be assured that in the time between the actual creation of the data object by the computational module and the execution of the save network command neither the parameters nor the input has changed, and that the resulting data object had not been edited.

In order to implement this behavior, most compute modules use a flag that they set when they create a data object and which they clear when the module's `update()` method is called, indicating that some input has changed. In order to check whether the data object was edited, the data object's `touchTime` variable is saved. `touchTime` is increased automatically whenever a module is edited. A typical method could look like this:

```
int HxResample::canCreateData(HxData* data, McString& createCmd)
{
    if (resultTouchTime != data->getTouchTime() ||
        parameterChanged)
            return 0;

    createCmd.printf("%s action hit; %s fire; %s getResult\n",
        getName(), getName(), getName());
    return 1;
}
```

8.4 Using version information

It is sometimes useful to know the Amira version when compiling a module. For example, a module may be capable of using features of a newer Amira version but could also working with an older version (possibly with a limited functionality). To write such-version dependent code, a special header file, `hxcore/version.h`, should be included.

It contains a define that indicates the current version of Amira coded in one number and additionally as separate defines for *major*, *minor*, and *patch* version. The following example is taken from Amira 5.4.1:

```
#define PRODUCT_VERSION 50401
#define PRODUCT_VERSION_MAJOR 5
#define PRODUCT_VERSION_MINOR 4
#define PRODUCT_VERSION_PATCH 1
```

Let's assume that Amira 5.4 has a exciting new function `foo()` that can be used in the `compute()` method of class `Bar` if available. The code could look like the following:

```
#include <hxcore/version.h>
//...

void Bar::compute()
{
//...
#if PRODUCT_VERSION >= 50400
    foo();
#else
    // do something similar
#endif
//...
}
```

Note: Older versions of Amira do not contain the file `hxcore/version.h`. To write version-dependent code with older versions, you would need create this file manually.

8.5 Troubleshooting

This section describes some frequently occurring problems and some general problem solving approaches related to Amira development.
The section is divided into two parts: Problems which may occur when compiling a new package, and problems which may occur when executing it.

8.5.1 Compile-Time Problems

Unknown identifier, strange errors: A very common problem occurring in C++ programming is the omission of necessary include statements. In Amira, most classes have their own header file (.h file) containing the class declaration. You must include the class declaration for each class that you are using in your code. When you get strange error messages that you do not understand, check whether all classes used in the neighborhood of the line the compiler complains about have their corresponding include statement.

Unresolved symbols: If the linker complains about unresolved symbols, you probably are missing a library on your link line. The Amira development wizard makes sure that the Amira kernel library and important system libraries are linked. If you are using Amira data classes, you will need to link with the corresponding package library `hxfield`, `hxcolor`, `hxsurface`, and so on. To add libraries to your link line on Unix, edit the `GNUmakefile`, find the line starting with `LIBS += ...`, and append `-l<name>`, where `<name>` is the name of the package you want to add. On Windows, use Visual Studio's project settings dialog. Details are given in Section 1.5.

Linker error LNK1000: Internal error during IncrBuildImage: This error may occur when using Microsoft Visual Studio 2008 to compile new packages. Microsoft has released a patch (KB948127) to address this issue: `https://connect.microsoft.com/VisualStudio/Downloads/DownloadDetails.aspx?DownloadID=11399`

8.5.2 Run-Time Problems

The module does not show up in the popup menu: If your module did compile, but is not visible in the popup menu of a corresponding data object, there is probably a problem with the resource file. The resource file will be copied from your package's `share/resources` directory to the directory `share/resources` in your local Amira directory. Verify that this worked. **Note:** Currently on Windows, the resource files are copied in a post link step. Therefore, if you change the resource file after linking, you must build the package again, e.g., by changing one of the source files.

If the resource file is present, the next step is to check whether it is really parsed. Add a line `echo "hello mypackage"` to the resource file. Verify that the message appears in the Amira console when Amira starts. If not, probably the environment variable `AMIRA_LOCAL` is not set correctly.

If the file is parsed, but the module still does not show up, the syntax of the rc file entry might be wrong or you specified a wrong primary data type, so that the module will appear in the menu of a different data class.

There is an entry in the pop-up menu, but the module is not created: Probably something is wrong with the shared library (the .so or .dll file). In the Amira console, type `dso verbose 1` and try to create the module again. You will see some error messages, indicating that either the dll is not found, or that it cannot be loaded (and why) or that some symbol is missing. Check whether your building mode (debug/optimize) and execution mode are the same. In particular, if you have compiled debug code you must start Amira using the `-debug` command line option or the debug executable (see Section 1.5).

