University of Oslo
Department of Informatics

Post-processing of segmented volumetric image datasets

Johan Simon Seland

Cand Scient Thesis

April 2003

Acknowledgements

The process of writing this thesis has been a long time in coming, and naturally many people deserve acknowledgments. First and foremost I wish to thank my parents, Anne Kå. Seland and Leif Nilsen for allowing me to pursue my own interests both academic and recreational.

Secondly Kristoffer Gleditsch who has been my “partner in crime” for the entire project. WE DID IT!

Thirdly, great thanks to the free software community, which has created almost all the tools used in the *Dr. Jekyll* project. I hope I have been able to give something back by releasing this project under the GPL.

The *Interventional Centre at Rikshospitalet University Hospital, Oslo, Norway* deserves mention for hospitality and for providing such a great working environment.

Merci beacoup to Daniela Becker, Thierry Gérraud, Didier Verna, Olivier Ricou and Akim Demaille (in no particular order) at *EPITA Research and Development Laboratory (LRDE), Paris, France* for letting us spend the spring in Paris and teaching me more about the C++ template mechanism that I ever wanted to know.

Thanks Morten Eriksen and Espen Rideng from *Systems in Motion, Oslo/Trondheim, Norway* for giving me early access to their volume rendering library. Keep up the good work!

Great thanks Michael Floater and Øyvind Hjelle at *SINTEF Applied Mathematics, Oslo, Norway* for giving pointers to information about triangles and surfaces.

Thanks to the always enthusiastic Hans Petter Langtangen at *Simula Research Laboratory, Fornebu, Norway* for resources about scripting languages, as well as for motivation.

Our internal advisor Knut Mørken from the *Department of Informatics at the University of Oslo* has been helpful in keeping us on track for the entire project, giving good advice when needed.

The biggest thanks however goes to our supervisor Lars Aurdal for starting of this project in the first place, and also for always believing in us.

Abstract

Description: This work concerns the problem of post-processing segmented volumetric images. Such images can arise in fields as diverse as medical imaging, material science, geology and other fields. An application in which to experiment with different post-processing methods has been written, named *Dr. Jekyll*. The input images to this application is assumed to be segmented before it is invoked.

A survey is given of the design goals of the application, as well as an overview of why the C++ language and a selection of libraries were chosen for the implementation.

Special focus is given to the design and implementation of algorithms in *Dr. Jekyll*. Effort was made to make the implementation as generic as possible, without sacrificing runtime speed. It is demonstrated how it is possible to use the template mechanism of C++ to achieve this goal.

Classical imaging algorithms, such a connected components analysis and mathematical morphology, is traditionally applied to binary or gray level images. This work formulates a version of these algorithms for segmented images.

Both the application and the algorithms implemented are generic in the sense that they are not tied to a particular field or imaging modality.

Furthermore the application is designed to be extendable, and to provide generic mechanism to implement other image processing tools. To further encourage such extendability the application is licensed under an open source license.

Keywords: Manual post-processing, segmented images, software.

Contents

Chapter 1

Introduction

The human visual system is the most important system through which our brain gathers information about our surroundings. While using our vision may seem like an intuitive process, complex mechanisms are involved in “decoding” the light hitting our retina into a meaningful model of the world. Humans are able to do this “decoding” whether they see the real world (light reflected of objects), or if they see a 2-dimensional image.

Programming a computer to do the same “decoding” automatically seems to be a very ambitious goal, which we are currently very far from reaching. The computing power of a modern microprocessor is however immense, and the work detailed in this thesis is to make a system where a computer and a human operator interacts to make an optimal “decoding.”

1.1 Overview of the thesis

This chapter serves as an introduction to the problem of post-processing segmented images. Furthermore, since the work detailed in this thesis is at the junction between several distinct disciplines within computer science, some terminology used throughout the thesis is explained at the end of this chapter.

The second chapter is an overview to the *Dr. Jekyll* application. *Dr. Jekyll* is the application which was developed as part of this thesis project. The chapter surveys the design goals, the high-level design, and the design patterns used. Chapter two concludes with a discussion of the languages and libraries which were used for implementing the application.

Chapter three details the various algorithms implemented, including their theoretical background and their extension to handle segmented images. Furthermore the concrete implementation and usage of the algorithms within *Dr. Jekyll* is ex-

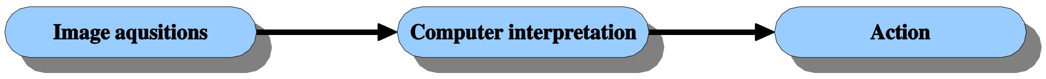


Figure 1.1: The image analysis process in an ideal world. An image is acquired, and the computer processes the image automatically.

plained. A summary of various visualization techniques for segmented images concludes the chapter.

Chapter four discusses the findings related to implementing and using these algorithms for interactive post-processing. These findings include some pitfalls which should be avoided when implementing algorithms.

Chapter five concludes the thesis, and discusses how well suited the language and libraries chosen were for the task at hand. Some possible directions for future work, both regarding the *Dr. Jekyll* application itself, as well as other fields, conclude the thesis.

Appendix A includes data from some benchmarks which were run.

The source code to the *Dr. Jekyll* application is *not* included in an appendix. The source code totals at around 15000 lines of C++ code, in addition about 5000 lines is generated by preprocessors. It would thus take several hundred pages to list it all. Interested readers can find it at `http://drjekyll.sourceforge.net`.

1.2 Introduction to the problem

With a fully automated system, image data would be acquired and read into the computer. Once this step has completed, a program will run unaided, and present us with a “perfect” model of the given data. Figure 1.1 illustrates these steps. This model, which can take many forms, can then be used to decide upon a further course of action.

A major problem occurs even in the early stages of the automated interpretation process, namely at the *segmentation* step. In this phase the various intensity levels of an image are assigned to distinct labels, hopefully sharing some physical or statistical property. This is illustrated in figure 1.2. This CT-slice of the abdomen has been segmented using a Markov field based segmentation algorithm[3]. We can see that the kidney (blue) has been assigned the same label as the soft tissue interior of the spinal column (off-white). Also the outside of the CT-gantry has been assigned the bone label. While the liver (yellow) is fairly well segmented, a lot of other regions have also received the liver label, making a direct visualization of the liver almost impossible.

Using this segmentation solely as the basis of a physical interpretation of the image data is apparently not sufficient. As a foundation for the calculation of sizes and placement of organs, it would lead to erroneous results.

This problem arises because segmentation algorithms are based on the assumption that enough information is included in the actual image to establish a valid model of the world. A human would use his conceptual knowledge of the data in addition to the information in the image to arrive at a reasonable image classification. To achieve this on a computer we must somehow add *a-priori* information to the image data. Several methods has been proposed to add this information:

- Better segmentation algorithms

  An active field of research is *model based segmentation*. Here one is not only focusing on the spectral properties of the pixels in the actual image, but also on previous knowledge about the shape and form of the dataset. This makes most model based segmentation algorithms highly specialized. Also, these algorithms can be fooled by images deviating from the expected. Nature shows great diversity, and often it is the not so common cases that are most interesting. An algorithm based on the “normal” shape could fail with such a dataset.

