University of Oslo
Department of Informatics

Interactive manipulation of three-dimensional images

Kristoffer Gleditsch

Cand Scient Thesis

April 27, 2003

Abstract

This thesis is about the design and implementation of an application for editing voxel data. We are primarily working with MRI and CT data in a medical setting, but neither the application nor the problem itself is specific to this field. Most, but not all, of the functionality is geared toward segmented datasets. In addition to being usable as an application in itself, our program should provide a prototyping framework for others who want to test algorithms and tools on three-dimensional datasets. Because of this, we have designed and documented a plugin API, and implemented a number of plugins performing different operations on the dataset.

The thesis touches on a lot of problems and choices that were made while implementing the application, from the overall application design down to our choice of libraries and tools. The programming language is C++. We made a choice to rely on libraries where we could, and so we make use of Blitz++, ImageMagick, Autotools, Qt, OpenGL and Open Inventor.

The finished application's capabilities are outlined, and the design, tool choices and usability of the application are discussed.

Preface

This thesis is submitted to the Department of Informatics at the University of Oslo as part of a candidatus scientiarum (cand. scient.) degree. It was written as a part of the Dr. Jekyll project in cooperation with the Interventional Centre at the National Hospital of Norway.

This is intended to be a discussion and elaboration of various aspects of the Dr. Jekyll project. The primary focus is the design and implementation of the Dr. Jekyll application.

Thanks to

Johan Simon Seland, the other half of the project team.

Lars Aurdal, my external advisor at the Interventional Centre, for never being afraid of a good digression.

Knut Mørken, my internal advisor at the Department of Informatics, for helpfulness and good advice.

The Interventional Centre at Rikshospitalet University Hospital, for providing an excellent working environment.

LRDE – Laboratoire de Recherche et de Développement de l'EPITA, Paris, France, for their hospitality. I am particularly grateful to Thierry Géraud and Daniela Becker for making it happen, to Olivier Ricou and Didier Verna for putting up with us in their office, and to Akim Demaille for lots of useful input.

Google, for finding what I'm looking for.

Everyone out there who created and shared with me the software I base my work and play on. They are too many to count, but they deserve thanks nonetheless.

Siri Spjelkavik for lots of feedback, but most of all for her smile.

Contents

List of Figures

Chapter 1

Introduction

1.1 Data

The concept of *data* is central both in this thesis and in the application developed as a part of it. In this section we will try to explain what data is, where it comes from, and why it is so important.

The human eyesight is a very complex and fascinating sensory system, but it is not without limitations. One of the biggest problems is that the reception and interpretation of reflected light is unable to give us any information about what lies behind objects that do not allow light to pass through them. If we need to find out what is inside or behind such non-transparent objects, we need to cut them open or remove them. If the object blocking the way is not expendable, or impossible to remove, normal eyesight has little chance of finding out what is behind it.

The discovery of x-rays by Wilhelm Conrad Röntgen in 1895 was a huge step forward: Suddenly it became possible to take pictures of previously invisible objects and phenomena. Following his discovery, the last century has seen much progress in the imaging and visualization areas, but the images we are able to take today are still seldom as good as we would like them to be.

One of the fields where better imaging techniques are always sought for is medicine. The Dr. Jekyll project originated in a medical setting, which is reflected in our examples and illustrations. However, there is very little in the application, or in the problem it tries to solve, that is specific to the medical field.

1.2 Images and image sources

In a hospital today, doctors have access to several different kinds of imaging machinery producing tree-dimensional images of the body. The two most common imaging methods are CT (*computed tomography*) and MRI (*magnetic resonance imaging*). A CT scanner uses x-rays to generate its images, while an MRI scanner uses magnetic resonance. Although both kinds of machinery produce data volumes, these volumes are not similar, and the two methods are suited for taking pictures of different phenomena. Also, physicians are usually restrictive when it comes to CT, since there is a significant amount of radiation involved.

Three-dimensional images are often referred to as *volumes*. In our context, the word *image* is a very generic term; it can have any number of dimensions. Volume, on the other hand, is used as a specific term, denoting an image with exactly three dimensions.

An image consists of a finite number of discrete data points, each containing information about a distinct position in the image. While a generic image consists of equally generic *pixels* (*picture elements*), a volume is made up of *voxels* (*volume elements*). Exactly how these pixels or voxels are laid out can vary, but we have chosen to limit ourselves to images organized as regular grids. This means that our volumes have three dimensions, and a voxel has three coordinates – X, Y and Z – positioning it in the volume, as well as an extent in all three dimensions.

1.3 post-processing – where we come in

After an image has been fetched from whatever machinery produced it, it is often put through a number of operations making it more suited for its intended purpose. This stage is called *post-processing*. The exact kind of operations performed on the image depends on the origin, the characteristics and the purpose of the data.

One such post-processing treatment is *segmentation*. When an image is segmented, the pixels are divided into groups based on some property defined by the algorithm. Each group is assigned a unique value n, and every pixel belonging to this group is set to the value n in the output image. In other words, a segmented image is an image where the only meaning contained in the value of each pixel is that it shares some specific trait with the other pixels carrying the same value. This means that a pixel with the value 5 and one with the value 6 are no more closely related than a pair with the values 2 and 98. The different values in a segmented image are often referred to as *labels*.

A segmentation is often followed by a *classification*. A classified image is

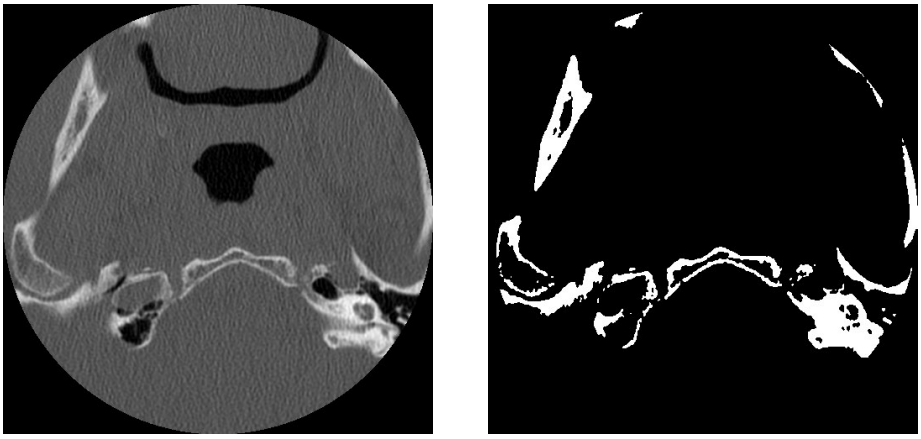

Figure 1.1: Example of a segmentation. On the left is a slice from a raw CT dataset of a neck. On the right we see the same slice after a binary segmentation, i. e. a segmentation using only two different labels.

a segmented image where each label has been given a physical interpretation. Examples of common label interpretations are “background” and (in a medical setting) “bone”, “liver”, and so forth. A classified image can contain more labels than the segmented image did, since two different components sharing the same label in the segmented image may be interpreted (classified) as belonging to different physical groups.

Figure 1.1 shows a segmentation of a neck dataset. This segmentation has used two different labels, setting each voxel in the image to one of those two. In a classification step, the simplest interpretation of these two labels could be “bone” and “not bone” or “background”. A more sophisticated classification would probably give separate labels to the different neck bones occurring in the image.

Unfortunately, segmented image datasets often contain errors. Before they can be used for their intended purpose, whether it is visualization, calculations or something else, they need some amount of “touching up” in order to be usable. The reason for such errors can be that the imaging equipment does not produce perfect images, introducing some amount of artifacts and disturbances. It can also be that the segmentation algorithm is not always able to make the right guess when it is trying to determine what label a voxel should belong to.

An example of imperfect segmentation is shown in figure 1.2. On the top it shows a slice from a raw CT volume of the abdomen. On the bottom, we see the same slice from the segmented version of the volume. On this segmented volume, we have run a three-dimensional *connected component analysis*. All voxels with the same value making up an unbroken path through the image belong to the same connected component. For details on

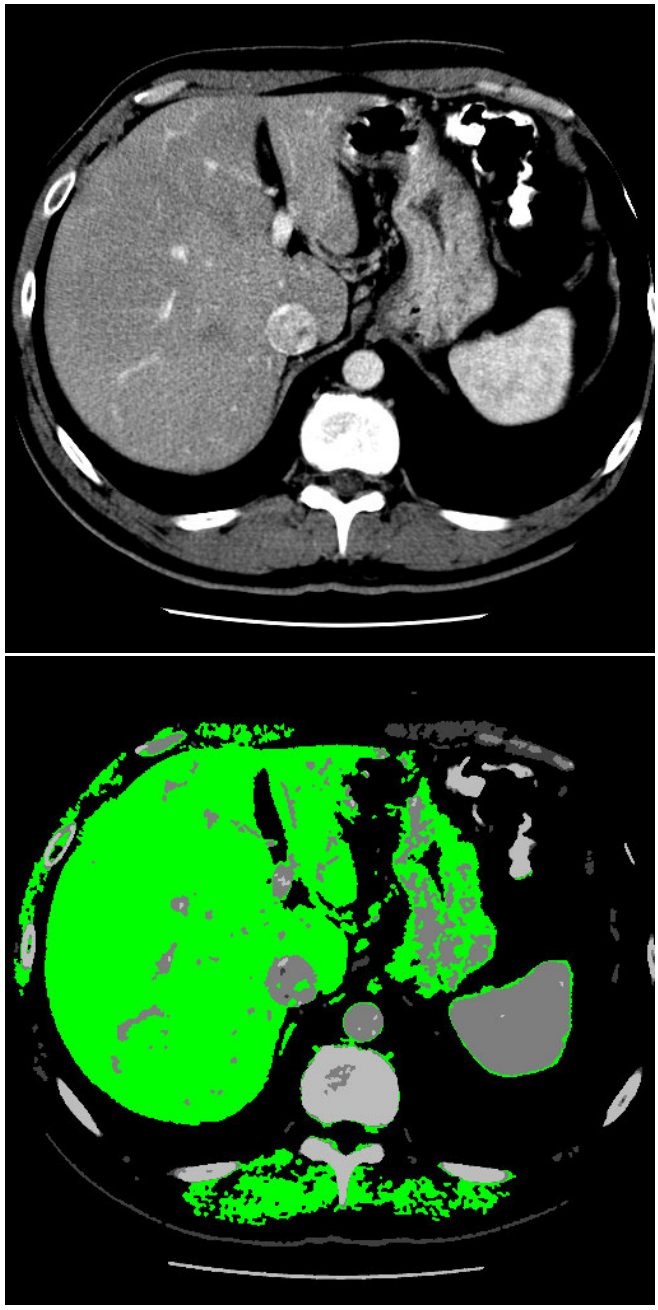

Figure 1.2: On the top is a slice from a raw CT volume of the abdomen. The large organ on the left side is the liver. Below is the same slice from the segmented version of the volume, with the liver component highlighted in green. Note that the connected component analysis was performed on the entire volume, so not all the green parts are connected in this particular slice.

this analysis and its implementation, see Johans Seland's thesis [72]. The component the liver belongs to is highlighted in green, and it is obvious that this component contains a lot more than just the liver.

Fortunately, the editing necessary to make this kind of volumes good enough for most purposes is usually not very difficult to perform by hand. Since volumes are often stored as a stack of normal image files, each containing a two-dimensional slice of the cube, it is possible to use conventional image manipulation programs for this editing. Unfortunately, it is not very convenient. Figure 1.3 shows a screenshot of the image manipulation program GIMP [9], where we have opened the 74 slices belonging to the segmented neck dataset shown in figure 1.1. It is fully possible to use this program to remove artifacts from the volume, or smooth the contours in order to make it look better in a tree-dimensional visualization, but it can hardly be called practical. It is also very difficult to visualize and manipulate contours and connections that are not nicely oriented normal to the slicing plane. This, in short, is the problem we are trying to solve: We want a convenient way of opening, viewing and manipulating three-dimensional datasets.

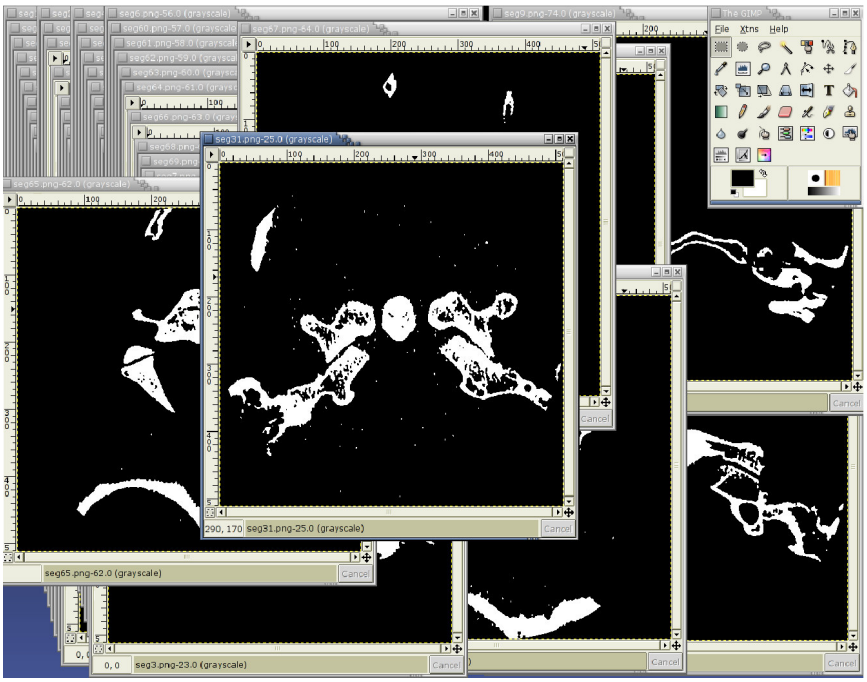

Figure 1.3: Trying to use GIMP on 74 neck CT slices

A thing to remember about GIMP and similar programs is that their built-in tools, although only two-dimensional, are mature and powerful.

Although we may succeed in building an application that handles three dimensions in a more elegant way, we can not hope to implement this kind of tools within the limits of our project.

This thesis is about **Dr. Jekyll**, the application we created in an effort to make this manual post-processing easier. It got this name because it is a logical continuation of **MrHyde**, a segmentation tool created by our supervisor Lars Aurdal in connection with his Ph.D. thesis [51].

The Dr. Jekyll application is the result of a team effort by two developers: Johan Seland and Kristoffer Gleditsch. Both have put a lot of effort into the development of Dr. Jekyll, but with different focus: Gleditsch has been working with the overall design of the application and the plugin interface. Seland [72] has been responsible for the post-processing tools and their underlying algorithms, and has made most of the plugins.

The result is one program but two different theses. They are separate works, but closely related and sometimes overlapping. Where the views and opinions of the authors differ, the theses will reflect this.

1.4 Goals

At the beginning of the project, we stated three high-level goals for Dr. Jekyll. A set of technical goals is stated by Seland in [72]. These two lists of goals apply on different abstraction levels, and should be perceived as complementing rather than conflicting.

Interface: Dr. Jekyll should provide a usable and intuitive interface for visualization and editing of three-dimensional image sets. This may seem like an obvious requirement, but it is nonetheless a challenge.

Visualization: Dr. Jekyll should provide a three-dimensional visualization of the image sets. This visualization should be interactive, displaying changes and giving feedback immediately.

Plugin interface: Dr. Jekyll should provide a clear and well documented API (*application programming interface*) to self-contained pieces of code called *plugins*. These plugins can be written by other people, using our application as a prototyping framework for their own algorithms.

As a consequence of the wish to be a framework usable by others, Dr. Jekyll should also be available under an open source license.

1.5 Thesis outline

This thesis is divided into seven chapters:

Chapter 1, this one, is the introduction. Here we explain where our data comes from, what we want to do with it and why, and the overall goals of the project. It ends with a list of related works.

Chapter 2 is a more detailed outline of what functionality we want the program to contain.

Chapter 3 discusses the tools and techniques used in the implementation.

Chapter 4 explains the design and structure of the program, explores some issues concerning the internal information flow in the program, and explains the plugin interface.

Chapter 5 outlines what we have accomplished, and discusses the capabilities of Dr. Jekyll both as an end-user application and as a development framework.

Chapter 6 is an evaluation of the choices we made in the previous chapters, explaining which choices we are happy and unhappy with and why.

Chapter 7 is a short conclusion.

Chapter 8 ends the thesis with a list of possible areas of future work.

There is also a short appendix explaining the license of Dr. Jekyll and its background.