A read or write routine does not work: The procedure for such problems is the same. First check whether the load function is registered. Then verify that your save-file format shows up in the file format list when saving the corresponding data object. For a load method, right click on a filename in the load-file dialog. Choose format and check whether your format appears in the list. If that is the case, you probably have a dll problem. Follow the steps above. If the library can be loaded, but the symbol cannot be found, your method may have either a wrong signature (wrong argument types) or on Windows you might have forgotten the `<PACKAGE>_API` macro. This macro indicates that the routine should be exported by the DLL.

In general, if you have problems with **unresolved** and/or **missing symbols** you should take a look at the symbols in your library. On Unix, type `nm lib/arch-*-Debug/libmypackage.so`. On Windows, type in a command shell: `dumpbin /exports bin/arch-Win32VC8-Debug/mypackage.dll`.

8.5.3 Debugging Problems

Setting breakpoints does not work: Since Amira uses shared libraries, the code of an individual package is not loaded even after the program has started. Therefore some debuggers refuse to set breakpoints in such packages or disable previously set breakpoints. To overcome this problem, first create your module and then set the breakpoint. If you want to debug a module's constructor or a read or write routine, of course, this does not work. In these cases, load the library by hand, by typing into the Amira console `dso open libmypackage.so` (if your package is called mypackage). Then set the breakpoint and create your module or load your data file.

Index

Amira
    local directory, 2, 5, 16
    root directory, 5
AMIRA_LOCAL, 9
AMIRA_ROOT, 11

AmiraMesh
    API, 49
    read routine, 52
    write routine, 51
apply button, 63

breakpoint, 11, 12, 163
build system, 23
busy cursor, 84

class hierarchy, 87
color editor, 158
Colormap, 51
colormap port, 76, 158
compiler, 3
compiling
    Unix, 11
component, 19
compose label, 39
compression, 50
compute method, 57
compute module
    adding new one, 20
    example, 55
console window, 33, 43
content type, 50
coordinate systems, 106
coordinates
    curvilinear, 97
    rectilinear, 97
    stacked, 96
    uniform, 96
create command, 160
create method, 95
curvilinear coordinates, 97

data classes, 87
database, 109, 158
debug mode, 10, 11
debugger, 11
degenerate cells, 103
demo package, 17
development wizard, 15
dialog boxes, 41
DLL, 2, 163
do-it button, 63
down stream connection, 48
dso command, 12, 163
duplicate vertices, 101
dynamic loading, 2
dynamic type checking, 48, 59

encoding, 39, 101
error dialog, 157
eval method, 105
evalReg, 59

field classes, 90
file dialog, 157
file format, 22, 29
file header, 22, 33
file name extension, 21

global objects, 157
global search, 106
gmake, 3, 11
GNUmakefile, 3, 11
graphical user interface, 3, 41

hexahedral grids, 102
HxColormap, 76
HxHexaGrid, 102
HxLabelLattice3, 84, 98
HxLattice3, 92
HxMessage, 33, 43
HxParamBundle, 108
HxPortButtonList, 82
HxPortFloatTextN, 57
HxPortIntSlider, 72
HxPortRadioBox, 78
HxTetraData, 101
HxTetraGrid, 99
HxUniformScalarField3, 60

interface, 44, 90
ITK, 64

label field, 98
link line, 162
load command, 40, 158
local Amira directory, 5, 16
local coordinates, 107
local directory, 2
local search, 106
location class, 105

MAKE_CFG, 11
material database, 109, 158
material ids, 100, 103
materials, 98, 109
McHandle, 73, 82
McStringTokenizer, 38
McVec3f, 75
message window, 157
module
  adding new one, 19
  example, 70
multiple file input, 21

non-conformal grids, 103

Open Inventor, 2, 71
OpenGL, 2, 3
overwrite dialog, 43

package, 1, 18
parallel flags, 11
parameters, 89, 108
parse method, 77
performance, 64
plot API, 80
polymorphism, 44
Pool, 157
portData, 59
PPM3D format, 31, 42
primitive data types, 92
procedural data interface, 105
progress bar, 61

Qt, 3, 41

question dialog, 157

read routine
adding new one, 21
example, 30
multiple files, 40
rectilinear coordinates, 97
register
data, 33, 39
read routine, 33
write routine, 43, 48
registry, 9, 17
regular grid, 91
renaming a package, 13
resource file, 2, 33, 43, 48, 162

save network, 158
save ports, 159
scene graph, 71
shared object, 2
smart pointer, 73, 82
SpatialData, 90
stacked coordinates, 96
storage-class specifier, 33, 43
surface, 35
patch, 38
surface field, 35

table coordinates, 107
Tcl interface, 77
Tcl library, 3
template function, 93
tetrahedral grids, 99
touch time, 160
transformations, 107
Trimesh format, 35, 44

uniform coordinates, 96
unknown identifier, 162
unresolved symbol, 162
update method, 78
upgrading to Developer Option 5, 12

Viewer, 158
Visual Studio
debug code, 10
release code, 10

warning dialog, 157
work area, 61, 157
world coordinates, 106
write routine
adding new one, 22
example, 41