  While certain advances have been made in model based segmentation, generic segmentation systems able to analyze a broad range of data can not be expected in the near future. [54]

- Manual and semi-automatic segmentation

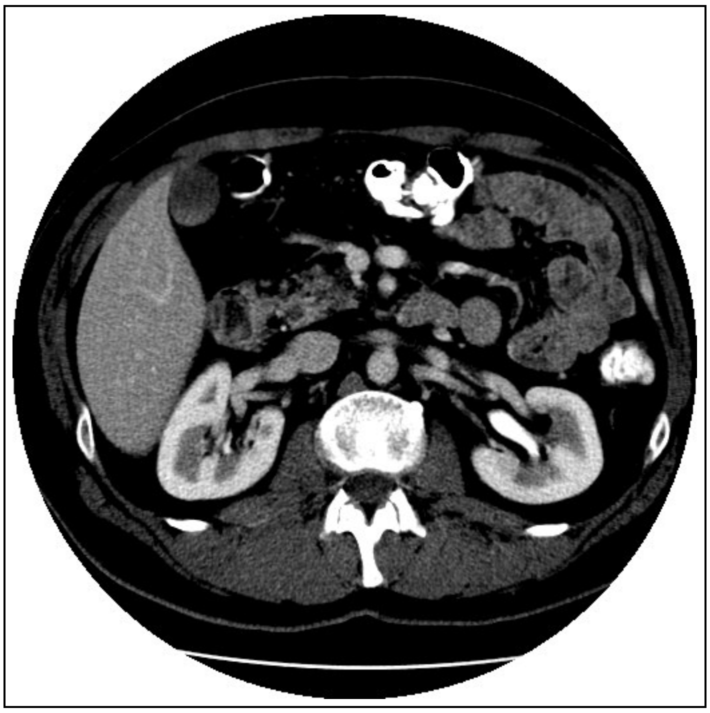
  During manual segmentation, pixels are grouped directly on the image using an image-editing program like Adobe Photoshop or Gimp or other more specialized programs. Manual segmentation is very labor intensive and time consuming, and becomes infeasible as the datasets grow larger. These programs are inherently 2D-based, and manually segmenting a volumetric dataset is an infeasible task.

  One possibility is to use semi-automatic methods, were the the computer is used at what it is good at, like contour detection, while at the same time having manual control.

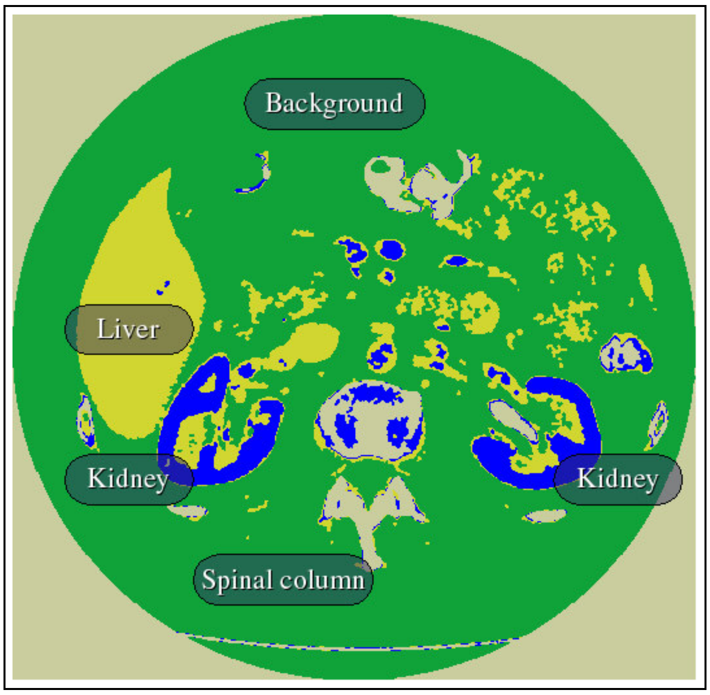
  Research also shows that manual segmentation produces poor reproducibility; for MR images, studies show that there can be up to 15% difference in manual segmentations generated by different experts[3].

- Manual postprocessing

  Another approach is to manually postprocess the (incorrectly) segmented images in an image-editing program. Many of the same problems with



(a) Raw slice from CT-scanner. (12-bit greyscale)



(b) Segmented slice using a Markov-fi eld based segmentation algorithm.

Figure 1.2: Raw and segmented image of a CT-slice of the abdominal. The goal of the segmentation was to extract the liver component. As image b) illustrates, many other organs were assigned to the liver label, and the interior of the spinal column was assigned to the same label as the kidney. The circular shaped disk around the images is called the CT-gantry. (Images courtesy of IVS, RH)

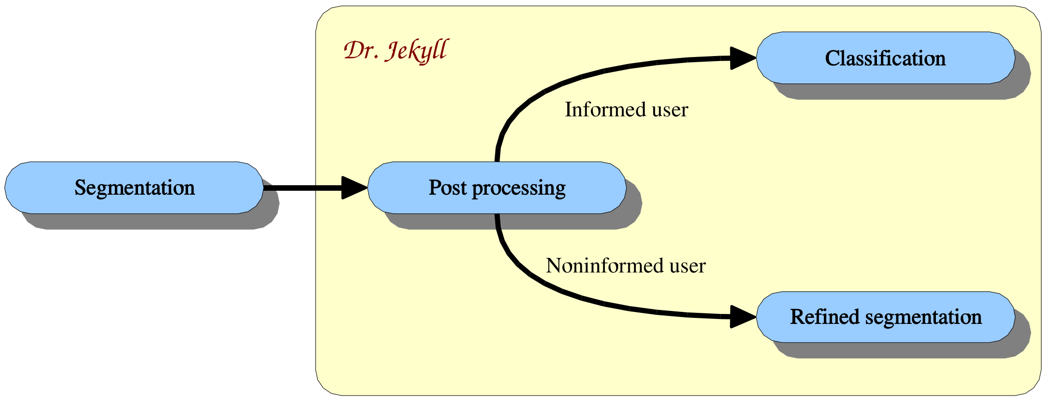


Figure 1.3: The workflow of the manual postprocessing task. After an automatic segmentation, the user invokes the *Dr. Jekyll* application. If the user is experienced in the field of the data, the end result may be a classification, where the labels have been assigned a physically meaningful interpretation. If the user have is not knowledgeable in the field of study, he might yet refine the segmentation by removing noise.

manual segmentation also arises here; it is still a time consuming and error prone task.

While the holy grail of image analysis is generic and correct fully automated segmentation algorithms, this is not an available solution today. A short term solution to classifying large datasets could be to implement better tools for manual postprocessing, utilizing the computing and visualization power available on a modern workstation. This thesis outlines the design and algorithms for such an approach.

1.3 Manual postprocessing

Exploration of the manual postprocessing approach is the aim of this thesis. The design and development of such a program, called *Dr. Jekyll*, was undertaken as part of the Cand. Scient. work by the author and fellow student Kristoffer Gleditsch. His thesis titled *Interactive manipulation of three dimensional images* [17] delves much deeper into details about the design and concrete implementation of *Dr. Jekyll* as an application. It also details how the application can be used as a an application framework for writing new “plugins”,

The *Dr. Jekyll* application is the result of a team effort by two developers: Kristoffer Gleditsch and the author. Both have put a lot of effort into the development of *Dr. Jekyll*, but with different focus: Gleditsch[17] has been working with the overall design of the application and the plugin interface. The author 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.

The name *Dr. Jekyll* arises naturally, since the segmentation tool written by Lars Aurdal as part of [3] was named *Mr. Hyde*. This program generates output which can be used as input for our *Dr. Jekyll* application. *Mr. Hyde* was originally written to process MRI images from the human brain and thus explores the duality of the human mind, the theme of the classic novel *The Strange Case of Dr. Jekyll and Mr. Hyde* by Robert Louis Stevenson.

Figure 1.3 illustrates the various steps in manual postprocessing. First the images are segmented using one of several segmentation algorithms. Thereafter post-processing takes place. If the user performing the post-processing is informed in the field the image arises from, the end-result of the post-processing can be a fully classified image. On the other hand, a non-informed user may refine the segmentation, by removing noise etc.

This thesis details the implementation and usage of the image processing algorithms implemented as part of *Dr. Jekyll*.

The problem with both manual segmentation and manual postprocessing seems to be a lack of effective programs, particularly for volumetric datasets. While segmentation algorithms do produce incorrect results (as in figure 1.2), they still provide a better starting point than the raw images. This is because segmentation algorithms are good at separating background from foreground, however pixels may wrongly grouped.

The algorithms needed for the two approaches also differ greatly. Segmentation algorithms must consider a wide variety of spectral intensities, while a postprocessing algorithm typically work on just a reduced set of labels.

However, programs for both approaches would also share some properties. The data is fundamentally stored as a vector field, and visualization front ends could work on both types of data sets. *Dr. Jekyll* has been designed to be flexible, and it could probably be extended to a manual segmentation tool as well. This extension is beyond the scope of this thesis.