1.6 Related works

1.6.1 Academic works

MPR Edit [67], created at the Johns Hopkins Medical Institutions, offered an interface for manipulating three-dimensional data as early as 1991. One of the main features of MPR Edit is the ability to define three-dimensional geometric structures, and then use them as a mask on the image. This makes it easy to (for example) remove structures that are obscuring the area of interest.

At the time [67] was written, three-dimensional interactive visualization of the datasets was a lot harder than today, primarily due to smaller and slower computers. It is mentioned, however, that this was a feature that should become possible within relatively few years.

Vrije Universiteit in Brussels has created an editor for 3D medical volume images [54]. They use splines as a means of making it easier for the user to create and change contours while still keeping the volume

consistent in all dimensions. The user interface shows an X, Y and Z-normal projection as well as a wireframe projection of the volume. This way, users editing a contour can see the result in the wireframe without delay. Our goal of a three-dimensional visualization giving immediate visual feedback was partly inspired by this.

1.6.2 Software projects

OpenDX – Open Visualization Data Explorer [26] from IBM Research is a powerful data visualization and exploration tool. It has been an open source project since 1999, released under the IBM Public License. It has a very good data importing module, and a broad variety of functions and options, most of which seem geared toward viewing data, as opposed to editing.

OpenDX is available from the web pages under the IBM Public License.

Viz [43], written by Per Øyvind Hvidsten, is a visualization tool for 3D datasets using the volume rendering approach. It offers several nice features for data exploration, including thresholding, assignment of color and transparency to different intensities, rotation, zooming, cutting, and so forth. Viz is a very specialized volume viewing program, and when it comes to operations like rotation, zooming, etc., the feature-rich interface is perhaps superior to the viewing interface of e.g. OpenDX. Viz can be downloaded from the home page under a non-commercial license.

3D-DOCTOR [1, 80] is developed by Able Software in Massachusetts, USA. It has an impressive feature list, including a rich and powerful interface for analyzing and visualizing data as well as generating different kinds of models and animations. It can be customized and extended using an embedded scripting language (3DBasic), but the application itself does not seem to be directly extendable. 3D-DOCTOR is a commercial application, but a demo edition is available from the web pages.

MRIVIEW [15], developed at the Los Alamos National Laboratory, is specialized for viewing and editing MRI head (brain) data. It is developed using the commercial language IDL from Research Systems [70]. MRIVIEW offers both two- and three-dimensional views of the data, as well as segmentation tools and color table manipulation.

MRIVIEW is available for free, but without source code. Since the IDL language is commercial, a license for this is needed to use the software.

OpenQVis [71] is an open source project aiming to perform high quality volume rendering on normal desktop computers. It is a joint project between the Computer Graphics Group of the University of Erlangen Nuremberg and the VIS Group at the University of Stuttgart, Germany.

OpenQVis seems to be a very interesting viewer application, especially since it is released under a license similar to our own.

MedVed [20] is a medical voxel editor developed as a student project for the Norwegian company SimSurgery AS [35]. Like Dr. Jekyll, it is based on C++ and Qt, but it uses a radically different approach to the voxel storage data structures. The voxels are not stored in a regular array, but in a layered structure derived from the surfaces in the image. MedVed is a commercial product.

With all these software projects available, it is natural to ask why we chose to implement our own application from scratch instead of extending an already existing program. OpenQVis in particular would be a very interesting candidate for this. There were several reasons for our choice to start from scratch:

- The most interesting candidates for such an "adoption" were not available, or had not yet matured enough to be usable by us, when we started our project in the autumn of 2001.

  The increased availability of such applications the last two years may be connected to the increased availability of powerful graphics hardware for normal computers. Writing an application like Dr. Jekyll is a lot easier in 2003 than it was five years before.

- One of the goals of Dr. Jekyll was to be available to other developers as a prototyping framework. Several of the applications mentioned above are available, but only under licenses that would make it difficult for us to change and redistribute it the way we would like.

- In order to change and experiment with an application, you have to know both the application and the tools it relies on thoroughly. It is not immediately obvious that it would be easier to learn the internals of e.g. OpenDX well enough to make changes to it, than it was to implement our own program from scratch.

  This argument applies to our own work as well: We have written an application that we want other people to extend, but in order to do that, they will have to learn something about how it works. This is the reason why we have put a lot of work into isolating the plugins and documenting their interface: The user should not have to learn or understand the internals of the program in order to extend it. The plugin interface is discussed in more depth in section 4.3.

Chapter 2

Desired functionality

In this chapter, we will take a closer look at what we want the application to do. We will go through the different things the users want to do, explaining how the application will work. In other words, this is a discussion about how we intend to fulfill the goals from section 1.4.

Section 2.1 is about reading and writing datasets to and from files.

Section 2.2 discusses the various ways of visualizing the cubes.

Section 2.3 outlines various ways of changing the dataset.

Section 2.4 talks about letting the user add new ways of changing the data.

2.1 Reading, writing and storing volumes

In order to visualize and manipulate data, the first thing the application needs to do is read it in from file. One way of storing a volume is by slicing it up, and storing each slice as a normal image file. Our application should let the user specify such a list of image files, either by choosing all the files directly using a GUI, or by using *format strings*. Format strings are well-known to most C programmers, as they are the basis of the `printf` family of functions. They let the user specify a string containing special markers, which are then replaced with values supplied by the user. For a more thorough explanation of format strings, see [63]. The program should also be able to save volumes back to disk, either to a new series of names or by overwriting the old filenames.

Datasets come in a variety of bitdepths, from binary data (in principle requiring only a single bit to store each pixel) to 16bpp (*bits per pixel*) and more. The program should be able to operate on a variety of bitdepths transparently.

2.2 Visualizing the data

After the data is read in from file, we need to display it on the screen. The basic problem with visualizing a volume is that the data has three dimensions while our means of output – the computer screen – only has two. There are different ways of dealing with this, so the application should offer several different kinds of visualizations.

2.2.1 Two-dimensional projections

The slicing approach used for storing volumes works for visualizing them as well: A relatively common way of visualizing volume data is by projecting two-dimensional slices through the cube. Since these projections can only display a small subset of the cube at a time, they will need to move around in order to give the user an overview. This can be done by defining a single coordinate as a focus point, and then letting the two-dimensional projections move through the cube so that they always display the plane where this focus point is placed. By having the focus point follow the mouse pointer around inside the visualization windows, the user can "navigate" through the volume.

Dr. Jekyll should offer this kind of two-dimensional visualization. It is not very complicated, but in order to be responsive, the delay from the user moving the focus point to the projections being refreshed should be small. This means that some work may have to be put into the extraction of the slices from the cube and drawing them on the screen, in order to make it computationally efficient.

2.2.2 Three-dimensional visualizations

Even though two-dimensional slice projections can be effective, a three-dimensional projection is often necessary to get an overview of the volume, and to make it possible to comprehend complex three-dimensional relations between different parts of the volume. Since the computer screen is flat, it is impossible to make a real three-dimensional image. However, through clever use of artificial light and shadow, it is possible to project an image onto the screen in a way that comes very close to the feeling of inspecting the real object. Such visualizations can be performed in several different ways, two of the main ones being *volume rendering* and *surface rendering*. The main conceptual difference is that a volume rendering uses the entire dataset when generating the projection, while the surface rendering first extracts the object surfaces from the dataset, and then makes its projection using those.

When working with volume data, some kind of volume visualization can be very useful, and Dr. Jekyll should offer at least one such visualization. This projection would allow the user to do things like rotating the volume in order to view it from different angles, or zooming in and out in order to see it close up or far away. As we mention in the list of goals in section 1.4, this visualization should be interactive, meaning that changes made to the volume show up immediately.

2.2.3 Other visualization-related functionality

In addition to the projections we want to display, there are some other things we would like to be able to do with the visualization windows of such an application. One of them is to *connect* two or more datasets. The idea behind this is that a user may open two cubes which originally come from the same dataset – for instance one raw cube, and one segmented cube derived from it. When looking at these two datasets in the two-dimensional projections, the user should be able to connect the focus points of the two cubes, so that they are always identical. This way it is possible to navigate around one of the volumes, and always have the projections of the other cube display the same slices.

2.3 Changing the dataset

In addition to letting the user look at the dataset, we want to let the user change it. This means that the program must be able to handle user input, and have a variety of components (different tools) able to translate that input into changes to the dataset.

2.3.1 Changing components

One interesting kind of tools is one operating on selected components. After a volume has been read in from file, the program could perform a connected component analysis on it. The user would then be able to use the mouse to select one or more such components in the image, and perform some operation on those. Examples of such operations would be reclassification (setting all the voxels to a different value) or different morphological operations (like erosion, which will “wear” off the edges of the component).

2.3.2 Changing labels

Another kind of tools could operate on the selected labels. The operations could be similar to the ones operating on components, but instead of being

limited to the selected components they would be limited to all occurrences of the selected label(s).

2.3.3 Changing voxels

A third kind of operations is the straightforward pixel manipulation, i. e. tools for editing raw pixel data. The basic, but very useful, example of this is pixel painting with the mouse: The user chooses a value, and moves the mouse around in the visualization window to mark the coordinates that should be set to the given value.

It should also be possible to combine these three kinds of tools. For instance, the user should be able to select a component and then perform a pixel painting limited to that component.

2.3.4 Tools in two and three dimensions

There are a lot of two and three-dimensional tools we would like to implement as part of our application's interface. As we mentioned in the introductory chapter, traditional image processing tools like Adobe Photoshop [2] and GIMP [9] have implemented sophisticated tools for operating on two-dimensional images. We would like to be able to offer similarly powerful tools for three dimensions, but we do not think that we will be able to implement this within the limitations of our project. We hope, however, that the resulting application will be usable as prototyping platform for others wanting to experiment with such tools; this is further discussed in section 2.4.

2.4 New ways of changing the dataset

In addition to the ones we implement, the user should be able to add his own tools if he wants to. These tools should be very free to define their own semantics, i. e. the user should be able to add anything from new voxel painting methods to advanced image analysis algorithms. We call these add-on components *plugins*. It should be possible to write such plugins for our application without knowing the program's internals, so there should be a clearly defined and documented interface. The amount of code needing to be changed when a new plugin is added should be minimized, which means that references to such plugins should be not be hard coded into the rest of the program more than strictly necessary.

Chapter 3

Tools and techniques

This chapter discusses the techniques, tools and libraries that were used in Dr. Jekyll.

Section 3.1 is about our choice of tools and libraries, going through the alternatives that were considered and the arguments for and against each. It ends with an explanation of one of our most important tools – the Qt signal/slot mechanism.

Section 3.2 talks about more general techniques used in the development of Dr. Jekyll.

Section 3.3 is a discussion about data storage. This is an essential question in an application like ours, and it is a question of how to do it just as much as it is a question of what tool to use for the job. I have chosen to let this section deal with both these aspects, even though some of it could just as well have been discussed in section 3.1.

3.1 Tools

3.1.1 Programming language

When sitting down with the intent of writing an application from scratch, one of the first decisions to be made is which programming language to use. The main issues we considered when deciding on a programming language were:

Speed: Dr. Jekyll is a data-intensive application; it reads, handles, displays and writes relatively large amounts of data. As we wanted our application to be interactive (also known as “snappy”), we needed a language where these operations could be done with as little overhead as possible.

Libraries: The use of software libraries has saved us a lot of work through the implementation of Dr. Jekyll. We use libraries for a wide variety of purposes: Reading and writing images, data structures, user interface, visualization. The amount of available software libraries within our problem domain was an important parameter.

Portability: Even though most of our development work is done on a single platform, we wanted Dr. Jekyll to work on as many different platforms as possible.

Abstraction: If speed was the only thing that mattered, writing the entire application directly in assembly language would be an attractive choice. Besides not being very portable, programming in such low level languages is generally considered to be a slow and error-prone process. In order to save programmer time, most people want a language with a reasonable level of abstraction.

Sadly, this is a direct contradiction of the first point in this list. The price of abstraction is more run-time overhead, so we have to compromise.

The list above is not exhaustive. For example, we knew that we would need some way of handling the fact that our input cubes contain very different amounts of data per pixel. Some cubes are binary, meaning that each pixel is either on or off (often represented as 0 and 1), and these could be stored using only a single bit per pixel. Other images need a byte per pixel (256 different values), and yet others need even more. The templates feature currently found in C++ and Ada [59] (and, judging by [48], likely to be adopted by Java as well) is a very convenient way of handling this variation of data types in otherwise similar objects.

Candidates

This is a short summary of the primary candidates:

C is a compiled, statically typed and standardized [61] language. It is one of the ancestors of C++, and fundamental to the Unix family of operating systems. Its main drawback in our context was the low abstraction level and lack of object orientation. It is as fast as C++ for data intensive work, and has a large number of libraries available.

Java: Developed by Sun Microsystems, Java is a statically typed object oriented language. It is not compiled to binary (machine specific) code, but to so-called bytecode. This bytecode is then run inside a JVM (*java*

virtual machine), which means the same bytecode can run on any platform with a JVM: "compile once, run anywhere". In principle, this is a huge step forward for cross-platform development.

However, Java has a more complex run-time environment than many programs compiled all the way down to machine code. This means it will also have more run-time overhead, which may make it less well suited for data-intensive applications. Nonetheless, volume rendering implementations in Java do exist; [73] from Nanyang Technological University, Singapore and [66] from VRVis Research Center in Vienna, Austria, both describe such implementations. While the first one does not seem to attain speeds that can be called interactive, the second one claims to do so.

C++ is a binary compiled, statically typed, "multi-paradigm" [75] and standardized [60] programming language. It offers the possibility of object oriented programming, and has a comprehensive tool chest in the Standard Template Library, STL [62, 36]. C++ compilers exist for most major computing platforms today, and since the language is standardized it is (at least theoretically) possible to write portable code just by adhering to the standard.

C++ is in wide use, and well tested in industrial settings. There are mature compilers and toolkits available, and the documentation is relatively good.

Two layers, consisting of one modern scripting language and one traditional binary compiled language. Script languages like Perl and Python offer facilities for accessing methods written in languages like C and C++. Although briefly considered at the start of the project, this alternative was not given serious consideration until much later. If we were starting the project over again from scratch, we believe this approach would be the strongest competitor to C++ as programming language.

This approach allows the programmers to use high-level programming languages for the overall application event flow and user interaction. At the same time, the low level data intensive parts can be written in binary compiled languages that are utilized by the script layer on top of it. This would give the programmer flexibility and abstraction on the top and speed on the bottom.

An additional advantage of using interpreted languages is that they do not require recompilation every time they are changed. Presumably, this would reduce the time spent waiting between each test cycle when developing.

The choice eventually fell on C++. Properly written C++ is portable enough for our use, and it seemed to offer the best compromise between

abstraction and speed.

3.1.2 About libraries

Writing code that can be reused by others over and over has been a goal of programmers since the very early days of computing. Exactly how to realize this goal has been a subject of discussion and research for just as long. The concept of having different components hiding implementation details from each other, outlined in [68], is over 30 years old. Although the tools and environments have changed radically since then, most modern software libraries do more or less exactly this: They implement and encapsulate different kinds of functionality, so the user won't have to care (or even know) about it.

When designing and implementing Dr. Jekyll, we knew from the start that our resources were limited. We had no wish to duplicate functionality already implemented by others ("reinventing the wheel"), and as a consequence we decided to use external libraries where we could. In hindsight, there is no doubt that Dr. Jekyll would have been much more limited in its functionality if we had not done this.

This functionality comes at a price, however: In order to compile and install Dr. Jekyll, the user must first install a handful of other libraries. In some cases, many or all of these dependencies come bundled with the operating system. If they don't, the user has to compile and install them himself.

3.1.3 GUI toolkit

Obviously, an interactive application needs some way of interacting with its user. For programs of a graphical nature, a GUI (*graphical user interface*) is generally considered a very convenient way of providing user input/output.

As we use Unix as our primary development platform, we could have chosen to program the X window system [45] directly. The X interface operates on a low level, hiding very little information from the developer. This has a tendency to make the development of complex interfaces using the X layer directly a slow, cumbersome and error-prone process. It would also limit the portability of the program to Unix platforms.

Most people developing GUI-based applications today use some kind of toolkit. By offering prefabricated GUI components like sliders, buttons, labels and menus, these toolkits enable the programmer to work at a much higher level of abstraction.

Candidates

Motif [22] is one of the GUI toolkit veterans of the Unix world. It is the base of the Common Desktop Environment (CDE), and has been used for a wealth of applications throughout the years.

Tk is another toolkit classic. It is usually, but not necessarily, used together with the Tcl programming language [38].