1.4 Terminology

As mentioned above, this thesis combines techniques from several distinct fields within computer science. Consequently some of the terminology may be unknown to the reader, or the reader might be used to seeing them in a slightly different context. The following section serves as an overview of this terminology, and tries to be consistent with other literature in the field. Most of the definitions in section 1.4.2 are from [47].

1.4.1 Programming terminology

For implementing the application C++ was used (see section 2.2.1 for a discussion of why C++ was chosen.) C++ supports many different styles of programming, and touts itself as a “multi-paradigm” language [50, 49, 48].

The following terminology presents how these concepts are used in the literature, although the reader should be aware that different programming languages have different names for some of these techniques.

Also some most programming languages has different forms of “syntactic sugar” which greyes out the boundaries of these concepts somewhat.

Object and class

A class is a collection of data *and* methods for accessing those data. Unlike traditional functional programming where data and functions are loosely coupled, a class symbolizes a tight coupling.

An advantage of tight coupling is that he user of a class need not know how the class is implemented internally. The internals of a class might change without affecting other parts of the program.

An object is an *instance* of a class, objects of the same class can be instantiated and accessed independently of each other, just like standard datatypes like integers and floats.

Method, function and functor

A function is a standalone piece of code which accepts a series of input arguments and returns a value based solely on the input values. A well-behaved function is not supposed to change any other part of the program. [1]

Unlike a function a method is tied to an object (instance of a class). Like a function it can accept a series of input arguments, and return a value, but it may also change the state of the object. Methods usually act upon a concrete instance of a class, and either queries or change certain aspects of the instance.

[1]However most computer languages allows for so called *static* variables in functions which are preserved throughout the program, thus a function might return a different value for the same parameter list.

A *functor* (for function object) is an object which can be called as if it is a function. The advantage of functors is that they can be passed as arguments to other functions (or methods, or functors). This can also be accomplished by C-style function pointers, but functors provide a type safe mechanism. In C++, a functor typically implements the `operator()` method, an example of operator overloading (see below).

Subclassing, inheritance and polymorphism

In addition to providing a convenient modularization of data and methods for accessing these data, a major part of the object oriented paradigm is the notion of inheritance. A class can be a subclass of another class. When a class is a subclass it inherits all the public properties of the original class, but it may also extend the functionality of the superclass (the parent class).

Subclassing therefore usually signalizes an IS-A relationship between classes. If a class `B` is a subclass of a class `A` then a `B` object can be used everywhere an object of the class `A` can be used, but not the other way around. This is also called the Liskov Substitution Principle [30].

Subclasses can of course be subclassed further, which leads to an inheritance hierarchy.

C++ also has other kinds of subclassing (private and protected inheritance) as well. These do not signalize an IS-A relationship, but as an implementation detail. In this thesis, subclassing always means a strict IS-A relationship.

Pointer and reference

A pointer is a variable holding a memory address, The type of the pointer is the type of data found at the memory address which the pointer is holding.

A reference is an alias (alternate name) for an object.

Both pointers and references can be thought of as a handle to data, a mechanism which lets the programmer “point” to data at a memory location.

Handles are used to avoid copying datastructures between methods and functions, thus avoiding copying the data which is being passed and letting the receiving method or function alter the data directly instead of a copy.

This method of parameter passing is typically used for large datastructures, where there can be considerable run time cost associated with copying the data.

Such passing of data is usually called *pass by reference* in contrast to *pass by value* where the data is copied and the original data is not altered in the called function.

It should be noted the C++ makes a distinction between pointers and references,

```
void swap(int& x, int& y)
{
  int  tmp = x;
  x = y;
  y = tmp;
}
int  a  = 5,  b = 10;
swap(a, b);
```

```
template<class T>
void swap(T& x, T& y)
{
  T tmp = x;
  x = y;
  y = tmp;
}
int  a  = 5,  b = 10;
double e = 2.71828, pi = 3.1415;
swap(a, b);
swap(e, pi );  //  Legal C++, though not in mathematics.
swap(a, pi );  //  Not legal  C++, incompatible types.
```

Figure 1.4: C++ code examples illustrating the difference between a traditional type-based function and a template based version. The type-based version has to be written for all datatypes which we would want to swap. The template based version can be written once and the compiler will expand it for all types. (The types must however support assignment semantics.)

but in the end they are (mostly) just different syntax for achieving the same end effect.

Templates and generic programming

Templates are a way of parameterizing the *datatypes* a function or class can use. They are in short a form of automated code generation where a family of classes or functions which look the same can be written once for all datatypes. Figure 1.4 gives an example of how a generic function swapping the contents of two variables can be written. Such an approach to programming is called *generic programming*. Templates is a feature not found in very many languages, but both Ada[22] and C++ provides variations of the mechanism and support the generic programming paradigm.

A very attractive use of templates is to write generic containers, and algorithms which can work over these containers. The C++ Standard Template Library is one example of such a library, and it is detailed in greater depth in section 2.2.1.

In order to avoid compromising the type-safety of the language or incurring a runtime overhead, templates are expanded, type checked and optimized for all types which call the template methods at compile time. Therefore template introduces no runtime penalty, but compilation times tend to be longer, since the compiler has to figure out all possible types a template function can be called with. The end result is that the same source code need to be compiled multiple times.

Operator overloading

Almost every programming language support a set of operators for its built in types. A typical example of such an operator is the "$+$" operator used for adding numeric types together. However, when it becomes possible to have user defined types (often implemented as a class), some languages (including C++) allow for redefining the meaning of the basic operators. Such redefining is called *operator overloading.*

In C++, the functions which implement operator overloading is called `operator@`, where `@` designates the operator to be overloaded.

Like any language feature, operator overloading has its uses and its misuses. An example of proper use of operator overloading is to implement arithmetic for complex numbers. When writing generic algorithms, operator overloading becomes a necessity, since it is the operators a class support, and not its type which decides if the generic algorithm can work on the type.

An objection to operator overloading, is that when reading source code, it is no longer obvious when a user defined function is called, making it harder to mentally visualize program flow.

1.4.2 Image processing terminology

Image

Definition 1 *An image f is a mapping from a rectangular subset $\mathcal{D}_f$ of $\mathbf{Z}^n$ into a bounded finite set of integers, N_0. By convention these integers are nonnegative,*

$$f : \mathcal{D}_f \subset \mathbf{Z}^n \longrightarrow \mathbf{N}_0. \tag{1.1}$$

Using this definition we avoid the distinction between grey tone images and binary images. A binary image is simply an image where $\mathbf{N}_0 = \{0, 1\}$ (or any other set of two distinct integers). In practice, n is almost always one of $2, 3$ or 4. A 4-dimensional image is a video of a 3-dimensional image. Color and multispectral images are simply arrays of images sharing a common definition domain.

Pixel and voxel

Each of the single discrete elements that together constitute an image are called pixels (from picture element). Unlike points from Euclidean geometry, pixels do not have infinitesimally small surface area, since they typically represent the mean luminance value of a continuous sampling area on a discrete regular grid.

Pixel is a term that is used in many different contexts, a formal definition accommodating all uses of the word pixel does not exist. The resolution of an image is typically given in pixels, but when displaying the image on a computer monitor, the image might be scaled or stretched so there might not be a direct correspondence between screen pixels and image pixels, even if they both represent the same image.

Using our definition of an image, a pixel is a unique position in the definition domain and a value from the value domain of the image.

When dealing with volumes, the discrete element is a voxel (volume element). A voxel is nothing but a pixel in $\mathbf{Z}^3$. In this thesis we will use the term voxel only when specifically dealing with 3-dimensional images, and pixel in the generic case.

Segmentation and classifi cation

Segmentation and classification are two terms with different meanings for different contexts. For image analysis, these definitions are often used:

Defi nition 2 *Segmentation is an image to image transformation taking as inputs images defined over the definition domain $\mathcal{D}_f$ and generating as output images over the same definition domain with values in the interval $\{0, \ldots, l_{max}\}$ (where l_{max} is typically a relatively small number). Values in the segmented image are assigned so as to indicate the membership of pixels in the original image to groups of pixels sharing some kind of (spectral or statistical) property.*

Notice that the values in the segmented image have no correspondence to the luminosities of the original image. Nor do values close to each other indicate that pixels with those values share any properties. The values are simply unique labels for pixels sharing one or more properties.