GTK+ [12] originated as a toolkit used by the GIMP ("Gnu Image Manipulation Program") application [9]. Today it is a very common toolkit on the Linux platform, and the GNOME [6] desktop environment is based on it. It is ported to several platforms, and has bindings to a variety of languages.

GTK+ is licensed under the GNU Lesser General Public License, LGPL [47], which gives the developers using it full freedom of choice with regard to the licensing of their applications.

Qt is made by the Norwegian company Trolltech [41]. It is the basis of the K Desktop Environment, KDE [17], which is the main alternative to GNOME as desktop environment on Linux today.

In addition to being a commercial product, it is available as free software under either the QPL [31] or the GPL [46]. The developer is free to choose which license to use. When developing applications using a Qt installation under either of these licenses, the source code of the application must be made available to the users who receive the binary.

One of the main selling points of Qt is multi-platform development. With Qt 3.1, the same code can be compiled and run on most versions of Microsoft Windows, numerous Unix variants using X, and Mac OS X. This is a big advantage for developers who do not want to be limited to a single platform by their choice of tools. Qt is also known for having very good documentation.

wxWindows [44] is another GUI toolkit available for developers wanting to create multi-platform software. It is actually a set of GUI libraries, one for each platform, sharing a common API. Like Qt, this makes it possible to compile and run the same code on all the supported platforms.

We chose to use Qt. The primary reasons for this were the cross-platform capabilities and the good documentation. When developing applications on top of unfamiliar frameworks, good documentation is priceless.

3.1.4 Encoding and decoding image files

When storing volume images to disk, a common approach is to divide it into slices one voxel thick, and storing each slice as a normal image file. The main advantage of this approach is that normal programs can read in and manipulate the volume, although on a slice-by-slice basis, and that the format encoding and decoding can be delegated into normal libraries. Since the volume is not contained in a single file, copying and moving it around becomes a bit more work.

In a single-file volume image format, it would be much easier to standardize the format of meta information. Most normal image formats allow the creator of an image to add free text attributes as meta information, but there is to our knowledge no standardized format for storing such information about volumes.

Candidates

Reference libraries: Many of the image formats in popular use today have a corresponding C library, which contains the functionality necessary for encoding and decoding image files in the given format. In some cases, these libraries are intended to provide a reference implementation. An example of this is the libpng library [24] for manipulating the PNG format [29]. In other cases, the library has a more informal connection to the format specification. An example of this is the libjpeg library [14] for reading and writing JPEG images [16].

Using such libraries would mean using – and depending on – one library for each image format we want to support. This would increase the number of libraries we would need to learn, as well as the number of dependencies the user would need to install.

The main argument against using these libraries is very similar to the argument against using the X library directly for the GUI in section 3.1.3. Most of these libraries operate on a lower abstraction level than we want to work on. We would prefer a high-level library, which hides as much complexity from us as possible.

Qt: The Qt GUI toolkit [41] includes functionality for dealing with images on a convenient abstraction layer. Qt's `QImage` and `QPixmap` classes contain code for reading and writing image files, as well as different image display mechanisms.

Since we have already picked Qt for our user interface, using these classes for image handling would allow us to reduce the number of external libraries we rely on. Besides reducing the number of different interfaces (and paradigms) the developers have to work with, this

would also mean one less dependency the user needs to install on his computer.

The biggest limitation of these classes is that the routines for decoding and encoding the image formats do not handle a grayscale bitdepth of more than 8 bits, even if the format itself – like PNG [29] – has no problems with 16 bit grayscale bitdepth. CT and MRI images are always grayscale, and often have a colordepth of more than 8 bits, which makes this limitation a problem.

ImageMagick [13] is a collection of tools and libraries for manipulating images in a wide variety of formats. It contains tools for image manipulation and display, among them the well-known Unix applications `display` and `convert`, as well as image manipulation library bindings for a number of languages. Among these language bindings are Magick++ [18], the C++ API to the core ImageMagick library.

Magick++ offers an interface with a reasonably high abstraction level. For instance, it transparently deduces the image format from the filename given by the user. If the user enters the filenames "image1.jpg" and "image2.png", the library understands that it is dealing with one JPEG and one PNG image. The application itself doesn't need to care or know about this at all, which is a big advantage.

Since ImageMagick is widely used, at least on the Unix platform, it is a reasonable assumption that it will support new image formats fairly quickly.

Among the drawbacks of Magick++ is the fact that it scales all pixel values to a `double` data type with range [0,1]. This means the application using it has to scale it back into the desired range. It also seems to be noticeably slower than the Qt classes when reading and writing files.

We chose to use ImageMagick for the image file reading and writing, primarily because of the problems with 16 bit PNG images and Qt.

3.1.5 Visualization toolkit

In addition to making buttons and menus, we need a way to display the data themselves on the computer screen. However, contrary to many of the other tool choices we have made, the different alternatives in this list are not exclusive. It is often desirable to have different kinds of visualizations in the same application, and the different toolkits and libraries listed here do not solve the exact same problem. For example, of the five toolkits mentioned below, four of them have at some time been used in Dr. Jekyll.

When the user moves the attention point around, it is important that the visualization windows are updated with as little delay as possible. If we spend a lot of time fetching the application from the data structure and displaying it on the screen, the feeling of interactivity will degrade quickly. Because of this, performance is more of an issue here than most other places.

Candidates

Qt [41] has built-in functionality for displaying two-dimensional images directly from `QImage`/`QPixmap` objects. As we mentioned when considering Qt in section 3.1.4, reducing the number of libraries would be an advantage. Qt does not have any native functionality for handling three-dimensional images.

The earliest versions of Dr. Jekyll used `QImage` for displaying the two-dimensional projections, but we later switched to OpenGL because of performance problems.

OpenGL [27] is a hardware-independent API for two and three-dimensional computer graphics programming, originally developed by SGI. The specification and reference implementation was made available under open licenses, and today it is a very widely adopted API for hardware accelerated three-dimensional graphics.

As a graphics language, OpenGL operates on a low level of abstraction. It is designed to be hardware independent, but it is low-level enough for this sentence to appear on the SGI OpenGL overview web page [34]:

> *A low-level, vendor-neutral software interface, the OpenGL API has often been called the "assembler language" of computer graphics.*

An important feature of OpenGL is that the programmer manipulates the geometric model, not the image on the screen. The image is generated by the driver layer as a two-dimensional projection of this model. This means the same model (the same source code) may not look exactly the same on different OpenGL platforms. In practice, though, this is seldom a problem.

Dr. Jekyll version 1.0 uses raw OpenGL for the two-dimensional image windows.

Open Inventor [25] is an abstraction layer on top of OpenGL. Like OpenGL itself, it was originally designed by SGI and later released under an open license. SGI is no longer developing the Inventor API, but there

are at least two other companies developing and maintaining their own implementations: The French TGS [40] and the Norwegian Systems In Motion, SIM [37].

Version 1.0 of Dr. Jekyll uses Open Inventor in the `Image3dVolwidget`. We used Coin [5], which is the implementation made by Systems In Motion, as well as their SimVoleon library to display a volume rendering of the dataset.

SIM provides classes for interfacing Open Inventor/Coin with several different GUI frameworks, including Qt and GTK+. This makes it very easy to combine the visualization and GUI libraries in one application.

VTK [42] is an open source toolkit for three-dimensional computer graphics, image processing and visualization. It consists of a C++ core, but has interface layers/bindings enabling it to be used from several other languages as well.

VTK is a powerful and general tool. Unfortunately, it can not be plugged directly into Qt. Several third-party "glue classes" exist, providing compatibility layers for using VTK in a Qt application, but not all of them seem to be as mature as one would wish. VTK is a visualization toolkit, which means it has good mechanisms for generating visualizations from raw data. With Open Inventor, we have to do this transformation ourselves.

Although it has been briefly tested, version 1.0 of Dr. Jekyll does not include any visualization windows based on VTK. This is something we definitely would have taken a closer look at if we had more time.

DirectX [21] is an alternative to OpenGL on the Microsoft Windows platform. Because our main development platform was Linux/Unix, DirectX was never considered a realistic candidate for use as visualization library in Dr. Jekyll. It is mentioned here primarily for the sake of completeness.

3.1.6 Build system

As of version 1.0, Dr. Jekyll consists of over 50 different source files that need to be compiled and linked. In addition, there are over 60 different header files which are included into the source files in the compilation process. In a program consisting of a single source file, it is not a big problem for the user to invoke the compiler directly. When the number of files grows, however, this is not very practical,

As if keeping track of all the files belonging to our application wasn't difficult enough, there is also the fact that the procedure for compiling a program is not identical on different platforms. In other words, it is not

enough to write portable C++ code in order for the application to be truly portable; it is also necessary to have a portable build system.

All of the build systems we considered work by generating input files to the venerable **make** utility [11]. Make is an application that takes as input one or more *Makefiles*, listing the input and output files and how to produce the second from the first. From these files, it can keep track of which files need to be regenerated based on modification timestamps, and regenerate only those in order to produce the final result.

Candidates

Autotools: The traditional way of handling this process, especially in the Unix sphere, is the set of application called **Autotools**. Autotools is a nickname for three separate, but closely related, applications: **autoconf** [7], **automake** [8] and **libtool** [10].

Autotools has the advantage of being very widely used on the Unix platform. This means that there is a wealth of documentation and examples available, and the chances are small of encountering a problem no one has already solved. It is also a very general tool, made for handling the build process independently of the frameworks, libraries or platforms being used.

The biggest disadvantage of autotools is the steep learning curve for first-time users. Although the documentation is good, the syntactic and semantic rules of the configuration files are not very intuitive. However, this is (in theory) a problem nobody should encounter more than once.

qmake [32] is a substitute for autotools. It is developed by Trolltech and distributed together with Qt. qmake has the big advantage of being designed specifically for use in Qt based projects. Since the handling of the Qt preprocessors was one of the build problems we had to spend the most time fixing, this is a big advantage.

Another advantage achieved by using qmake is platform independence; its input files, like the program code it builds, is supposed to work on several platforms without change. Although autotools is not platform-specific, the user has to check for platform-specific quirks and options himself.

A drawback of this approach is that it is a very Qt-specific solution. On the other hand, it can be argued that this is a non-argument since an application written with Qt must already be considered quite Qt-specific in itself.

Another, perhaps more serious, drawback to the qmake approach is

that this way of handling the build process does not scale to more than one library doing it. If every major library/framework implemented their own build system, combining two such libraries would become a lot harder than it has to be.

We chose Autotools. This choice was made early in the project, and we wanted the build system to be independent of the other choices we were still in the process of making.

3.1.7 Communication mechanism

When implementing GUI applications, there is no conventional event loop; most of the things that happen in the applications are triggered by events from the outside, i. e. from the user. That means the application needs a way of letting events propagate to all interested components as they arrive. Since we have already made the decision to base Dr. Jekyll on Qt, it is natural to use one of Qt's features: *Signals and slots*. This section introduces this mechanism.

The conventional way of doing this is using callback functions: The component receiving the input event from the user knows about the other components that want to be informed when this event occurs, and is responsible for calling them when it arrives. Although computationally cheap (the callbacks are normal procedure calls), the traditional callback mechanism is not very convenient. The class calling the callback function has to know about all the classes receiving it, making for an inflexible design and breaking the ideal of data encapsulation.

Qt tries to solve this problem by introducing the concept of signals and slots [33]. The class sending what normally would be a callback now declares a signal as a part of its interface (see figure 3.1). The receiver declares one of its functions as a slot, and a third class can connect the signal to the slot without the two classes knowing about each other at all.

All classes participating in signal/slot communication must inherit the `QObject` superclass and declare the macro `Q_OBJECT` in the beginning of the class.

When the two preceding classes have been declared, a third class can connect the signal to the slot. This third class needs a pointer to each of the participating classes. In this example it instantiates one of each, with the sender called `a` and the receiver called `b`.

When this is done, a call to `a->receive_input(int)` will cause the signal `send_trigger(int)` to be emitted, which again causes the procedure `b->receive_trigger(int)` to be called. The sending and the receiving class does not need to know about each other at all, which makes

```
// The class sending the signal:
class Sender : public QObject
{
    Q_OBJECT
public:
    void receive_input( int input_arg ) {
        std::cout << "Input: " << input_arg << std::endl;
        emit send_trigger( input_arg );
    }
signals:
    void send_trigger( int sig_arg );
};

// The class receiving the signal:
class Receiver : public QObject
{
    Q_OBJECT
public slots:
    void receive_trigger( int recv_arg ) {
        std::cout << "Received: " << recv_arg << std::endl;
    }
};

// The class instantiating and connecting a sender and receiver:
class Connector : public QObject
{
    Q_OBJECT
public:
    // This is a class constructor, i.e. the procedure that is called
    // when the class is instantiated.
    Connector() {
        Sender *a       = new Sender;
        Receiver *b     = new Receiver;

        connect(a, SIGNAL( send_trigger(int) ),
                b, SLOT( receive_trigger(int ) ));

        a->receive_input(3);
    }
};
```

Figure 3.1: The sending, receiving and connecting classes in a Qt signal/slot interaction. The `main()` routine instantiating the `Connector` is not shown.

it much easier to enforce data encapsulation. From a programmer point of view, the signal/slot mechanism is a huge step in the right direction compared to traditional callbacks.

As elegant as it seems, the signal/slot mechanism is not without drawbacks. It introduces several new keywords to the language (e.g. the “sig-

nals:" and "slots:" markers and the `emit` statement) which again have to be handled by a special preprocessor: `moc`. All Qt classes have to be run through this preprocessor, and the resulting moc-files need to be compiled together with the original source files. This quickly becomes a headache when using templates, as discussed in section 3.2.1.

3.2 Techniques

3.2.1 Templates

When dealing with data from modern MRI and CT machinery, datasets consisting of $512 \times 512 \times 512$ elements are not unthinkable. Some machines are even capable of producing movie output, multiplying the amount of data we already have per image with the number of frames in the movie. If 16 bits (2 bytes) are used to represent each data element, one such cube takes 256 megabytes. Fitting a few such datasets in the RAM of a modern workstation is perfectly possible, but the number of datasets does not have to increase much before it becomes problematic.

In order to keep the application performance good enough to permit truly interactive use, it is vital that the datasets we work on are kept in RAM. When active data is moved to hard disk (swapped out), performance drops by several orders of magnitude, making interactivity much harder to achieve. In order to have room for many simultaneous datasets, we want each dataset to take up as little space as possible.

Not using more space than strictly necessary to store each dataset is difficult, because datasets come in different bitdepths. Most of the segmented images we have worked with only use 8 bits per pixel, since it is not usual with more than 256 different labels in an image. The raw images, on the other hand, usually uses 16 bits, and we don't want to remove information from the images we read in. There are two ways of handling this: If we use the same datatype for all images, this must be big enough to store all possible bitdepths. This is simple, but costs a lot of RAM. The alternative is to find a way to vary the datatype depending on the image.

To solve this problem, we used *templates* for our data container classes. Templates is a mechanism enabling the programmer to write classes containing elements whose types are compile-time parameters. A very simple template class is shown in figure 3.2. It contains a single value of the parameterized type **T**, and offers primitive methods for getting and setting it.

Templates make it possible to write "generic" container classes – containers that are not limited to containing one specific type of objects. The C++ Standard Template Library [36, 62] is heavily based on this (hence the name).

```
#include <string>

// The declaration and definition of the template class foo:
template< typename T >
class foo
{
public:
    void set_t( T new_val ) { current_val = new_val; };
    T get_t () {  return current_val ; };
private:
    T current_val;
};

// A trivial main function:
int main ()
{
    foo< int >              afoo;
    foo< std :: string >    anotherfoo;

    afoo.set_t ( 4 );
    anotherfoo.set_t( "Gazonk!" );
};
```

Figure 3.2: A trivial template class and an equally trivial `main` function instantiating two variants of it.

Using templates, our data storage classes can store each voxel using either 8, 16 or 32 bits. This is done using the data types `uint8_t`, `uint16_t` and `uint32_t` from the `<inttypes.h>` header file. This header file is a part of C99, the 1999 revision of the C standard [61]. As the names indicate, these data types are defined to contain 8, 16 and 32 unsigned bits, respectively. This means we do not have to worry about different word lengths on different architectures. The choice of what datatype to use is done runtime, when reading the images from file.

Throughout the implementation of Dr. Jekyll, this ability to handle several data types was both a blessing and a curse. The C++ template mechanism softens the rigid typing system quite a bit, but at a price: Template classes can't take part in Qt's signal-passing mechanism, since the `moc` preprocessor (introduced in section 3.1.7) is incapable of processing template classes. In other words, classes taking part in the Qt signal/slot passing system cannot also be template classes. Fortunately, it is possible to get around this by having a template subclass that inherits signals and slots from a non-template superclass. This way, only the superclass needs to be run through `moc`.