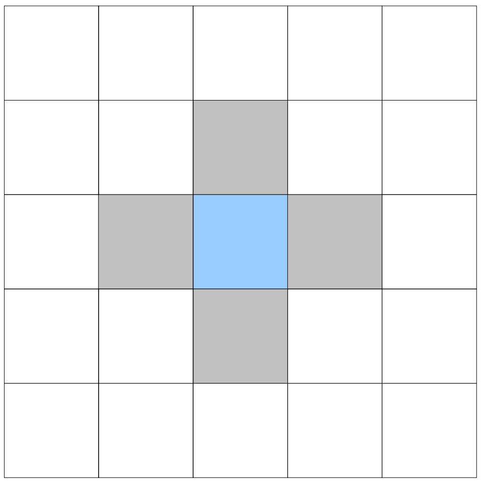


Figure 1.5: A pixel (in blue) and its neighbors in 4-connectivity (in gray).

Definition 3 *Classification is to assign each pixel in the input data to one of a finite number of known categories or classes. Each such class has a physical (or other semantically meaningful) interpretation.*

Segmentation is not the same as classification. Pixels in a segmented image share some spectral property, but no physical interpretation has been given to the different segments. An image that has been classified however, is given a physical interpretation, based on information *not necessarily contained* in the original image.

Notice that for volume rendering the term classification designates the process of assigning opacity to a voxel[29].

Graph, neighbor, path and connectivity

Several algorithms involve computing some neighborhood relationships between pixels. In order to define such relationships the notion of a graph if necessary.

Definition 4 *A non-oriented graph $\mathcal{G}$ is a pair (V, E) of vertices V and edges E where*

$$\begin{aligned} V &= (v_1, v_2, \ldots, v_n) && \textit{is a nonempty set of vertices,} \\ E &= (e_1, e_2, \ldots, e_m) && \textit{is a set of unordered pairs } (v_i, v_j) \textit{ of vertices.} \end{aligned}$$

A graph is said to be simple *if it does not contain any loops and if there exists no more than one vertex linking any given pair of vertices.*

All graphs used in the processing of images are simple, and they are usually sampled on a regular grid, called a digitization network. An image pixel is then the same as a vertex.

Graphs have been the subject of much study during the last century, and many interesting properties have been derived.

Definition 5 *The neighbors of a pixel v in a graph $\mathcal{G} = (V, E)$ are denoted $N_\mathcal{G}(v)$,*

$$N_\mathcal{G} = \{v' \in V \mid (v, v') \in E\}.$$

Two pixels are neighbors if they share a common edge.

Definition 6 *A sequence $(v_0, v_1, \dots, v_l)$ of distinct vertices of a graph $\mathcal{G}$ is a* path *of length l if v_i and v_{i+1} are neighbors for all i in $(0, 1 \dots, l-1)$.*

Definition 7 *A subset of an image is called* connected *if for any two pixels p and q of the subset there exists a sequence of points $p = p_0, p_1, \dots, p_n = q$ of the subset such that p_i is a neighbor of p_{i-1}, $1 \leq i \leq n$.*

The most frequently used connectivities in image analysis are 4- or 8-connected graphs for 2-dimensional images, and 6- or 26-connectivity for 3-dimensional volumes. An illustration of a pixel an its 4-connectivity is given in figure 1.5.

1.4.3 Visualization

For a computer an image is simply a series of numbers, for a human however, it is necessary to “translate” the image into visual stimuli in order to comprehend it. This process is called visualization, and during the last years it has emerged both as a field of its own, as well as a helpful tool for many different disciplines.

For a 2-dimensional image it is possible to make a one-to-one correspondence between data coordinates and screen coordinates. 3-dimensional images must either be considered slice by slice, or by using some form of projection of the 3-dimensional image onto the inherently 2-dimensional computer monitor. Such projections always result in distortions of the image. Various visualization techniques are explored further in chapter 3.6.

Chapter 2

Overview of Dr. Jekyll

Dr. Jekyll is the application implemented as part of this project.

This chapter gives an overview of the design of *Dr. Jekyll*. It starts with design goals, then goes on to the high level design. The last part of this chapter discusses the concrete language and libraries selected for the implementation. That section also includes a short summary of other competing technologies that could have been used.

A discussion of how well suited the languages and libraries were for the given task at hand is delayed to section 5.3.2.

2.1 Technical goals

At the start of the project, several technical goals were defined:

- Interactivity. *Dr. Jekyll* is designed for interactive use, and the user-experience is important. Manual post-processing is an iterative process, and the user must be able to experiment with various algorithms in an interactive environment to see if the desired result is achieved.

- No hard-wired limitations on the size of datasets (except the hard limit imposed by the amount of RAM). Typical datasets are large today, and expected to grow in the future.

- Scalability and portability. To be able to utilize advances in hardware technology, it is crucial to write code that is not tied to a particular hardware platform or operating system. Incorporating new hardware technology should only be a matter of recompiling the source code.

This is particularly important with the advent of 64-bit computers. On 32-bit computers it is not possible to address more than 4GB of memory. Datasets of this size are not uncommon today, applications written for 32-bit architectures using very large datasets therefore only holds a subset of the dataset in memory at a given time[60]. Section 3.1 ahead discusses image sizes to a larger degree.

With 64-bit computers the limitation is increased to 18 Exabytes ($18 \cdot 10^9$GB).

➥ Extendability and modularity. It should be possible to extend *Dr. Jekyll* with new algorithms and functionality, without affecting the other parts of the program. Writing such an extension should be possible without knowledge of the internals of the program. New image processing tools are a typical example of such *plugins*.

➥ Don't reinvent the wheel. There are many high quality libraries available to solve a variety of tasks. There is no reason to duplicate functionality that already exists in a library.

While this might seem like a lot of buzzwords, each of them has a purpose and has greatly influenced the design of *Dr. Jekyll*.

Another equally important design goal, which is not technical but nevertheless has great influence on the techniques chosen was to make the application freely available.

In order to allow for multiple developers and extendability, the modular design illustrated in figure 2.1 was agreed upon. Each module communicates with each other through clearly defined interfaces, and has distinct responsibilities.

The various modules are summarized below:

➥ Kernel

The main responsibility of the kernel is to initialize the other modules, and set up message passing between them. Another important part of the kernel is the loading and storing of the actual image data.

In order to support an extendable design, it should be possible to write a new plugin without altering any old code, in effect the kernel doesn't know about any concrete plugins, only their common interface. One way to achieve this is to have *object factories*[15]. A plugin registers itself with the factory at runtime, and the kernel holds a list of all available plugins. Object factories are further detailed in section 2.1.2.