We have used this inheritance trick several places. Although this has been relatively easy to fit into the design of the application, it an obstacle

that we would have been glad to be without.

3.2.2 Patterns

Originally influenced by the field of architecture [49], the object oriented development community has assimilated and embraced the concept of *design patterns* over the last decade. The first book on the subject is still regarded as one of the references in its field. It was published in 1994, and was written by Gamma, Helm, Johnson and Vlissides [56].

In short, a pattern is a description of a good solution to a common problem. In their book, Gamma, Helm, Johnson and Vlissides describe four essential elements of a pattern:

1. A name
2. A description of the problem it solves
3. A (more or less) abstract description of the solution
4. A list of consequences arising from the use of this pattern

Aside from being a catalog of tested good solutions to problems one may encounter, a very important feature of patterns is their function as a common vocabulary. They provide a lot of the abstractions programmers use when designing their applications with a name. Suddenly it is no longer necessary to explain in detail how component A propagates its data to components B, C and D; it is enough to simply state "I used an observer pattern for that".

The jury seems to be still out on the question of whether using design patterns actually cause programmers to write better and more maintainable programs. The empirical study done in [69] concludes that it is usually, but not always, useful to use an applicable pattern even if the problem at hand is simpler than the problem in the original pattern description.

We used several patterns occurring in the catalog part of [56]. The following is a list explaining which ones. It also explains where they are used, referring to specific parts of Dr. Jekyll. We have chosen to present the patterns and their application here, although these components are not properly introduced until chapter 4.

Observer: The observer pattern describes how an object of interest (the subject) notifies its observers when it changes. This pattern is applied several times in Dr. Jekyll:

- The `Cube` acts as a subject, and the visualization classes and plugins as observers (see figure 4.1, page 40). Upon a change in its data, the `Cube` emits a signal, triggering an update/refresh operation in the observers.
- Inside `Cubeview` (figure 4.3), `Attention_point` acts as a subject, emitting a signal whenever the user moves the focus point. The `Imagewindow` subclasses will update their contents when they receive this signal.

Singleton: The singleton pattern describes how to make sure a class only has one instance, and how to make sure that all the users of that class do not use or (accidentally) create any other instances than this one.

Dr. Jekyll is using this pattern, although with a twist. Dr. Jekyll has only one `Pluginfactory` instance, but the static accessor method isn't part of the `Pluginfactory` class itself. It is a separate method in a global namespace, because of initialization race conditions concerning global static objects in C++. Item 47 of [65] explains this issue to its full depth.

Mediator: The mediator pattern is about keeping two separate objects separate by having a third object controlling how they interact. This is very similar to how the entire Qt signal/slot mechanism works, with a sender and receiver not knowing about each other, and a third class connecting them. These similarities are further explained in [53].

The mediator pattern also applies on a higher level. The `Controller` can be thought of as a mediator between most of the other components; they do not know anything about each other, and `Controller` manages the connections between them. For instance, the plugins and `Cubeview` know nothing about each other, but `Controller` makes sure that the signals emitted by `Cubeview` every time the user clicks a mouse button is relayed correctly to the currently active plugin.

Chain of responsibility: The idea behind the chain of responsibility is that a widget can receive and pass along input without knowing who – if anyone at all – is going to handle it. The input event is sent along the chain until one of the objects picks it up, or it falls off the end and is ignored.

This pattern is a good description of how mouse clicks are handled in Dr. Jekyll; the visualization windows do not know who is going to receive it, they just send a signal with the necessary information up the chain. Normally, this signal will end up in a plugin or worker.

Iterator: The principle behind the iterator is to provide a layer of indirection between the sequential access of a container and the its underly-

ing implementation. The C++ Standard Template Library, STL [36], relies very heavily on this mechanism.

In addition to these patterns, the principle behind the `Pluginfactory` class comes directly from the "Object factory" described by Andrei Alexandrescu in chapter 8 of [50]. It is mentioned here because it is a pattern-like abstraction we have found very useful, but the author does not speak of it as a pattern.

3.2.3 Structuring components

A challenge in all systems of some complexity is how to pass information back and forth between components without losing control over who talks to who, when and how. We used Qt's signal/slot mechanism as the fundamental communication mechanism. This makes setting up and taking down connections between components fairly straight forward, but it only solves half the "cleanliness issue". As the system grows, it still becomes very hard for the developers to keep track of who gets what information from who. It is desirable to organize the lines of information in a way that makes it easy to figure out who should get what information from who and how.

In this section we use the word *aggregation*. This is used in the same way as in Gamma et al. [56], to imply that one object owns or is responsible for another.

Layers

There are several ways of thinking about the components in a system like this. All the classes are in some way aggregated by another, with the `Controller` at the top of this aggregation tree. In the beginning, we thought of this as a tree, with branches and layers as illustrated in figure 3.3. In this figure, the objects aggregate the object(s) below them.

The arrows in figures 3.3 and 3.4 symbolize information going from one object to another, mostly in the form of Qt signals.

This way of thinking comes very easily to most computer scientists, as they are used to trees and hierarchies. However, when the number of components grows, the tree quickly becomes cluttered by arrows going up, down and sideways.

Nested boxes

An alternative way of thinking is in the term of boxes within boxes, as illustrated in figure 3.4. In this figure, an object aggregates the one(s) inside

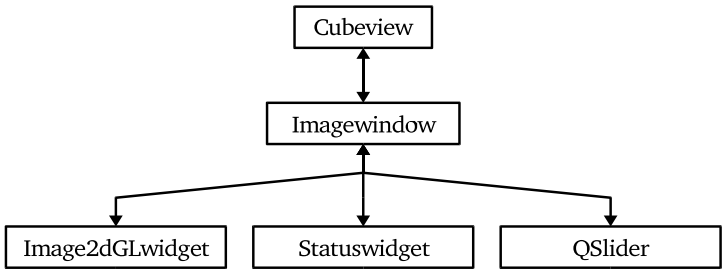

Figure 3.3: Thinking of signals and slots as going up and down through layers in a component hierarchy.

it. The arrows, representing signals going back and forth, may go between outer and inner box, or between two boxes in the same parent box, but not between two children in different parent boxes. This makes for a cleaner and more consistent interface, and we find it easier to deduce from the drawing who should be responsible for setting up connections: Both from a parent to a child and between two children, the parent is responsible for setting up and taking down the connection.

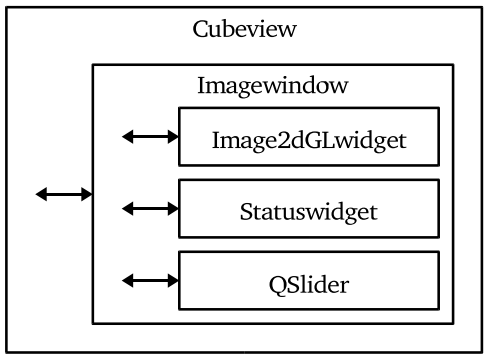

Figure 3.4: Thinking of components as boxes within each other, signals going inward and outward rather than up and down.

This explanation of how class charts should be drawn may seem like a trivial detail. Nonetheless, our experience after numerous whiteboard discussions during the development of Dr. Jekyll is that it can make a very real difference. The boxes-within-boxes approach seems to be a more convenient and scalable way of thinking about the components in a complex GUI application. Most of the figures we use in this thesis to illustrate the components of Dr. Jekyll uses this approach.

3.3 Handling data

In this section we will take a look at different ways to store and access the voxels themselves. This is a very central question for an application as data-intensive as ours. The first part of this chapter discusses different ways to lay out the voxel data in memory, while the second part is about the different tools and libraries we had available for the job.

3.3.1 Data layout

To a computer programmer, the obvious way to store a three-dimensional dataset is by using a three-dimensional array, where each voxel can be reached by using the appropriate X, Y and Z-coordinates as indexes. This is a conceptually convenient data structure, because it maps directly into the coordinate systems we are used to thinking in. It is also possible to look up the value of a single voxel at a known position with very little computational overhead.

Another way of storing cube data is by using an *octree*. In an octree the cube is divided into octets, and these octets further divided into sub-octets until we have a tree of internally homogeneous octets. A proper explanation of octrees is given in [55]. On a segmented volume, which (presumably) consists of relatively large clumps of voxels with the same label, octrees have the potential of being a lot more space-efficient than the raw array approach. However, fetching the value of a single voxel is more work, and setting the value of a single voxel may cause large parts of the tree to be recalculated.

We chose to use three-dimensional arrays for storing our cube data, because this is a very general and conceptually simple data structure.

3.3.2 Tools for handling three-dimensional arrays

Since the decision to store the data as three-dimensional arrays was made before we started searching for libraries to use, we do not discuss other kinds of data handling libraries here.

Nested STL vectors

The first data structure we tried for the image cubes was a thrice-nested STL vector, as demonstrated in figure 3.5.

Although relatively easy to use, the code becomes rather verbose because we have to do an instantiation inside each iteration. Also, for data-intensive

```
#include <iostream>
#include <vector>

// Since this is an example, we'll just assume that DT represents the
// data type we are supposed to store.

using std::vector;
vector< vector< vector< DT > > > cube;

int xmin = 0, ymin = 0, zmin = 0;
int xmax = 8, ymax = 8, zmax = 8;

for ( int x = xmin; x < xmax; ++x) {
    cube.push_back( vector< vector< DT > >() );
    for ( int y = ymin; y < ymax; ++y) {
        cube[x].push_back( vector< DT >() );
        for ( int z = zmin; z < zmax; ++z)
            cube[x][y].push_back( 0 );
    }
}

std::cout << "The value at position (1,4,3) is: "
          << cube[1][4][3]
          << std::endl;
```

Figure 3.5: STL vector initialization and access. Note that this code does not compile, since the `main()` function is removed in order to make the example more readable, and the DT datatype is not defined.

applications like ours, it can be a problem that this data structure does not store our raw data contiguously in memory,

C arrays with helper pointers

An approach other projects have used successfully is storing the data in one big dynamically allocated C array, using separate arrays of pointers to address the start of each slice and row. The idea is illustrated in figure 3.6.

In principle, this approach is computationally fast, perhaps even the fastest of the alternatives we considered. This is because an access to a voxel in the volume can use **a[x][y][z]**, thereby dereferencing from **a** to the Xth entry in the images array, then to the Yth entry in the row array, and last to the Zth entry in the column itself. Behind the scenes, the pointer arithmetics involved will only consist of additions; all three array references will only involve adding the offset to the start address of the array. This can be thought of as a convenient way to pre-cache the result of the pointer arithmetic $d + \text{length}_x \times x + \text{length}_y \times y + \text{length}_z \times z$, which has the com-

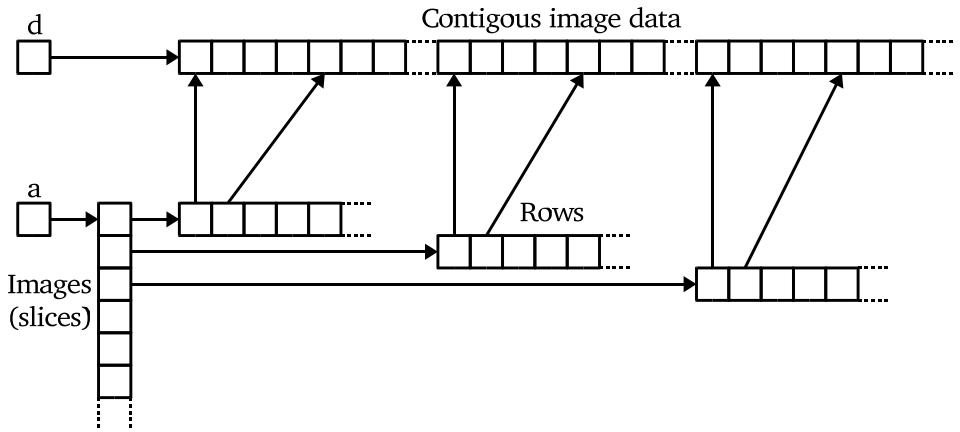

Figure 3.6: Storing the data in one contiguous C array, using helper pointers into it for slices and rows. a[x][y][z] will go to the Xth entry in the images-array, then to the Yth entry in the corresponding rows-array, and then to the Zth entry in the image data block pointed to from there.

putational drawback of involving multiplications.

The main drawback of this method is probably that a correct implementation would take a lot of work and testing in order to be usable, and we don't want to implement it ourselves if it can be avoided.

Blitz++

There are several array libraries available for C++. One such library, which was already quite mature by the time we started Dr. Jekyll, is Blitz++ [3, 78]. A code example using Blitz++ is shown in figure 3.7.

As this code demonstrates, it is very easy to instantiate and initialize multidimensional Blitz++ arrays. They are stored contiguously in memory, and even provide us with a **cube.dataFirst()** function that gives us a pointer to the start of the memory area occupied by the array. This data pointer is very handy; for example, it can be passed directly to the OpenGL **glDrawPixels()** routine, which will draw the block of data starting at this address on the screen. The functions provided for manipulating these arrays have proven reasonably fast as well, which means we get a lot of the flexibility and ease-of-use of STL vectors combined with a lot of the speed from a custom-made data layout.

The `cube = 0;` syntax used in the example sets the content of the entire cube to zero. This is not usual C++ syntax, and is made possible by the use of *expression templates*, a technique developed in part by the creator of Blitz++ [77].

```
#include <iostream>
#include <blitz/array.h>

// DT is a placeholder for our real type.

int size = 8;
blitz::Array< DT, 3 > cube(size, size, size);

cube = 0;

std::cout << "The value at position (1,4,3) is: "
          << cube(1,4,3)
          << std::endl;
```

Figure 3.7: Blitz++ vector initialization and access. This code does not compile, for the same reason as in figure 3.5.

Boost

The Boost project [4] aims to provide free, peer-reviewed C++ libraries, partly with an ambition of making them part of standard C++ over time. In October 2002, it incorporated the **boost::multi_array** library [57], which was inspired by Blitz++ (section 3.3.2). A code example of instantiation, initialization and access of a boost::multi_array is shown in figure 3.8.

```
#include <iostream>
#include <boost/multi_array.hpp>

// DT is our datatype, the variable size says how large the cube
// should be.

int size = 8;
boost::multi_array<DT, 3> cube(boost::extents[size][size][size]);

for( int x = 0; x < size; ++x )
    for( int y = 0; y < size; ++y )
        for( int z = 0; z < size; ++z )
            cube[x][y][z] = 0;

std::cout << "The value at position (1,4,3) is: "
          << cube[1][4][3]
          << std::endl;
```

Figure 3.8: Boost vector initialization and access. This code does not compile, for the same reasons as in figure 3.5.

If this had been available when we started the Dr. Jekyll project, it is

hard to tell which one we would have chosen of this and Blitz++. Both seem to offer a lot of the same functionality and quality.

Other external libraries

Other numerical C++ libraries we considered were Pooma [28], The Matrix Template Library [19] and the Template Numerical Toolkit [39].

Blitz++ was chosen after a bit of "window shopping" because it seemed to concentrate on matrices of any number of dimensions, and did this rather well.

Chapter 4

The program

This chapter discusses the structure of Dr. Jekyll. We discuss the design of the core application and the plugin interface, but not the plugins. The design and implementation of the plugins and the algorithms they contain are discussed in depth in [72].

Section 4.1 explains the architecture of Dr. Jekyll, outlining how the different components are connected.

Section 4.2 discusses some issues connected to the propagation of information between the components.

Section 4.3 concerns itself with the design of the plugin interface, explaining the thoughts behind it.