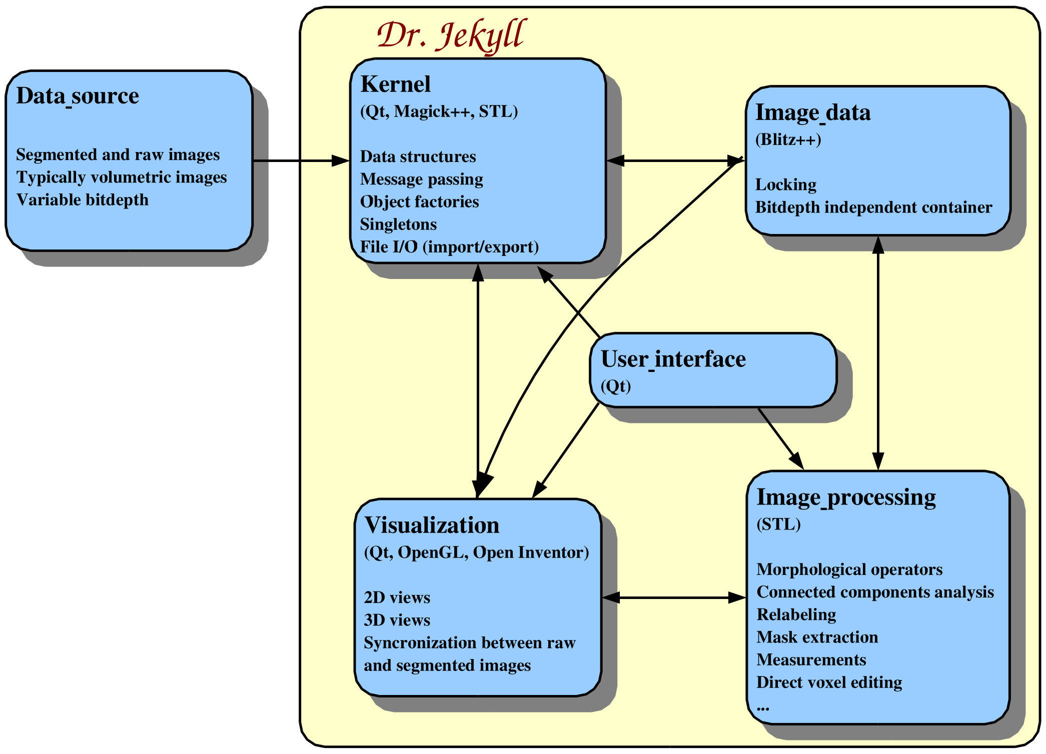


Figure 2.1: A high level overview of *Dr. Jekyll*, showing what libraries are used in each part of the program. This figure has no direct correspondence to the actual classes of the application.

- Image data

  The image data module stores the actual values of the input data. Since several visualizations and plugins may access the data at the same time, it is necessary to have some sort of locking mechanism on the data. The purpose of the locking mechanism is twofold. Only one plugin may alter the data at any given time, and visualizations must be told when the data has changed so they can update themselves.

  Another important feature of the image data modules is efficiency regarding memory usage. Since data may have various bitdepths, it should be possible to store data with the right bitdepth.

- Visualization

  Since the purpose of *Dr. Jekyll* is image manipulation, effective visualization tools are vital. Many of the image processing algorithms are interactive by nature and need to get data coordinates from the user. The visualizations should be able to provide these.

In order to generate (near) optimal segmentations, it is necessary to see both the original raw data and the segmentation which is being processed at the same time.

For 3-dimensional data, different visualizations may reveal different structures in the images. There is not a single visualization which will always yield the best result.

➥ User Interface

The user interface must allow for intuitive access to the various modules. It must be easy to find the desired functionality, and no task should be repetitive.

➥ Image processing

The actual alteration and analysis of the image data are the purpose of the program. The image processing tools communicate with both the visualization and image data module in data coordinates. The kernel and visualization programs should not know about the different image processing tools.

Actual processing tools are explored in detail in chapter 3.

2.1.1 Modular and object oriented design

A modular design like the one outlined above lends itself naturally to *object oriented programming (OOP)*. The common parts of for example an image processing or visualization tool, can be expressed as an abstract base class. For an overview of object oriented terminology, see [49].

Scientific programs, where the developers often have a background from physics or mathematics, in contrast to computer science, have often been written in procedural languages like Fortran. Using the procedural approach is an intuitive implementation technique, and at the abstraction level is quite close to mathematical notation. The problem is that procedural programming involves too many visible details, and small changes to one part of the code could lead to substantial modification of existing code. With this comes the danger of introducing new bugs in old code.

Studies show that object oriented programming is far superior to the procedural approach, even for scientific programs [27]. One reason for the slow acceptance of object oriented programming in the scientific community is that OOP techniques have been to slow for numerical calculations. Advances in compiler technology are closing the gap, and today the benefits of OOP outweighs the disadvantages [7].

The usage of the generic programming paradigm, has also been proven to decrease runtime speeds for some tasks, since it allows for more aggressive inlining of function calls than traditional programming[36].

2.1.2 Design patterns

Design patterns are proved solutions to common problems that arise during software design and implementation. The concept of *design patterns* in programming sprang out of the object-oriented community in the early nineties, so it is a relatively new topic in computer science. Fundamental to any discipline is a common vocabulary for expressing concepts, and a language for relating them together. One of the goals of design patterns, is to create such a vocabulry, to help software engineers solve recurring problems throughout all of software development.

It is important to notice that to be designated a design pattern, the pattern must have been applied successfully in more than one system. Patterns thus aim to be an example of “best-practice” for a concrete problem. Those patterns that are proved not to work in practice, are therefore called an *anti-pattern*, and they can be seen as a lesson learned.

A pattern has four essential elements;

1. A *name* we can use to describe the pattern and to increase our design vocabulary.
2. A *problem* which describes when to apply the pattern, including its context.
3. A *solution* describes the elements that make up the design. The solution is not a concrete design or implementation, but an abstract description of a design and a general arrangement of elements.
4. A list of *consequences*, which are the results and trade-offs of applying the pattern.

In order to be useful, authoritative documentation of many different patterns are necessary. The classical catalog of such patters is the book *Design Patterns* by Gamma et al (often called the Gang of Four) [15]. The following summary of patterns used in *Dr. Jekyll* uses the names from that book.

A tenet within the pattern community is that it is impossible to represent patterns in code. However recent work by Andrei Alexandrescu shows that it is possible to

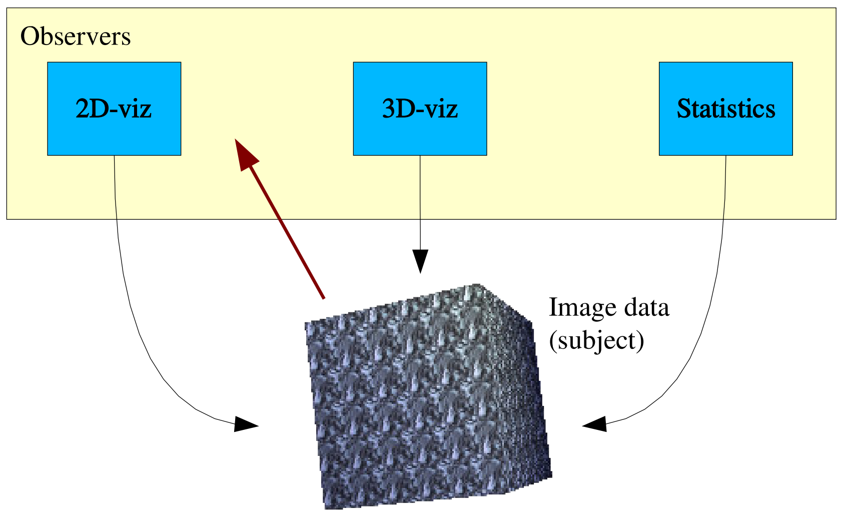


Figure 2.2: The observer pattern; three different modules listens for changes to the image data, the image data does not know anything about the observers or their number.

automate the task of pattern *implementations* (but not the pattern itself) using generic programming [2, 61, 62]. The implementation of the factory pattern within *Dr. Jekyll* is based on such an approach.

Below is a summary of the patterns used in *Dr. Jekyll*. A brief introduction to what the pattern does is included, as is a description of where it is applied within *Dr. Jekyll*. However for an indepth description, see *Design Patterns* [15].

➥ Observer

The observer pattern defines a one-to-many relationship dependency between classes, so when one object changes state, all it's dependents are notified and updated automatically.

This pattern is used by the image data module to tell the various visualizations and image processing tools to update themselves when the data has changed. Figure 2.2 demonstrates how three observers are notified when a 3-dimensional image is updated.

➥ Mediator

Mediator encapsulates how a set of objects interact. It promotes loose coupling of objects, by keeping them from referring to each other directly.

Mediator is used for all communications between the various GUI components[8]. Also the message-passing between the visualizations and image processing tools employ this pattern.

- Factory

  A factory provides an interface for creating families of related objects without specifying their concrete classes.

  Dr. Jekyll uses factories to allow for different image processing tools to “register” themselves with the factory at runtime. The kernel then generates a list of the available tools by querying the factory. The user may then instantiate any of the available plugins from the user interface.

  The nice thing about using a factory, is that all code, including the “registering” (telling the program that the plugin is available) of the plugin can be done in the source files were the plugin belongs. When implementing new plugins, no other files need to be changed, making it easier for a third party to extend the program.

- Singleton

  The singleton pattern ensures that a class has only one instance, and provides global access to it.

  The factory responsible for generating the image processing tools is realized as a singleton.

- Iterator

  Iterator hides the implementation of aggregate/container objects from their implementation. This allows for altering underlying data structures without altering code that refers to the data structure. E.g. a linked list could be converted to a binary tree without altering any other code. Figure 2.3 illustrates how the iterator pattern can be realized while using the STL.

  Most containers used in *Dr. Jekyll* are designed to be traversed by iterators. However iterators rely upon the notion that the data are organized in a strict ordering, something which does not make sense for 2-dimensional and 3-dimensional images.

  All STL-containers are designed to be traversed by iterators. But so is also the *Structural elements* class (see section 3.4.3, which is provided by *Dr. Jekyll* and used by several of the algorithms.

- Chain of responsibility Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain

```
int[array_size] A;
// Fill A.
for (int i = 0; i < array_size; ++i) {
  do_something(A[i]);
}
```

```
typedef std::vector<int> mycont;
mycont A;
// Fill A.
for (mycont::iterator i = A.begin();
      i != A.end(); ++i) {
  do_something(*i); // *i gets what i points to.
}
```

Figure 2.3: These two code examples illustrates the difference between classical "C-style" array traversal (left), and modern "Iterator-based" traversal (right). Notice how the loop in the iterator based version is not tied to the size of `A`. Also note how access to the data-element is completely independent of the container type. Changing the container type of `A` is as easy as altering the typedef of `mycont`, not other code should be affected. However the syntax of the iterator based traversal is more obscure and requires the programmer to know about the iterator pattern.

the receiving object and pass the request along the chain until an object handles it.

The Qt GUI library which is used throughout *Dr. Jekyll* (see section 2.2.2) uses this pattern for communication between the widgets.

For instance the widget which displays the visualization of an image may pick up a mouseclick, but it do not know how to handle it. It then emits a signal, which is caught and remitted several times until finally a plugin which was waiting for a pixel position gets it and is able to do an action based upon the mouseclick.

One notable exception to using patterns is the implementation of the "undo" functionality. The *undo-pattern* (described as part of the *command* pattern in Gamma [15]) prescribes that each plugin shall be able to rollback the changes it makes to a dataset. Since many algorithms completely changes the shape and connectivity of a dataset, and are algorithmically non-reversible, the only option is for the plugins is to store the complete dataset. In order to avoid duplicating this functionality, the undo functionality was implemented as a persistent plugin attached to each dataset. The undo plugin stores a complete copy of the entire dataset each time the data is changed.

2.2 Language and libraries

2.2.1 C++ and the Standard Template Library (STL)

The programming language of choice in *Dr. Jekyll* is C++. It is a compiled, type-safe, cross-platform, standardized language originally designed by Bjarne Stroustrup [49]. C++ became an official ISO standard in 1998 [23] and it is supported on nearly every computer platform. This ensures cross-platform portability.

C++ has some distinct key features, in contrast to other languages, that made it compelling to use it in *Dr. Jekyll*. C++ compiles to the target platform, and it supports templates. The following is a discussion of how these two features sets C++ apart from is two major competitors, namely Sun's Java and Microsoft's C# (C-sharp)[13].

1. Compiles to target platform

   Both Java and C# compiles to a virtual machine, ensuring that the same binary will run on multiple hardware platforms, as long as these platforms have a suitable virtual machine. In theory, applications running on a virtual machine can be as fast as an application compiled directly to native machine code (in the end everything ends up as assembler code). But, due to their runtime environments, there is usually a speed tradeoff. Depending on the application, native C++ is from a few percent to several orders of magnitude faster than Java. Pure C is a compiled language, but it lacks C++'s object oriented abstraction, type safety and template library which *Dr. Jekyll* relies upon.

2. Templates

   A unique feature of C++ compared to Java and C# is the support for templates. (Though a similar mechanism is slated for the new Java 1.5 standard (called generics) [69], and C# may also get templates in a new revision.) Templates are an abstraction where the *type* of the parameters to a method/function is determined at compile-time (and not at code-time). In short, templates are a form of automated code generation. They are particularly well suited for containers, where they allow complex data structures to be modeled in a generic way, without knowing what kind of objects they will hold. Template based containers allows a data structure to be *large enough, but not larger* than necessary. Typical datasets for *Dr. Jekyll* are quite large, and reducing memory usage is important to increase speed.

In addition to the features mentioned above, C++ also has some other strengths that makes it a good programming languages for a project like *Dr. Jekyll*.

A large part of the C++ standard details the STL (Standard Template Library). The STL includes common data structures (called containers) and algorithms from computer science in a highly optimized, generic framework. Much of the STL is based on the use of the iterator pattern, a way to access the underlying elements of a container, without knowing details of its implementation.

The STL is documented in [24] and [36]. Using the STL and the iterator pattern it is very easy to experiment with different datastructures. Figure 2.3 shows how this can be realized. Programming with the STL typically produces much more robust and compact code than implementing everything from scratch. Both Java and C# include comparable standard libraries.

Since C++ has been in widespread use for many years, almost every operating system has several working C++ compilers. Traditionally, there have been small, but subtle, differences between compilers from different vendors., making cross-platform C++ development troublesome. With the arrival of the C++ standard in 1998, this has greatly improved, and *Dr. Jekyll* has been compiled on both Linux and SGI IRIX. (Licensing, not technical issues, makes it troublesome to compile on Windows.)

Because of its widespread use and compatibility with C, there is a lot of high quality libraries available. There is also a wide variety of mature compilers, debuggers and tools available from numerous vendors.

While C++ has a lot of advantages, programming *Dr. Jekyll* has also exposed many of the disadvantages of C++. The syntax is often highly complex, especially when using templates, and the language contains many subtleties. C++ gives complete control to the programmer, and trusts the programmer to know what he is doing, including leaking memory. When using templates the error messages produced by the compiler can get extremely long and hard to understand. A further discussion of these issues are delegated to section 5.3.1.

Several books were essential to gain deeper understanding of C++, both for elegance and efficiency, these books were [53, 52, 34, 35].

Alternatives

Java and C# has been debated in the preceding section, and they have many of the same strengths and weaknesses, with C++ probably being the fastest of them. Also, C++ is not tied to a particular vendor.

A totally different approach would be to implement *Dr. Jekyll* in a scripting language like Perl or Python. Scripting languages are from 3-10 times slower than a compiled language. However the development time is often 3-10 times less as well. [42]

Scripting languages are also not as strictly typed as C++, the type of a variable is deduced at run time. This technique is called *dynamic typing.* This can be both a blessing and a curse since there is no “safety net” (in form of a compiler) when you try to assign incompatible types to each other, however strict type checking requires effort on behalf of the programmer to provide type information. Furthermore dynamic typing allows for conversion between types (like integers to strings) without the need for explicitly calling conversion functions. On the other hand strict typing usually results in a faster program since no type-checking takes place at runtime.

There has been very few empirical studies on the cost and benefits of type strict type checking. It is one of these ideas that more or less are taken for granted, however one study by Prechlet et al[40] concludes that strict type checking (in the form of compiler messages) increase programmer productivity and reduces bugs.

An emerging approach is however to combine the best of both worlds, writing C++, C or Fortran code for the speed-critical parts, and using a scripting language to “glue” together the various parts[28]. In hindsight is it not obvious that it would not have been better to write *Dr. Jekyll* using this paradigm.

However high-quality third party libraries are not as widely available as they are for C++, and they require “bindings” which are written by a fourth party as well, often of various quality and lacking documentation.

While calling traditional C, C++ or Fortran functions from scripting languages is quite easy, it is not obvious how to call template parametrized C++ functions. This is because the parameter and return types of a template function is first known at compile time, when the compiler deduces all possible types from the program. Thus the functions can not (today) be stored in a library like traditional functions.