4.1 Design overview: The big picture

Most of the components of the application fall within one of five categories:

1. The data storage classes
2. The plugins containing the algorithms
3. The classes implementing the user interface
4. The worker classes containing the logic behind the user interface.
5. The glue

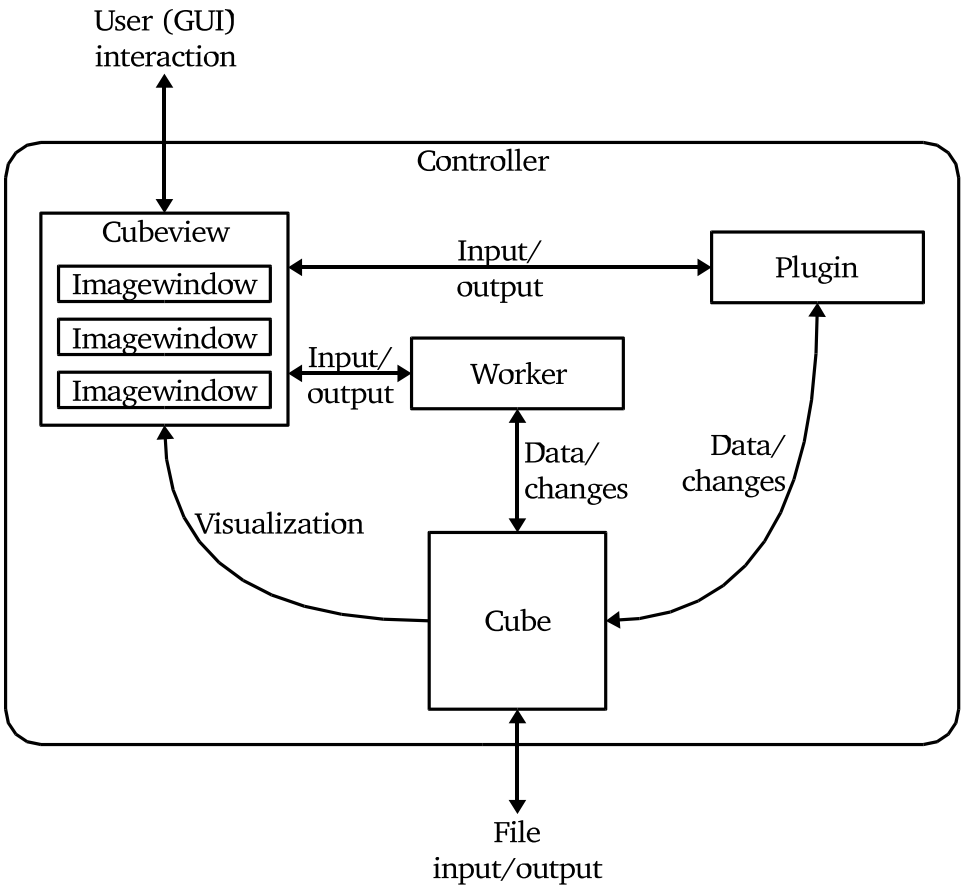

Figure 4.1: A rough overview of Dr. Jekyll. An arrow symbolizes information of some kind being sent from one component to another. The glue class `Controller` keeps track of the other classes; `Cubeview` is the top-level GUI component, `Cube` is the top of the data container hierarchy. `Plugin` and `Worker` both get user input from `Cubeview`, and they retrieve and change the data in `Cube`.

Figure 4.1 shows a simplified sketch of how the main classes in the application interact with each other.

All C++ programs must have a function called `main()`, which is automatically executed when the program is run. In Dr. Jekyll, this function is relatively small: It parses the different command line parameters (printing a help text and exiting if the parameters are not correct), initializes the various libraries needing it, and then instantiates and leaves control to the most important glue class: `Controller`.

`Controller` is both a GUI element and an important part of the internal logistics of the program. In the GUI, it is the top level menu that appears when the application is started. From its menu bar the user has the normal "quit" and "help" choices, as well as the possibility to open new cubes and

to save or close old ones. Behind the scenes, `Controller` keeps track of all the `Cube` instances, one for every dataset that is opened. Attached to every `Cube` is a `Cubeview`, which is responsible for all the visualizations of that particular dataset. There is also a number of `Worker` objects operating on each volume, and `Controller` is responsible for instantiating and connecting those. The plugins have their own instantiation mechanism, which will be explained in section 4.3.

4.1.1 The data container classes

The cube data itself is stored by the family of `Cube` objects. The inheritance hierarchy is shown in figure 4.2.

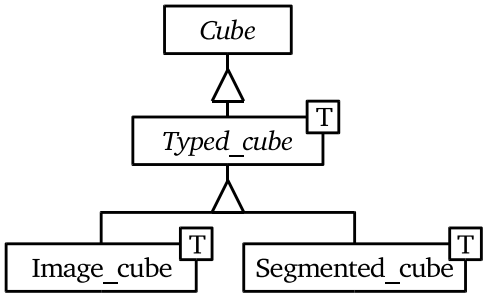

Figure 4.2: The way the `Cube` data container superclass is inherited

The reason for the two-step inheritance is that `Cube` is a Qt class, inheriting `QObject`. Since it declares its own signals and slots (a Qt mechanism explained in section 3.1.7), it can't be a template class (a limitation discussed in section 3.2.1). Hence the split-up: `Cube` contains the Qt mechanisms as well as the bitdepth/datatype independent operations and accessors, while the template class `Typed_cube` contains the parts that is bitdepth/datatype-specific but not image type specific. `Segmented_cube` is a specialization of `Typed_cube` offering functionality for handling segmented images, typically storage of label names, while `Image_cube` stores normal images (raw data).

4.1.2 The visualization classes

An overview of the object interactions inside the `Cubeview` is shown in figure 4.3.

The core component of the visualization system is the `Cubeview` class. When `Controller` instantiates a `Cube`, it also creates a `Cubeview` which is responsible for the visualization of that cube object, including the opening and closing of visualization windows when the user requests it.

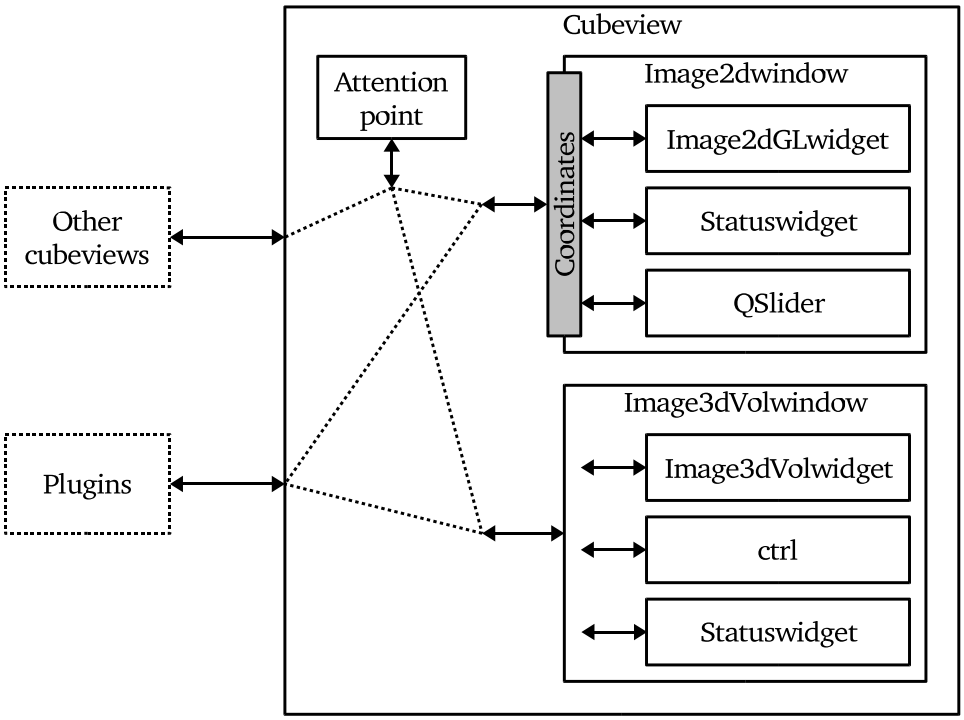

Figure 4.3: An overview of `Cubeview` and its contents. The dotted lines represent the propagation of user input. `Cubeview` contains an `Attention_point` and a number of `Imagewindow` subclasses like `Image2dwindow` and `Image3dVolwindow`, which in turn contain other components. Everyone communicates with `Attention_point` to know where the user is looking, and this information can be propagated to other `Cubeview` objects in order to look at the same coordinate in several datasets.

When a new visualization window is opened, a subclass of `Imagewindow` is used. The simplest variant is called `Image2dwindow`, and is a slice-by-slice two-dimensional projection of the volume. `Image3dwindow` is another child of `Imagewindow`. It is in turn inherited by `Image3dVolwindow`, which displays a volume rendering of the dataset.

The volume rendering implementation used in `Image3dVolwindow` is based on the SimVoleon library, which is API compatible with the VolumeViz library made by TGS [40]. We were kindly permitted to use a pre-release of this library by the company developing it, Systems In Motion (SIM) [37]. This library is not available under an open source license, so unfortunately the volume visualization will be unavailable to most users.

Inside the `Imagewindow` derivatives, there is yet another set of objects interacting with each other and the outside. The only one common to all the different visualization windows is `Statuswidget`, which shows the coordinate of the currently selected voxel, its value, the size of the currently

selected component (if any), controls for scaling (zooming) the projection, and a few other things.

A glue class worth mentioning is the `Coordinates` class. It is needed because the coordinate system used in `Image2dGLwidget` is different from the one used globally in the application (see section 4.2.1). `Image2dwindow` uses it as a converter, relaying all signals through it in order to send and receive screen-relative two-dimensional coordinates to and from the inside, and cube-relative three-dimensional coordinates to and from the outside.

While every `Cubeview` can instantiate lots of `Imagewindow` derivatives, it can only have one `Attention_point`. When the user is viewing the data, there is always a single coordinate which is the center of attention. All the two-dimensional projections follow this point, displaying the plane it is in. In other words, the attention point is where the X, Y and Z-normal two-dimensional visualization windows meet. The `Attention_point` class is responsible for keeping track of this coordinate

This structure is intended to make the application flexible, in the sense that new visualizations can be added without changing the basic interaction patterns between the different components.

4.1.3 The worker classes

The workers originally started out as plugins, and they still share much of their interface to the rest of the application. There are currently only two workers: `Undoworker` and `CCWorker`. The first is responsible for the undo functionality, reverting the last change to the dataset when the user presses `Control-Z`. The second is responsible for the connected component selection, highlighting the component containing the coordinate selected by the user. For an example of this highlighting, see the segmentation error example, the lower half of figure 1.2.

These workers could have been implemented as normal plugins. There are some technical reasons for making them a separate category, but the most important reason is aesthetic: The workers are fundamental parts of the logic behind the user interface, while the plugins are data-oriented algorithms. For a discussion of the plugin hierarchy and interface, we refer to sections 4.3 and 5.2.

4.2 Information flow issues

There is a lot of data being sent back and forth between the different components of Dr. Jekyll. This section discusses some of the issues arising from this. Section 4.2.1 talks about the different coordinate systems being used in

different components, and section 4.2.2 explains some of the difficulties that arise when some components don't understand all the possible datatypes.

4.2.1 Coordinate systems

A problem we encountered when connecting the different components was the different coordinate systems used by the different components. In the X window system, every window has a two-dimensional coordinate system with (0,0) in the upper left corner. The OpenGL rendering windows, on the other hand, places (0,0) in the lower left corner. Examples of both are shown in figure 4.4.

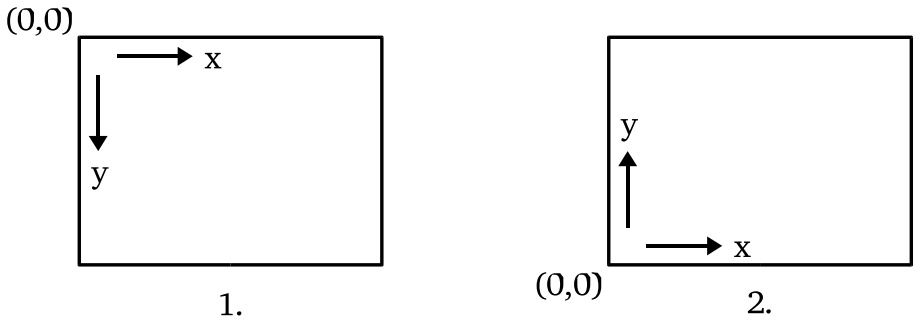

Figure 4.4: Example 1 shows the coordinate system in the X window system; example 2 shows the coordinate system in OpenGL.

Somewhere in the component chain these coordinates have to be transformed between image/frame-relative (X,Y) and cube-relative (X,Y,Z). We chose to implement a separate class for doing this: `Coordinates`. As shown in the Cubeview drawing in figure 4.3, This class is used by `Image2dwindow` as a translator module that all coordinate information is passed through on the way in and out. An implication of this is that the information following the dotted lines in figure 4.3 is cube-relative only; no image/frame relative coordinates are ever sent outside `Imagewindow`.

When using an XYZ-indexed storage method for the data, there are several different coordinate systems to keep track of. One way of looking at the cube is with (0,0,0) in the lower left corner and the Z axis pointing into the screen; this is convenient because it is possible to let the X and Y axes match the X and Y of the single images making up the volume (assuming the volume is stored as separate slices), and moving along the Z axis can be thought of as leafing through the stack of images.

4.2.2 Data Types

Unfortunately, not all the different components can handle all our different data types. There are places where we need to convert cube data from one data type to another before the receiving object can use it.

Bitdepths larger than 8 bits are primarily interesting for computational (algorithm-related) purposes; even on modern display hardware, there is little or nothing to gain by using more than 8 bits grayscale colordepth. Seen together with the fact that GUI classes can't easily be templates (see section 3.2.1), this makes it very tempting to introduce a data transformation to make sure the data sent to the visualization layer is always of the same type, regardless of the actual template type used in the container. By doing this, the visualization classes would not have to handle more than one data type, saving a lot of work. Version 1.0 of Dr. Jekyll uses this shortcut in the visualization classes: All the `Imagewindow` subclasses operate only on `uint8_t` data.

This approach becomes a problem when working with segmented volumes. In these volumes, the value as such of each voxel is meaningless: It does not carry any meaning except the fact that all the voxels carrying the same value happen to belong to the same label. This means that even though the three label-values 5, 6 and 7 are very close numerically, there is no similarity between them. In order to see the borders between components more clearly, the user may want to use a *random colormap* on the visualization. This operation would map each value to a different color in the display, for example so that these three labels are shown as red, green and blue respectively. This is where the color down-scaling becomes a problem: If there are more than 256 segments, different values will be collapsed into the same, and we will lose information in the visualization layer. This is not acceptable, so Dr. Jekyll needs some rethinking, and probably also reprogramming, in this area.

4.3 Plugin interface

One of the three main goals of Dr. Jekyll was to offer a documented plugin interface for new algorithms. We want our interface to satisfy the following requirements:

Genericity: We want it to be generic, in the sense that it is not specific to a certain type or category of plugin.

Completeness: We'd like it to be complete, so it does not put unreasonable constraints on what new plugins can do.

Ease of use: The interface should be as simple and powerful as possible, making it easy to use.

Extensibility: Even though we try to design a complete and generic interface, it is not very probable that we will be able to predict the needs of all the different plugins people may want to write. Because of this, it should be possible to extend the interface as easily as possible.

A very convenient way of specifying such an interface is to create a class containing the necessary declaration and definitions, and then letting all the plugin classes inherit it. This is what we have done with the `Plugin` class:

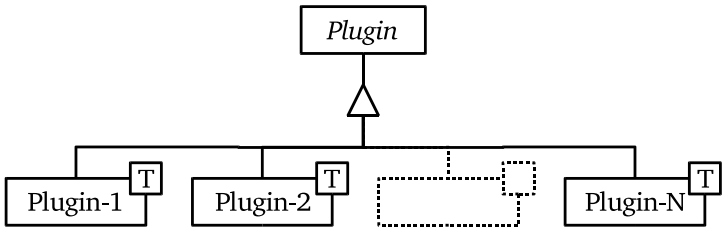

Figure 4.5: The `Plugin` superclass, inherited by all plugins.

The `Plugin` class is an *abstract class*. This means it is not possible to instantiate it directly – code trying to do so will not compile. In order to use such a class, one must instantiate one of its subclasses.

Abstract classes in C++ are tightly coupled to the concept of *pure virtual* functions. A pure virtual function is a class function that is declared, but not defined, i. e. no function body is provided. Since a class cannot be instantiated unless all its members are defined, creating a pure virtual function is one way of stating that a class is supposed to be abstract. Another way is to declare the class constructors as `protected`, since a protected class member is only available to subclasses of the class where they are declared.

Since not all the plugins need all of the interface, the superclass defines "no-op" (empty) functions for the parts that are not compulsory. If, for example, a plugin is not interested in keypresses from the user, it will simply not reimplement the `recv_keypress` function. This will cause the definition in the superclass, ignoring the keypress and doing nothing, to be used.

Since `Plugin` is abstract, and therefore allowed to declare functions without defining them, it is also possible to force subclasses to define certain functions: If there is a function that all the plugins must reimplement, the superclass can declare it as pure virtual.

Chapter 5

Results

The main result of the Dr. Jekyll project is, of course, the Dr. Jekyll application itself. This chapter gives an overview of Dr. Jekyll version 1.0, released in April/May 2003, when this thesis was finalized. The chapter is split in two parts: The first part gives an overview of the functionality offered by the program in its current state, i. e. viewed as a standalone viewing and editing application. The second part focuses on Dr. Jekyll as a platform for further development and prototyping, presenting its capabilities.