2.2.2 GUI and message passing: Qt

One of the major obstacles in writing cross-platform applications has been the portability of the graphical user interface. Traditionally the GUI has been tightly coupled to the operating system. The Norwegian company Trolltech [81] produces a platform-independent GUI toolkit named Qt [77]. It is a commercial product, but since *Dr. Jekyll* is licensed under the GPL[68] (see section 2.3) it was possible to obtain a freely available version of the library at no cost.

In addition to providing widgets of all kinds, like toolbars and sliders, Qt provides an inter-object communication facility called *signals and slots*. Signals and slots allow for truly anonymous object communication. An object is designed with signals it can emit, these signals are connected to slots in other objects, but neither the object emitting the signal, nor the receiving object need to know about each other. Of course some object in the middle must know of both objects and set up the communication, a variant of the mediator pattern[8].

In *Dr. Jekyll* this mechanism is used for communication between most of the modules depicted in figure 2.1. The exception to this is access to the data. There is a small overhead introduced with each signal, and it would be prohibitively expensive to use it for access to the actual image data. Image data are therefore accessed directly by passing a handle to the image data container.

In addition to being a GUI library, Qt comes with predefined classes for mutexes and semaphores. These are used for locking of the image data, to avoid multiple plugins trying to modify the data at the same time.

While developing *Dr. Jekyll*, one of Qt's limitations has been a constant source of irritation. The signals and slots mechanism (which introduces two new keywords to C++) relies upon the use of a preprocessor to integrate these keywords into the language. This preprocessor, called the *Moc (Meta Object Compiler)*, does only process a subset of the C++ language, it can not process classes which use templates [58]. Fortunately it is almost always possible to work around this limitation by letting the template parts of a class be a subclass of a non-template class which can be preprocessed. The non-template superclass can then provide virtual functions which the template subclass can redeclare. However, this leads to unnatural class hierarchies, and forces a class to be split among multiple source files, when they conceptually are one class.

Alternatives

There are a few other cross platform GUI libraries available which can be used to write a cross-platform GUI-application. Java provides actually two different GUI libraries; AWT (The Abstract Windowing Toolkit) and Swing. While both of these are functional, they are notoriously slow and one is tied to using Java as a programming language.

One of the goals of Microsoft's ".NET initiative" is a promising approach for standardizing a cross-platform and cross-language development framework. It is still too early to tell if the promises come true, but if they do, it would be a very attractive alternative. It also seems that all .NET applications must be run on a virtual machine, and thus will often be slower than a native application.

An interesting alternative is WxWindows[83], which is a compatibility layer on top of a platform's native API. This is both a blessing and a curse, since it effectively reduces the API to the least common denominator of the native libraries. Originally started at the University of Edinburgh, it is today being developed and maintained by volunteers.

For the signals and slots mechanism there exists other alternatives as well. Libsig++[74] and Boost.Signals[72] both provide template based signals and slots mechanisms under a freely available license. Both of these avoid the use of preprocessors which Qt uses. However, they both rely on features of C++ which are in the standard, but which not all compilers support yet. Therefore, since it nevertheless was necessary to use Qt's signals and slots mechanism for the GUI, it was also chosen to use them for communications between different parts of the application (see however section 5.3.1 for some thought on this).

2.2.3 Portable graphics: OpenGL and Open Inventor

The only standardized cross-platform library for high performance graphics is OpenGL. A descendant of SGI's IRIS GL, it was originally released to the public in 1992, and quickly became the dominant market standard. The OpenGL standard is controlled by the OpenGL Architecture Revival Board (OpenGL ARB), the standard is detailed in [66] and [25].

A unique feature of OpenGL is that is allows for hardware acceleration of both 2D and 3D graphics when available. Traditionally the domain of high-end (and high cost) workstations, dedicated graphics processors are today commonplace on PC's thanks to the gaming industry. Exploiting such accelerators boost graphics performance tremendously and make 3D visualizations possible on a standard desktop computer.

OpenGL operates at very low level, working with single pixels and vertexes. Open Inventor is a library built on top of OpenGL to allow for a higher abstraction level[63]. It is based on a *hierarchical object model*; graphical objects are stored in a scene graph, and operations can be applied to just one branch of the graph. These operations can include texturing, animation, scaling etc. The underlying library executes the necessary OpenGL calls.

Open Inventor was originally developed by SGI, but they have now ceased development. While not a formally defined standard, the last released version is throughly documented and two companies sell source level compatible versions of Open Inventor. These companies are the French TGS [80] and the Norwegian Systems In Motion [79]. The version released by Systems In Motion is called Coin[73] and is freely available under the GPL license (see section 2.3), it was used for developing *Dr. Jekyll*. Systems in Motion also provides bindings to Qt via the SoQt library (also under the GPL).

At the time of this writing Systems In Motion was developing volume rendering extensions for Coin, called SimVoleon. These extensions were made available to us, but as of yet they have not been released to the public, and the license has yet to be determined.

Alternatives

Using OpenGL and Open Inventor is a typical “scientific programming” approach to computer graphics. The gaming industry, which is not overly concerned with cross-platform portability, primarily uses Microsoft’s own DirectX for 3D graphics[76]. Because of the lack of cross-platform tools, it was not evaluated for use in *Dr. Jekyll*.

A very popular library for scientific visualization is a library called VTK (The Visualization Toolkit). As the name implies, it is more tilted towards doing visualizations of datasets than for interactively manipulating them. VTK is described in [44] and is released under an open source license. It is available from [82]. Open Inventor was chosen over VTK because the former provides better integration with Qt than VTK. However a possible extension of *Dr. Jekyll* is to integrate VTK as well. (See section 3.6.3 for a discussion of this.)

SGI has more or less abandoned Open Inventor as their high-performance API for interactive graphics, and they are focusing on a technology called OpenGL Performer (previously IRIS Performer). Performer is commercially available on SGI, Linux and Windows workstations, however no free version is available. While Performer and Open Inventor can be extended into each others domain, they each have their own niche. Inventor is made for easy-of-programming and integration with the user interface, while Performer is designed to extract the highest level of performance. Specially from high-end, dedicated hardware from SGI.

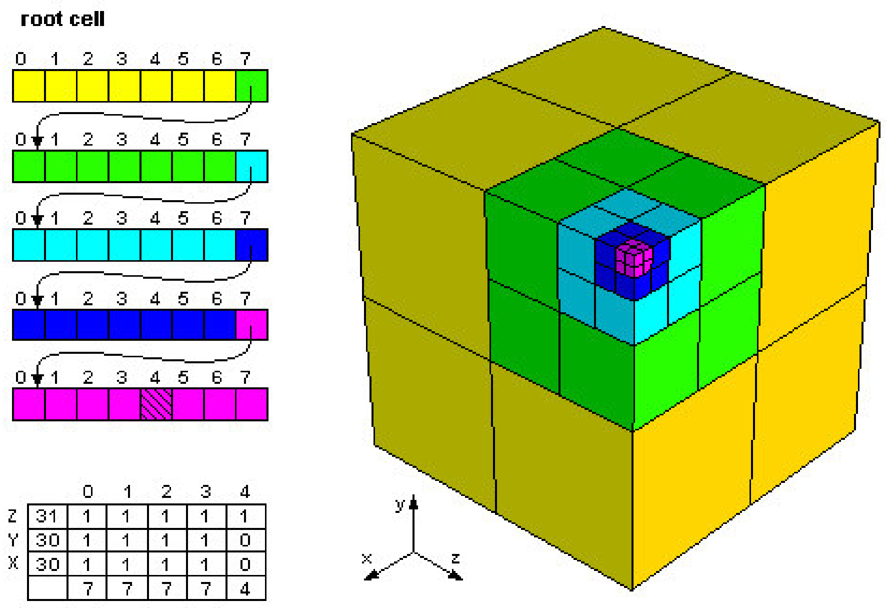
A short overview of Performer is given in an SGI white paper[46], the programmers manual is also available online[12].