5.1 Application functionality

Through the top-level menu widget, a screenshot of which is shown in figure 5.1, the user can use the menubar to open, close and save cubes, as well as exiting the application. All the open cubes show up in the list view, displaying the name (if given one by the user) and information about the size, bitdepth and whether the cube is segmented or not.

In order to perform operations on a cube, the user must first select it in the cube list. (In the figure, the lower of the two cubes is selected.) All buttons being pressed will then apply to this cube. The buttons are divided into three categories:

Visualization: There are four buttons in this box, each of them opening a different view of the cube when pressed. As the button titles suggest, three of them open an X, Y or Z-normal two-dimensional visualization. The last one opens a volume visualization, but as explained in section 4.1.2 this is, regrettably, not normally available.

Tools: This box contains buttons for connecting and disconnecting cubes and for resetting the current region of interest. Two cubes that are connected will share their attention point, meaning that they are al-

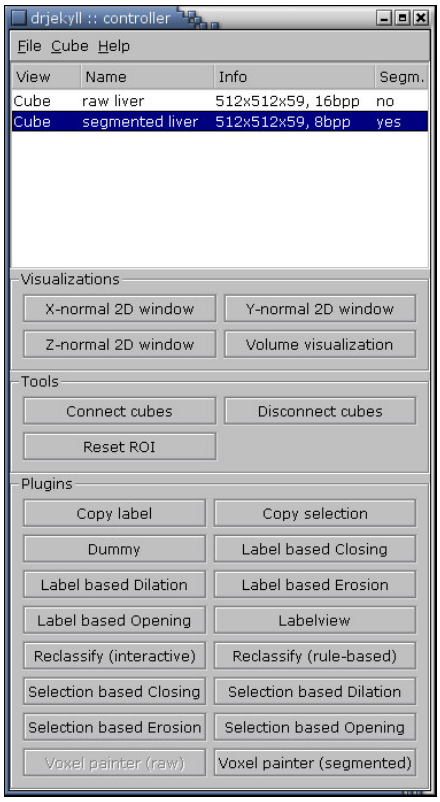

Figure 5.1: A screenshot of the `Controller` widget with two cubes opened. The segmented cube is selected, so the button with the raw cube voxel painting in the lower left is disabled.

ways focused on the same coordinate. This can be very useful when navigating one cube containing raw data and another cube containing a segmented (or otherwise post processed) cube derived from this raw cube.

The region of interest, often called ROI, is a subset of the data in the given cube. All plugin operations applies only to this subset. In other words, plugins changing the data they work on do not necessarily apply that change to the entire cube; by defining a ROI, the user can limit the changes to a given region. Although the different visualization windows offer functionality for setting and clearing this ROI, the `Controller` also offers a button for resetting it.

Plugins: This is the biggest box of buttons, and contains a button for each available plugin. When a button is pressed, the relevant plugin is activated on the cube object that is currently selected in the list view. The button box is updated every time the cube selection changes, so but-

tons not making sense in the context of the current cube are disabled.

The first thing the user will usually do is to open a dataset (cube), and then some visualization windows of this cube. A single, two-dimensional visualization window (the `Image2dwindow` class) is shown in figure 5.2.

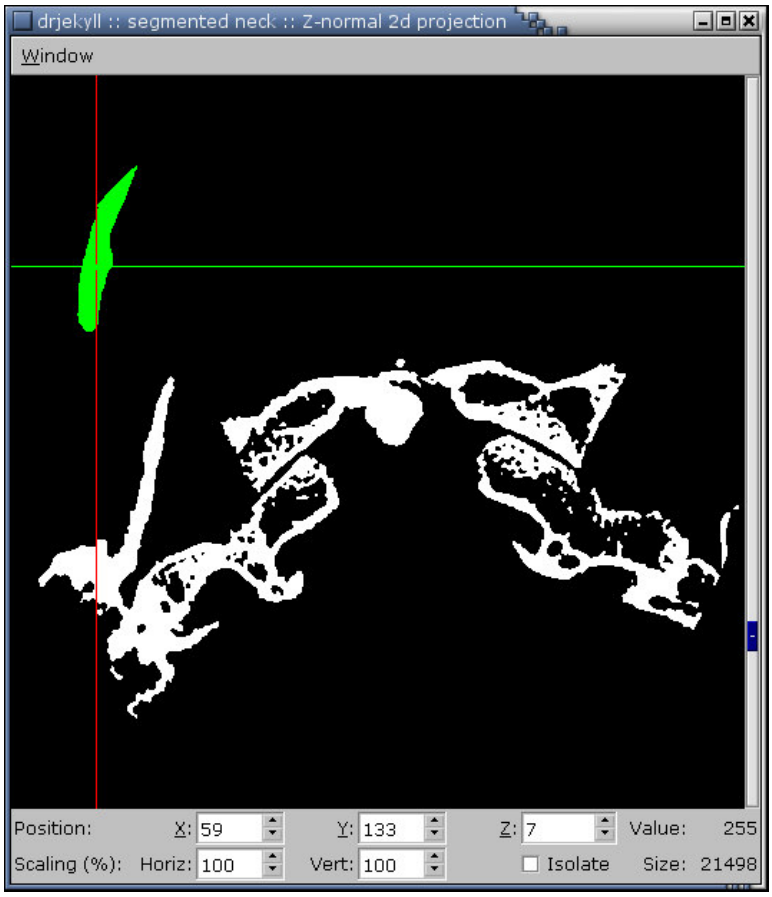

Figure 5.2: A screenshot of a single `Image2dwindow` widget.

In the bottom of the visualization window, we see a box displaying some information, most of which the user can change. This box is the `Statuswidget`. The **Position X:**, **Y:** and **Z:** fields show the coordinates of the current attention point. On the right edge, the **Value** field shows the value of the voxel in the attention point. On the second line, the **Scaling Horiz:** and **Vert:** fields let the user zoom the visualization horizontally and vertically. The **Isolate** button allows the user to limit all operations to the displayed slice (by setting the region of interest), and the **Size** field shows the number of voxels in the currently selected connected component (highlighted in green). The **Position** and **Scaling** fields can be changed by the

user, directly in the widget.

Figure 5.3 shows four two-dimensional visualizations. The cubes shown here are the ones that are shown in the cube list in the screenshot of the `Controller` widget in figure 5.1. The two on the left side show the "segmented liver" dataset, the two on the right show "raw liver". The user has used the "connect" function on these cubes, so they are centered on the same cube coordinate. In the view of the segmented dataset, the user has selected the component under the attention point, and all the voxels belonging to this component are highlighted in green.

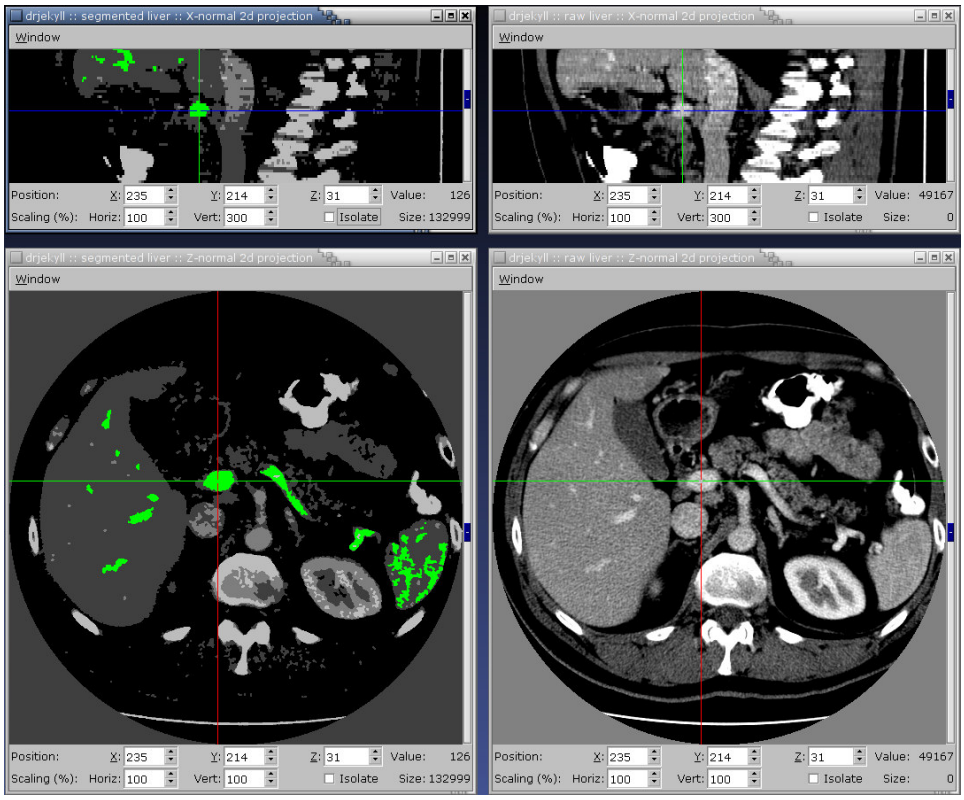

Figure 5.3: A screenshot of four `Imagewindow` subclass widgets. The two on the right display a raw dataset, and the two on the left show the segmented version of the same dataset. The two X-normal views on the top are zoomed to a factor of 300% in the vertical direction. The green color marks the selected connected component.

As the user moves between the different visualization windows, the selection in the cube list updates itself to reflect what cube the user is looking at. This way, when the user moves the mouse cursor from a visualization window to the `Controller` widget and clicks a plugin button, the plugin will operate on the cube that the user expects it to.

When the user wants to perform a change on the dataset he is looking

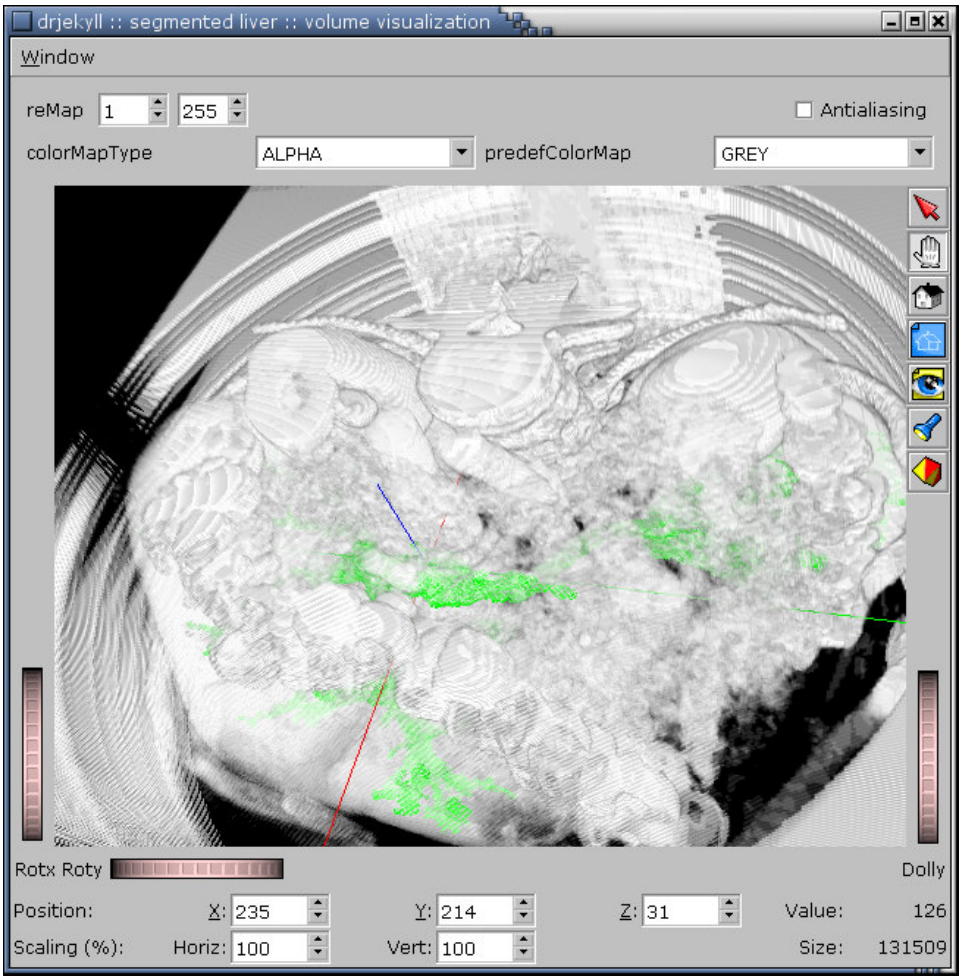

Figure 5.4: A screenshot of the volume visualization. The highlighted connected component is shown in green, and the current attention point is marked by the three colored lines crossing. The attention point and the highlighted component are the same as in figure 5.3.

at, he will first choose which dataset (presuming he has multiple cubes open at the same time). This can be done explicitly, by choosing it from the list, or implicitly by focusing on a visualization window of that cube. Then he will push one of the plugin buttons. A very simple example of a plugin is the voxel painter plugin; this comes in two variants, one for raw and one for segmented datasets. The segmented variant will open the following dialog box:

When the user has chosen which label and which structure element to paint with, he can move the mouse in the two-dimensional visualization windows to select which coordinates to paint.

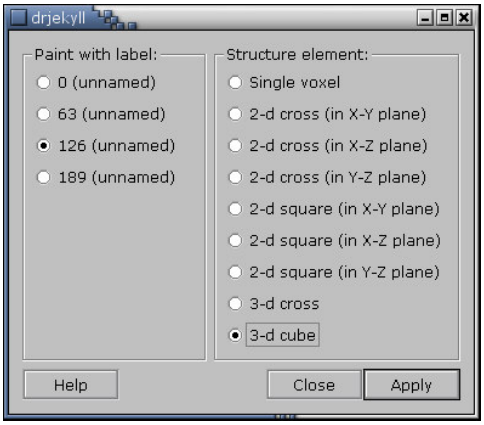

Figure 5.5: The voxel painting dialog. In the left column, the user chooses which label to paint with, in the right column he chooses which structure element ("brush") to use.

Although the other plugins will perform different functions, most of them work in roughly the same way: The user pushes the plugin button, which opens the plugin dialog. Then the user will give the plugin input, both through the dialog and by selecting voxels in the visualization windows. The plugin will respond by highlighting areas in the visualizations and/or by changing data.

We have not had the possibility to have a large user audience test our program. However, we have had some user testing and feedback. The opinion of the user in question was that Dr. Jekyll has reached a point where it can be useful in real-life post-processing work.

5.2 Development framework functionality

Besides being a volume viewer application, Dr. Jekyll tries to offer an extensible and flexible framework for prototyping of new image treatment algorithms and methods. Most of this extensibility is intended to be realized through the plugin interface.

5.2.1 Plugin interface

The plugin interface is specified and documented in the declaration of the `Plugin` class. The thoughts and intentions behind this interface are discussed in section 4.3, while this section gives an overview of the functionality available to someone developing a new plugin. For more specific documentation, we refer to the documentation of the `Plugin` class.

Meta information: The plugin is required to reimplement some trivial functions that serve to describe the plugin. This includes the name of the plugin, a sentence describing what it does, and what kind of data it applies to (raw data, segmented data or all kinds of data).

Input: There are eight different input functions (slots) declared in the `Plugin` superclass, but the new plugin is only required to reimplement one of them. Default no-op implementations are provided for the other seven, which the plugin may reimplement if necessary.

- The first four are for receiving new attention points, selected points, and pressed and released keys.

  The visualization layer differentiates between the attention point, which is the coordinate which is currently "targeted", and the selection point, which is the coordinate actively selected by the user. A selected point will always be equal to the attention point at the time of selection, but the user does not have to select all attention points. Most plugins only need to handle the selected point, i. e. the coordinates that are actively marked by the user.
- Then comes the required one, `resync()`, which is a trigger telling the plugin to update its internal state. This signal comes whenever someone else may have updated the data in the cube, so any state based on the cube data should be updated.
- Most plugins have a dialog window for interacting with the user, and almost all of these dialog buttons have a row at the bottom saying "help", "apply" and "close". The `Plugin` class declares a virtual handler function for each of these buttons.

Output: The simplest and most direct way of outputting data is to alter the displayed data itself. When the `Plugin::activate()` function is called, one of the parameters sent with it is the current `Cube` object. Through this object the plugin can get to the raw cube data itself. Other output methods include the ability to highlight voxels. This highlighting mechanism is used in figures 5.2, 5.3 and 5.4, where the highlighted areas show up in green. There are also predefined signals for opening new cubes (useful if the result of some operation should be opened and shown in a new cube instead of overwriting the data in the current one) and for turning the connected component selection on and off.

Apart from the explicit mechanisms declared in the `Plugin` superclass, the most important input/output method for plugins is the interaction with the `Cube` object itself. Through this object, the plugin can access the underlying dataset and its meta information directly. The main dataset is a

Blitz++ array (see section 3.3.2). This array is declared in `Typed_cube`, so in order to access the raw data the plugin must transform the `Cube` instance it gets to a subclass instance: `Typed_cube`, `Segmented_cube` or `Image_cube`. This transformation is called *downcasting*. Dr. Jekyll provides library functions doing this downcasting, in order to make it easy for plugin developers to get to those subclasses.

There are two Blitz++ arrays in the cube object. The first, called `entire_cube_`, contains all the data in this particular cube. The second is called `interest_cube_`, and references – not copies – the data in the `entire_cube_`, but only the subset contained within the current region of interest (ROI). The array indexes are adjusted so the same coordinate refers to the same voxel in both the cubes. This subcube is automatically readjusted every time the user changes the ROI information. By using this array, the plugins don't have to know anything about the ROI mechanism; their operations are transparently limited to the data contained in the ROI. Blitz++ makes this subcube-referencing very easy, and saved us a lot of work during the implementation of ROIs.

There are other kinds of information the plugins want to get from the `Cube` family. In order to make this easy, `Cube` and `Typed_cube` are thoroughly documented using the Doxygen inline documentation system [76]. The documentation of `Plugin` also uses this documentation format.

Chapter 6

Discussion

In this chapter, we will discuss and evaluate the application we've made:

Section 6.1 is about the overall design of the application; we try to evaluate the structure by looking at component interconnections.

Section 6.2 discusses our choice of tools and libraries, explaining which ones we would have made again and which ones we wouldn't.

Section 6.3 talks about the choice of C++ as programming language.

Section 6.4 is an evaluation of the application as a whole.

6.1 High level design

In this section we will try to evaluate the overall design of Dr. Jekyll. Exactly how to assess software design quality is still a subject of discussion, we have tried to follow one of the suggested approaches.

6.1.1 Counting couplings

In [64], Lindvall, Tesoriero and Costa proposes a method for evaluating the *maintainability* of a software architecture. The basic assumption is that isolating different components from each other (a principle also going by the names "encapsulation", or "loose coupling"), indicates a good architecture. In a system where all the components know about and reference each other, one change in a component somewhere has a higher risk of requiring something to be changed (*secondary changes*) in on or more other components. In [52], Collofello and Orn calls this *ripple effects*.

The method outlined by Lindvall et al. considers each *module* as a separate unit. A module refers to an intentional clustering of classes, which

means that references between classes in the same module are not counted. They then count how many times a module references one of the other modules. It follows from the underlying assumption that a system where many modules have a low number of such couplings has a better design than an architecture where a lot of modules have a high number of couplings, i. e. where they reference many others.

However, those references are often there for a reason, and the trade-off involved in reducing the cross-dependencies in a system is that some of the modules will take on a role as hubs, passing information back and forth on behalf of the others. Lindvall et al. have done this on purpose, by adopting the mediator pattern from [56].

As we touched upon in section 3.2.2, Dr. Jekyll has done very much the same thing, even though we had no clear intention of applying the mediator pattern when we started out. If we, as Dalheimer in [53], assume that the Qt signal/slot mechanism is just a twist to the mediator pattern, we can draw the references between the components of Dr. Jekyll as we have done in figure 6.1. Note that this figure is not complete, nor is it an exact map of the logical layout of the components. As in [64], this drawing tries to show the logical modules instead of the actual classes. In particular, `Pluginfactory` and `Controller` are considered to be parts of the same module, even though they are two separate classes in the implementation. The reason why this figure bears little resemblance to figure 4.1 is that this one shows references, while the other shows information sending.

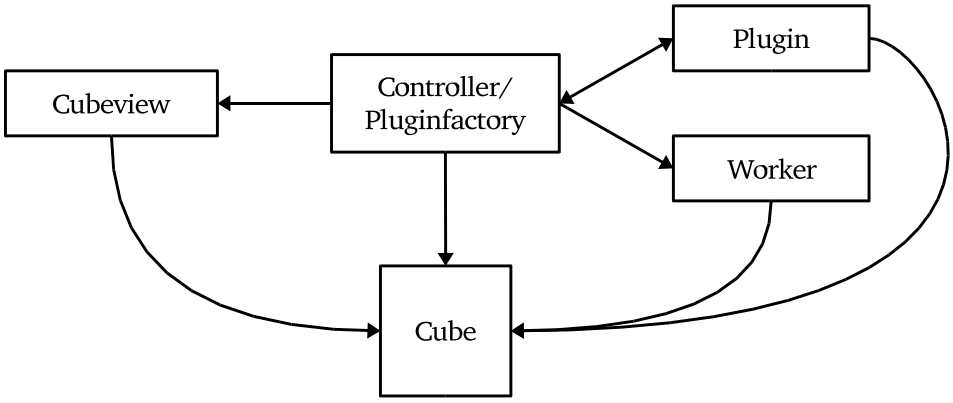

Figure 6.1: The references between the main components. An arrow from A to B indicates that A contains a reference to B.

From the figure, we can see there are no direct couplings between the user interface module, the plugin module, and the worker module. All the modules reference the cube itself, but that is not surprising, given that it serves as a data storing backend, and that all the other classes work on the

data it contains. We perceive this as an indicator that the overall design of Dr. Jekyll is sound.

There are two problems with this approach. The first is that Dr. Jekyll is, after all, not a very big application. It is therefore questionable whether this drawing (or its accompanying statistics) is a good indicator of the maintainability of the design.

The second problem is that it can be questioned whether our use of the Qt signal/slot system really is a very good example of the mediator pattern. In the original pattern, the colleagues only talk to the mediator, which serves as an information hub. When something comes in from one colleague, the mediator will update those colleagues needing it. However, the role of the parent as a hub has a tendency to disappear when programming with Qt; the parent will set up all the necessary couplings initially, by using `connect()` on the signals and slots offered by the various colleagues. Then, when one colleague has new information it needs to send to the others, it will emit a signal. This signal will then be automatically propagated to the slots of the other colleagues connected to it. As Dalheimer points out, two QObject classes being connected does not need to explicitly know about each other, nor do they need to know about the (mediator) class connecting them. Our signal/slot example, listing 3.1 on page 26, shows this. However, if a signal needs to carry any additional information, this has to be done in the form of function parameters. In the case of our example, the parameter list consists of a single integer. This argument list has to match the argument list of the slot. (There is one exception to this: A slot can choose to ignore the parameters of the signal it receives.) In other words, even though there are no explicit references between the modules, this interface compatibility requirement creates an implicit dependency. If the changes made to a component include changing the signature of a signal function, this is very likely to cause the ripple effects that we tried to avoid.

One obvious way of avoiding this problem is to adhere more closely to the original mediator pattern. This would mean not making connections directly between colleagues, but only between mediator and colleague. In hindsight, we find it likely that Dr. Jekyll would have been less vulnerable to ripple effects if we had followed this practice.

6.1.2 Examining ripple effects

In addition to the analytical approach in section 6.1.1, we will try to evaluate the design from a more empirical angle. Dr. Jekyll has been changed and extended several times since the main architecture stabilized. By examining how easy or hard some of these changes were to incorporate, we hope to reach a conclusion about how well the design of Dr. Jekyll is able to cope with changes and extensions in general.

Regions of interest

Less than a month before Dr. Jekyll was going to be released in a stable version 1.0, we had to make a significant change due to user feedback. This consisted of adding functionality for defining and handling simple *regions of interest*. A region of interest, often called a ROI, is a specific part of the image that you want to perform some specific function on. This is often implemented using a bitmask, specifying for each coordinate in the image whether it belongs to the ROI or not. We have simplified it a little bit, so that a ROI can be any regular subcube of the dataset specified by two coordinates: $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$.

As of version 1.0, the only way the user has of specifying a ROI is by using the checkbox marked "Isolate" in the two-dimensional visualization windows (see figure 5.2). This will define a ROI consisting of the slice as seen in that window.

Initially, we were afraid that this would mean rewriting huge parts of the application. This turned out not to be the case. Of course, the GUI layer had to be changed to include the button, and the `Cube` itself needed the functions necessary to get, set and keep the ROI. However, most of this programming was fairly straight forward. What caused us more of a headache was the fact that we broke one of the stated assumptions throughout the application: That all cubes begin at the coordinate (0,0,0). Apart from finding and removing this assumption, the changes we needed in the plugins were surprisingly small.

All in all we estimate that the work with implementing ROIs in Dr. Jekyll, from the first change to the last bugfix, took less than a man-week of work. It did not require any changes to the underlying architecture; the figures 4.1 and 6.1 are the same as they were before ROIs were added. We believe this to be an indication that the design of the application is sound.

A button here and a button there

Because the components treating the data and the visualization/interaction components are disconnected, it is very difficult for the members of the first category to directly change the appearance of the classes in the second category. Imagine, for instance, that someone writes a new plugin that needs some kind of button in the `Imagewindow`. If the plugin class had direct access to the `Imagewindow`, it could manipulate its internal Qt layout directly, adding the button and connecting its outgoing signal. For the plugin developer, this would be a quick and convenient way of doing it.

In Dr. Jekyll version 1.0, it is not possible for plugins to add buttons or menu entries to the windows without changing a number of other classes, including the relevant `Imagewindow` subclass, `Cubeview`, and `Pluginfactory`.

This is because the plugins do not have any kind of access to the visualization classes. All buttons must be added to the `Imagewindow` code. This is the reason why there is no menu entry for the undo plugin.

We believe the lack of dependencies between different classes to be a good thing, but in this case the encapsulation comes at a price. We would prefer a somewhat easier way for the plugins and workers to add GUI components to the user interface, and see this as a weakness in the design.

Plug it in

One of our goals was to make it easy for users to add their own algorithms, in the form of plugins. In order to add a plugin to the program, the only files (apart from the ones containing the plugin in question) that need changing is the build system. There is no central plugin list needing to be updated, since the plugins register themselves with the `Pluginfactory`.

This means a developer can write a new plugin by looking only at the source code of the `Plugin` superclass, which all plugins inherit, and the documentation (not the source code) of the `Cube` and `Typed_cube` classes. He or she does not need to know how the rest of the application works, and we see this as a strength in the application design.

6.2 Choice of tools and libraries

In this section we will discuss the choices that are outlined in section 3.1, trying to conclude whether each choice was a good one or not.

6.2.1 Blitz++

In hindsight, we are very happy with the choice of Blitz++ as a backend data container. Throughout the development of Dr. Jekyll we have never been able to trace an application crash to a bug in Blitz++. Neither has it proven to be a performance bottleneck. Although we have had performance problems at times, the solutions have been to use more Blitz++ functionality just as often as it has been to use less. Lastly, it has proven to be very powerful while still giving the programmer a very large degree of freedom in deciding how the data should be arranged in memory.

We conclude that the use of Blitz++ arrays as data storage format was a good choice.

6.2.2 ImageMagick

Unlike Blitz++, ImageMagick was a familiar name to us when we were evaluating different libraries. It is an old and rather well-known open source project, and it has not disappointed. The only thing we miss in it is a bit more speed, as the image reading functionality in the Qt library is noticeably faster. This is not a big problem when reading single images, but when reading in volumes of over 50 images from file the wait time becomes an issue.

Like Blitz++, ImageMagick has never presented a stability problem. It also deserves credit for its functionality, allowing Dr. Jekyll to handle literally dozens of file formats transparently. Apart from the slight performance hit, we have no hesitations recommending ImageMagick to people writing similar applications.

6.2.3 Autotools

After the initial configuration, autotools has been a robust and reliable build system. However, incorporating the two Qt preprocessors (**moc** and **uic**) required a considerable amount of work and documentation reading, and adding new files requiring preprocessing to the build system is not as simple as we would like it to be. It is possible that a lot of these problems could have been ameliorated with the use of pre-written autoconf macros like **autoqt** [79].

If asked for advice by someone starting a Qt project from scratch today, we would recommend them to take a very close look at qmake before choosing autotools. The fact that qmake has built-in logic to handle the preprocessing steps is an advantage that should not be underestimated.

6.2.4 Qt

Most software libraries influence only isolated parts of the application code. For example, the code in Dr. Jekyll which uses Magick++ is limited to the contents of a few functions. If we want to switch to another library for image reading and writing, it is enough to replace the contents of those functions. This is not necessarily the case with a GUI framework. Every component taking part in the user-driven event loop in a GUI application contains code specific to the framework, so switching from one library to another is no longer a straightforward task. This makes it necessary to choose GUI framework with great care, because the decision can not easily be reversed.

As a toolkit for creating graphical application, Qt is very convenient and well documented. The signal/slot mechanism works quite well, and there

are not many others who can match the "write once, compile everywhere" portability it offers.

However, the template/moc problem described in section 3.2.1 became more and more of a headache through the implementation process. For instance, the type limitation in the GUI components discussed in section 4.2.2 could have been handled a lot easier if signals and slots could have been templates.

If starting the project from scratch today, we can not completely rule out the possibility that we would have used Qt. We are certain, though, that we would have spent a lot more time evaluating the alternatives. If we did not depend as heavily on templates as we do, Qt would be a clear favorite.

6.3 Choice of programming language

Like the choice to use Qt, the choice of C++ as programming language has been the subject of much debate. C++ is powerful, but it has its share of peculiarities and syntactic quirks. On his "Frequently Asked Questions" (FAQ) page, Bjarne Stroustrup writes that one can easily spend between one and two years becoming fluent in C++ [74].

The different parts of Dr. Jekyll have very different requirements: In the high-level GUI and application logic parts, we would have been glad to give up some run-time efficiency for more flexibility and a higher abstraction level. In some of the components operating on the dataset itself, we would have done the opposite.

If starting with a clean slate today, there are two obvious approaches we would have considered:

- If the entire application had to be written in the same language, the compromise between abstraction and run-time speed offered by C++ makes it a clear favorite for this kind of program.

  However, we would have performed some experiments before going with C++. We would have checked how modern Java compilers/virtual machines compare with regard to run-time speed. We would also do some research on script language extensions like Numerical Python [23, 58].

- If writing in only one language was not required, we would have taken a very close look at the last alternative of section 3.1.1: A sandwich solution of a scripting language and a traditional binary compiled language. This method has the potential of offering the best of both worlds: High-level programming for the user interaction logic layer,

and run-time speed at the layer where the data is stored and manipulated.

Numerical Python, discussed in the previous point, can be thought of as a ready-to-use implementation of the lower of these two layers.

This approach is not as thoroughly tested as the single-language approach, and would probably require more experimenting and exploration of technically unknown territory.

6.4 Evaluating Dr. Jekyll

In this section we will evaluate the application as a whole. The three goals stated in section 1.4 are a good starting point for this discussion:

Interface: Dr. Jekyll lets the user open, view, change and save a dataset using the graphical user interface. We believe we have succeeded in giving the application a consistent and usable interface.

Visualization: We have implemented two-dimensional slice-based visualizations, which are able to update themselves with a relatively low delay. This is partly because of the ability to send data directly from Blitz++ to OpenGL.

In addition to the two-dimensional projections, we have successfully integrated a separate volume rendering library into our application. This volume visualization is currently not available to other users.

Extending Dr. Jekyll with other volume visualizations (for instance using VTK) would be very interesting, since this is an area where we have not been able to do as much work as we had hoped.

Plugin interface: The plugin interface is specified in the `Plugin` abstract superclass, and documented using an inline documentation system. We have written over a dozen plugins using this interface, and believe we have succeeded in our ambition to make a plugin interface usable by others. Naturally, the ultimate test of whether the interface is good enough is to have somebody else try to use it.

All in all, we believe we have succeeded in making a usable voxel editor. The visualization goal was only partly reached, but the other goals were fulfilled.

We are very happy with our strategy of using libraries for as many tasks as possible. Although some of the libraries have been more pleasant to work with than others, we are convinced that this strategy has enabled us to make an application far more powerful than it otherwise would have been.

It is always possible to extend a program further, and Dr. Jekyll is no exception to this; we can think of many ways to extend and refine it. Nonetheless, we feel that version 1.0 deserves to be called a finished work.

Chapter 7

Conclusion

We have, in cooperation with others, designed and implemented a voxel editor. The result is an application usable both as a post-processing application and as a development framework for other people wanting to test imaging algorithms. The application is designed with the aim of being flexible and extensible.

We have built our application using numerous external libraries. We are very satisfied with the choice of Blitz++ as the data container at the core of the program; we do not believe we would have done any better by writing our own data storage component. We are also satisfied with the choice of ImageMagick for file input/output. Although Autotools has worked fine as a build system, we might have chosen qmake if we had known how much work was needed to make Autotools handle the Qt preprocessors correctly. The Qt GUI library has worked well, with the exception of its inability to handle template classes properly; this limitation has been a problem. The choice of C++ as a programming language has been discussed, and even though we would have considered other languages if starting from scratch today, C++ would still be a very strong candidate.

In general, we believe that our general approach of using external libraries whenever possible has worked well.

The resulting application, Dr. Jekyll, is released to the public under the GNU General Public License, GPL. This license allows other people to use, modify and redistribute it, but it dictates that modifications and extensions to the application should be licensed under the same terms.

Chapter 8

Future work

This chapter is a list of areas where we believe it would be interesting to refine or extend Dr. Jekyll. They are not sorted in any particular order.

The display-bitdepth problem: As explained in section 4.2.2, all the data sent to the display layer is transformed to a bitdepth of 8. While not a problem for normal visualization, this removal of information from the datasets takes time, and makes a colormap function inaccurate. It would be nice to see this limitation removed.

A portable build system: The tools used for the build system (section 3.1.6) can be configured so that users can compile Dr. Jekyll on a wide variety of platforms. However, this configuration is not fully done; the build system in version 1.0 does not handle other compilers than GCC correctly. This is not a very difficult task, but due to limited time it was pushed down on the priority list.

The voxel painting plugin does not handle too fast mouse strokes. If the user holds down the buttons and moves the mouse in order to draw a line, it is necessary to move the mouse slowly. If moved to fast, it will not draw a connected line, but single points in a row. This is because the painter plugin has no concept of drawing a line – it gets each selected coordinate from the GUI layer, applies the structure element and sets the corresponding coordinates in the cube to the given value. It is possible that the time required to do these updates is too long, allowing the mouse to move too far before the plugin finishes and the GUI system can pick up the new mouse move event.

A possible solution to this is threading, allowing the GUI layer to queue mouse events for the plugin.

Multispectral images: Version 1.0 only handles images where each pixel is represented by a single value, so-called *monospectral* images. A nor-

mal color image is *multispectral* – each pixel is represented by several different values. Extending Dr. Jekyll to handle multispectral images is not very difficult on the conceptual plane, but it is a change that would touch a lot of code. It would also require a solution to the display-bitdepth problem, since multispectral images can't be downscaled to monospectral without dramatically changing their content.

Color maps: A feature found in many visualization applications is the ability to manipulate the display colormap. The normal application of this is to allow the user to set the opacity of certain ranges of pixel values, making them translucent. This is a very convenient way of removing unwanted information from an image.

Parallel processing: When the datasets grow large, some operations start taking a long time, and the user has to wait for all of them. One of these operations is the connected components analysis and some of the calculations behind the component highlighting function. It would be very interesting to see if the application could be made multithreaded, and these functions moved out to a separate thread running in the background, in order to minimize the time the user has to wait.

Worker factory: One of the differences between the plugins and the workers is that the workers does not have an object factory handling their instantiation. Instead, `Controller` connects each worker explicitly. Since there are currently only two workers, it is not a big problem at the moment, but this approach does not scale very well. It would be interesting to see how difficult it would be to use an object factory similar to `Pluginfactory` for the workers.

A region-marking worker: The connected components analysis used in version 1.0 of Dr. Jekyll is very useful on segmented images, but not very useful on raw data. A better suited tool here would be a tool that, when the user selects a voxel, starts in that voxel and grows outward, marking all voxels with a value within a given tolerance from the selected, and stopping when the value exceeds that tolerance. This is not very difficult to implement, as it is a fairly straight-forward breadth-first search through the voxel data, but we have not had the time to do so. A tool where this functionality is implemented is MedVed [20].

User interface: There are some things that could be done better in the user interface. For instance, it would be nice if the two-dimensional visualizations could be rotated, so that the user could lay out the slices as he or she wanted.

Another nice feature would be the possibility to display two-dimensional slices that are not normal to one of the axes.

VTK-based volume rendering: As we mentioned in section 3.1.5, it would be very interesting with a volume rendering (or other kind of volume visualization) using VTK.

Three-dimensional tools: Dr. Jekyll offers some functionality for editing volumes, but the tools are generally not very sophisticated. It would be very interesting to experiment with better three-dimensional tools, and we believe the Dr. Jekyll is suited as a platform for such prototyping.

Scale information: The voxels are usually not equally long along all the axes, and the projections should zoom to display this correctly. The "open cube" dialog has fields for entering this information, but the necessary support in the visualization layers is not implemented.

Surface-based three-dimensional visualization: In addition to the volume rendering, it would be very interesting with a surface-based volume visualization, with the possibility of turning display of single components and labels on and off. Volume and surface rendering are different, and surface rendering may be better suited for many segmented datasets.

Annexes

Appendix A

Licensing

There is a huge number of libraries and applications available under free or open source licenses, giving us great freedom of choice when deciding which ones to use. Nonetheless, when writing and using open source software, it is very important to be aware of what licenses you are bound by, and what they mean for your application.

Dr. Jekyll uses several libraries, and almost all of them come with different licenses. This chapter is a summary of which libraries we use under which license, what this means for our source code, and what license we have chosen and why.

A.1 Library licenses

A.1.1 Qt

Qt is available under two main licenses. The first possibility is a commercial license, with a per-developer price per year. There are no royalties or restrictions on the use and distribution of the software developed. Alternatively, people developing software in an X-based environment, like us, can use the *Qt Free Edition*, which includes all the functionality of the commercial *Enterprise Edition*. When developing software with the free edition of Qt, the developers can choose whether they want to use it under the GNU General Public License (hereafter called the GPL) [46] or the Qt Public License (QPL) [31, 30].

We have chosen to use Qt under the QPL. It forces Dr. Jekyll to be open source software, giving people who get the binary program the right to have the source code as well, but it does not specify the exact license to use.

A.1.2 Blitz++

The Blitz++ library [3] is also available under two different licenses. It is up to the user to decide which license he or she wants to use the library under. The alternatives are the GPL or the "Blitz++ Artistic License" (the latter derived from the "Perl Artistic License").

We used Blitz++ under the artistic license, as it allows us full freedom to decide if and how we want to release our software to the public.

A.1.3 ImageMagick

The ImageMagick developers want their software to be used by as many other developers as possible, and their license reflects that. With a few basic conditions, it allows users to do almost whatever they want with the library, including using, changing, distributing and selling it.

A.1.4 Boost

The core of the Dr. Jekyll application, which is the focus of this thesis, does not currently use any Boost libraries. However, some of the plugins use the boost::shared_ptr library, which means we have to observe its license. The licensing of this library is very similar to the Blitz++ Artistic license, giving us permission to copy, use, modify, sell and distribute the software. The only requirement is that the copyright statement is not removed.

A.1.5 Coin and SoQt

We used the Coin Open Inventor implementation for the three-dimensional visualization, and the SoQt "glue class" for combining Qt and Coin. Both Coin and SoQt are made by the Norwegian company Systems In Motion (SIM), and available either under the commercial Coin Professional Edition License (PEL) or under the GPL. We used both under the GPL, which is the strictest of all the licenses affecting us.

A.1.6 SimVoleon

At the time of writing, the SimVoleon library is not released to the public. Its licensing is not determined, and we have been given access to it under a non-disclosure agreement. This is not entirely unproblematic considering that Dr. Jekyll is licensed under the GPL. However, SIM holds the copyright of the only two libraries "infecting" Dr. Jekyll with the GPL, meaning that they can allow people to use those libraries under other licenses if they want to. Likewise, we are the copyright holders of Dr. Jekyll.

In practice, by giving us permission to use SimVoleon and combine it with Coin and SoQt, SIM changed the license we use Coin and SoQt under to say that linking to the non-free SimVoleon library is OK.

We are not copyright lawyers, but we believe this solves the license problem.

A.2 The Dr. Jekyll license

Dr. Jekyll is released to the public under the GPL. This is partly because of the library licenses we are bound by, and partly because we feel that this is an opportunity to give something back to the free software community.

Bibliography

[1] 3D-DOCTOR. http://www.ablesw.com/3d-doctor/index.html.

[2] Adobe Photoshop. http://www.adobe.com/products/photoshop/main.html.

[3] The Blitz++ library. http://www.oonumerics.org/blitz/.

[4] The Boost C++ libraries. http://www.boost.org/.

[5] The Coin graphics library. http://www.coin3d.org/.

[6] The GNOME project. http://www.gnome.org/.

[7] GNU Autoconf. http://www.gnu.org/software/autoconf/.

[8] GNU Automake. http://www.gnu.org/software/automake/.

[9] The GNU Image Manipulation Program. http://www.gimp.org/.

[10] GNU Libtool. http://www.gnu.org/software/libtool/.

[11] GNU Make. http://www.gnu.org/software/make/.

[12] The GTK+ graphical user interface toolkit. http://www.gtk.org/.

[13] ImageMagick. http://www.imagemagick.org/.

[14] The independent JPEG group. http://www.ijg.org/.

[15] An interactive tool for brain imaging. http://www.lanl.gov/p/p21/mriview.shtml.

[16] The JPEG homepage. http://www.jpeg.org/.

[17] The KDE project. http://www.kde.org/.

[18] Magick++. http://www.imagemagick.org/www/Magick++/.

[19] The Matrix Template Library. http://www.osl.iu.edu/research/mtl/.

[20] Medved. http://www.simsurgery.no/technology/MedVed.html.

[21] Microsoft DirectX homepage. http://www.microsoft.com/windows/directx/.

[22] The Motif graphical user interface toolkit. http://www.opengroup.org/motif/.

[23] The numerical python homepage. http://www.pfdubois.com/numpy/.

[24] The official PNG reference library. http://www.libpng.org/pub/png/libpng.html.

[25] Open inventor homepage. http://www.sgi.com/software/inventor/.

[26] Open visualization data explorer. http://www.research.ibm.com/dx/.

[27] OpenGL homepage. http://www.opengl.org/.

[28] POOMA: Parallel Object-Oriented Methods and Applications. http://www.acl.lanl.gov/pooma/index.html.

[29] The Portable Network Graphics (PNG) homepage. http://www.libpng.org/pub/png/.

[30] Q public license (annotation). http://www.trolltech.com/licenses/qpl-annotated.html.

[31] The Q public license version 1.0. http://doc.trolltech.com/3.1/license.html.

[32] qmake user guide. http://doc.trolltech.com/3.1/qmake-manual.html.

[33] Qt whitepaper. http://www.trolltech.com/products/qt/whitepaper.html.

[34] SGI OpenGL overview. http://www.sgi.com/software/opengl/overview.html.

[35] SimSurgery AS. http://www.simsurgery.no/.

[36] Standard template library programmers guide. `http://www.sgi.com/tech/stl/`.

[37] Systems in motion homepage. `http://www.sim.no/`.

[38] The Tcl Developer Xchange. `http://www.tcl.tk/`.

[39] The Template Numerical Toolkit. `http://math.nist.gov/tnt/`.

[40] TGS homepage. `http:///www.tgs.fr/`.

[41] Trolltech. `http://www.trolltech.com/`.

[42] The visualization toolkit (VTK) homepage. `http://www.vtk.org/`.

[43] The Viz data visualization tool. `ftp://ftp.ffi.no/spub/stsk/viz/index.html`.

[44] The wxWindows graphical user interface toolkit. `http://www.wxwindows.org/`.

[45] The X.Org consortium. `http://www.x.org/`.

[46] GNU General Public License. `http://www.gnu.org/licenses/gpl.html`, June 1991.

[47] GNU Lesser General Public License. `http://www.gnu.org/licenses/lgpl.html`, February 1999.

[48] Java specification request 14 – add generic types to the java programming language. `http://jcp.org/en/jsr/detail?id=14`, May 1999. Slated for inclusion in the 1.5 specification of Java, which has not yet been released.

[49] ALEXANDER, C., ISHIKAWA, S., SILVERSTEIN, M., JACOBSON, M., FIKSDAHL-KING, I., AND ANGEL, S. *A Pattern Language*. Oxford University Press, New York, 1977.

[50] ALEXANDRESCU, A. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001.

[51] AURDAL, L. *Analysis of Multi-Image Magnetic Resonance Acquisitions for Segmentation and Quantification of Cerebral Pathologies*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Mar. 1997. ENST 97 E 034.

[52] COLLOFELLO, J. S., AND ORN, M. A practical software maintenance environment. In *Proceedings of the Conference on Software Maintenance, 1988.* (Scottsdale, AZ, USA, 1988), pp. 45–51.

[53] DALHEIMER, M. K. Design patterns in Qt. *The O'Reilly Network* (October 2002). `http://www.onlamp.com/pub/a/onlamp/2002/01/10/designqt.html`.

[54] DEKLERCK, R., SALOMIE, A., AND CORNELIS, J. An editor for 3d medical volume images. In *TASK-Quarterly* (October 1997), vol. 1, pp. 155–162.

[55] FOLEY, J., VAN DAM, A., FEINER, S., AND HUGHES, J. *Computer graphics: principles and practice*, 2nd. ed. The Systems Programming Series. Addison-Wesley Publishing Company, 1996.

[56] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, October 1994.

[57] GARCIA, R. The boost multidimensional array library. `http://www.boost.org/libs/multi_array/doc/index.html`.

[58] GREENFIELD, P., MILLER, T., HSU, J.-C., AND WHITE, R. L. An array module for python. In *In proceedings of the XI international conference on Astronomical Data Analysis Software and Systems* (October 2002).

[59] ISO/IEC. *ISO/IEC 8652:1995 Programming Languages – Ada 95*, 1995.

[60] ISO/IEC. *ISO/IEC 14882:1998 Programming Languages – C++*, 1998.

[61] ISO/IEC. *ISO/IEC 9899:1999 Programming languages – C*, 1999.

[62] JOSUTTIS, N. M. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[63] KERNIGHAN, B., AND RITCHIE, D. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[64] LINDVALL, M., TESORIERO, R., AND COSTA, P. Avoiding architectural degeneration: an evaluation process for software architecture. In *Proceedings of the eighth IEEE Symposium on Software Metrics* (2002), IEEE, pp. 77–86.

[65] MEYERS, S. *Effective C++*, second ed. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997.

[66] MROZ, L., AND HAUSER, H. RTVR: a flexible java library for interactive volume rendering. In *Proceedings of the conference on Visualization 2001* (2001), IEEE Press, pp. 279–286.

[67] NEY, D., AND FISHMAN, E. Editing tools for 3d medical imaging. *IEEE Computer Graphics and Applications 11*, 6 (1991), 63–71.

[68] PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (1972), 1053–1058.

[69] PRECHELT, L., UNGER, B., TICHY, W. F., BRÖSSLER, P., AND VOTTA, L. G. A controlled experiment in maintainance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering 21*, 12 (December 2001).

[70] RESEARCH SYSTEMS, INC. The interactive data language. `http://www.rsinc.com/idl/`.

[71] REZK-SALAMA, C., ENGEL, K., AND HIGUERA, F. V. The OpenQVis project. `http://openqvis.sourceforge.net/`.

[72] SELAND, J. S. Post-processing of segmented volumetric image datasets. Cand. scient. thesis, Department of informatics, University of Oslo, Norway, May 2003.

[73] SIONG, O. C., AND PRAKASH, E. C. Implementation of a java based volume browser for 3d volume graphics. In *TENCON 99. Proceedings of the IEEE Region 10 Conference* (Cheju Island, South Korea, September 1999), vol. 1, IEEE, pp. 686–689.

[74] STROUSTRUP, B. Bjarne Stroustrup's FAQ. `http://www.research.att.com/~bs/bs_faq.html`.

[75] STROUSTRUP, B. Why C++ isn't just an object-oriented programming language. In *OOPS Messenger* (October 1995), vol. Addendum to OOPSLA'95 Proceedings.

[76] VAN HEESCH, D. The Doxygen documentation system. `http://www.doxygen.org/`.

[77] VELDHUIZEN, T. L. Expression templates. *C++ Report 7*, 5 (June 1995), 26–31. Reprinted in C++ Gems, ed. Stanley Lippman.

[78] VELDHUIZEN, T. L. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)* (1998), Lecture Notes in Computer Science, Springer-Verlag.

[79] WOSSUM, G. autoqt. `http://autoqt.sourceforge.net/`.

[80] WU, Y., AND YENCHARIS, L. Commercial 3-d imaging software migrates to pc medical diagnostics. *Advanced Imaging Magazine* (October 1998), 16–21.