2.2.4 Data storage: Blitz++

Efficient storage is perhaps the most crucial step in an application. For applications like *Dr. Jekyll* which deals with potentially large amounts of data, it is paramount to be able to efficiently lookup, alter and copy data,

For discrete image data there is basically two different approaches to storing the data:

1. The first is the “traditional” method where every pixel is stored in memory at a unique position, mapping directly to the pixels position in the image. The amount of memory used by this method is $\mathcal{O}(n)$ where n is the number of pixels in the image. However, the lookup of a pixel value when the



		0	1	2	3	4
Z	31	1	1	1	1	1
Y	30	1	1	1	1	0
X	30	1	1	1	1	0
		7	7	7	7	4

Figure 2.4: OCtree example. The cube is subdivided into 8 subcubes. One of the subcubes is subdivided further.

position is known is an $\mathcal{O}(1)$ operation. For such containers, accessing a unique pixel position is an $\mathcal{O}(1)$ operation.

2. Another way of organizing voxel data is to use an OCTree. In an OCTree the image space is subdivided into octants. Each octant may be full, partially full or empty. A partially full octant is then subdivided into suboctants until they are either full or empty. Figure 2.4 illustrates spatial subdivision. (For a 2-dimensional image the analog of an OCTree is a quadtree. The planar image is subdivided into quadrants.)

   For segmented data OCtrees can conserve memory usage. Calculations involving tree traversals are involved when looking up a single voxels's value, and altering a voxel's value can be an expensive operation since it can force recalculation of large parts of the tree. Visualization of an OCTree also requires specialized algorithms.

While a segmented image can benefit from OCTrees, raw data images with many different pixel values grouped closely together can not. In order to be as flexible as possible, at the cost of memory usage, the “traditional” approach was chosen. Even such a well proven technique as storing every pixel in memory has many challenges. Preferably it should be memory efficient, as well as providing possibilities for generating subvolumes and slices.

The Blitz++ library [71], written by Todd Veldhuizen, is a C++ library which tries to match the speed of Fortran for calculations, while preserving the high level abstractions of C++. Recent benchmarks show that it has succeeded [59]. Blitz++ allows for n-dimensional, resizeable matrices. These matrices are template based and can hold data of any type. It is also possible to generate slices and submatrices with a convenient syntax.

Alternatives

There are many available libraries for doing linear algebra, and they do include their own containers for multidimensional data. For generic multidimensional containers the selection is not as broad. However one package deserves mention, namely Boost.MultiArray.

As a generic multidimensional storage container, Boost.MultiArray[16] holds great promise. Boost.MultiArray was inspired by Blitz++ in the design, and the author of Blitz++, Todd Veldhuizen provided comments on the design and implementation. Boost.MultiArray is specially interesting, since it could be proposed as part of a future revision of the C++ standard. It was however not available when the *Dr. Jekyll* project was started. While changing from Blitz++ to Boost.MultiArray would require the updating of a lot of files, it is not an impossible task. Before Boost.Multiarray is standarized however, there are no sound reasons to do so, since they technically provide the same functionality. It should also be noted that Boost.MultiArray was not published at the start of the *Dr. Jekyll* project.

2.2.5 Image data I/O: Magick++

The last library which *Dr. Jekyll* depends upon is Magick++ [75]. Typical data sources will generate input data as slices of 2D images. There are many different file formats for storing images, and Magick++ supports the import of most of them through a common C++ interface.

Alternatives

Qt itself includes much of the same functionality as Magick++ through a class called QImage. Obviously it would be better to avoid throwing yet another lib-

STL	With C++ compiler
Qt	GPL or Qt Professional License 1 (GPL version is UNIX only.)
Coin (Open Inventor)	GPL or Coin Professional License (version 1 is LGPL.)
SoQt	GPL or Coin Professional License
SimVoleon	Not yet publicly released, license undecided
Blitz++	GPL or Blitz Artistic
Magick++	ImageMagick license
OpenGL	With OS/Graphics Hardware (The freely available Mesa implementation is released under the MIT license.)

Table 2.1: Summary of the licenses for the various libraries used in *Dr. Jekyll*

rary in the mix. QImage however lacks one important feature which was deemed necessary for the application, namely the support for 16-bit grey scale images.

It could be argued that neither the human eye, nor most computer screens can distinguish between more than 256 levels of grey for a given lightning condition. So the use of if 16-bit grey scale is rather limited when the image is to be shown directly on a computer screen. However photographic equipment like MR and CT scanners do produce gray scale images with a larger bitdepth than 8-bit. Down-scaling the images to 8-bits (as QImage do) in effect removes information from the image, and algorithms on the images can use this information.

Another option instead of Magick++ is the NetPBM utilities. They are primarily a collection of command line utilities for image processing (and supporting hundreds of different image formats) it also has a library for calling the functions from within an application. This library is only a C library and lacks some of the features of Magick++.

2.3 Licensing

There is a vast collection of libraries available on the Internet for free download. While downloading the library might be free, the usage terms of each library might be different and for an application like *Dr. Jekyll* it is quite complicated

All of the libraries *Dr. Jekyll* depends upon are released as *Open Source*. The definition of Open Source is given in [55], in short it allows for free modification, use and redistribution of the libraries. However there exists many different Open Source licenses, and the exact nature of what one is allowed to do or not to do varies between them.

Since *Dr. Jekyll* is based on a large number of “freely” available libraries, it was necessary to check the license for all of them to see if they could be used together. As table 2.1 summaries there are many different licenses to take into account, and each license has its own subtleties. Detailing the various differences are beyond the scope of this document, they can all be found at the Open Source Initiatives website[56].

Because the version of Coin and SoQt used are released under the GPL, it dictates that *Dr. Jekyll* has to be released under the GPL as well. This is not entirely unproblematic, since *Dr. Jekyll* can use the (as of this writing unreleased) SimVoleon library for volume rendering. In order to allow for this, we (the copyright holders) must extend the license of *Dr. Jekyll* to allow for this linking. This can not be done without the permission of Systems in Motion however.

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

It was therefore chosen to release *Dr. Jekyll* under the GPL license, making the source code freely available to the rest of the world. This had added side effect the we could host the project on Sourceforge [78], a website providing open source projects access to homepages, version control, bug and feature trackers, mailing lists etc.

2.4 Concluding remarks

The above discussion has summarized choices made during fall 2001, chapter 5.3.2 discusses how the choices worked in practice for developing *Dr. Jekyll*. In hindsight many of the choices still seem sound, both for the overall design, and for the technical merits of the language and libraries chosen.

The use of design patterns is helpful in describing the implementation of the algorithm to others, but it is also valuable as a tool for making the right abstractions.

When choosing libraries, not only the technical merits of each library must be considered, but also the licensing issues pertaining to each of them plays a role. The libraries chosen dictates the final licensing of the entire *Dr. Jekyll* application.

All of the libraries chosen has their specific purpose, and with the exception of Qt, which is used as the “glue” which holds the application together, most of the libraries are used only in a few classes. Changing anyone of the other libraries, should therefore not require extensive reengineering of large parts of the application.

Chapter 3

Algorithms for postprocessing

This chapter begins with a summary of some of the issues which arises when implementing algorithms for large datasets. Thereafter the algorithms used for postprocessing in *Dr. Jekyll* are discussed. Emphasis it given to detailing how the algorithms are *implemented* since this has been a main part of the work, but also the theoretical background is included. The chapter ends with a discussion on how to visualize volumetric images, when interactive feedback is paramount.

3.1 A note about image sizes, bitdepths and caching

Since the main goal of the application is to be an interactive, the running time of algorithms are crucial. Efficient algorithms are arguably the most important factor contributing to the running, but as image sizes increase, using memory efficiently greatly affects the running time as well. This section serves as a short introduction to issues regarding large image sizes.

As a basis for discussion consider figure 3.1. It contains a plot of the raw storage requirements of a 3-dimensional image with equal length in each dimension. As can be seen from the function storage requirements grows exponentially as the image size increases, but also note that the number of bits used for each pixel plays a significant role when determining the storage requirements. (Naturally using 8 bits pr. pixel has half the storage requirements of 16 bits, which again have half the storage requirements of 32 bits.)

A related problem to the memory requirement is that the maximum amount of addressable memory is limited to 4GB on 32-bit computers. For practical reasons, operating systems like MS Windows and Linux running on processors which implements the Intel IA-32 instruction set only has 2GB of data available to each

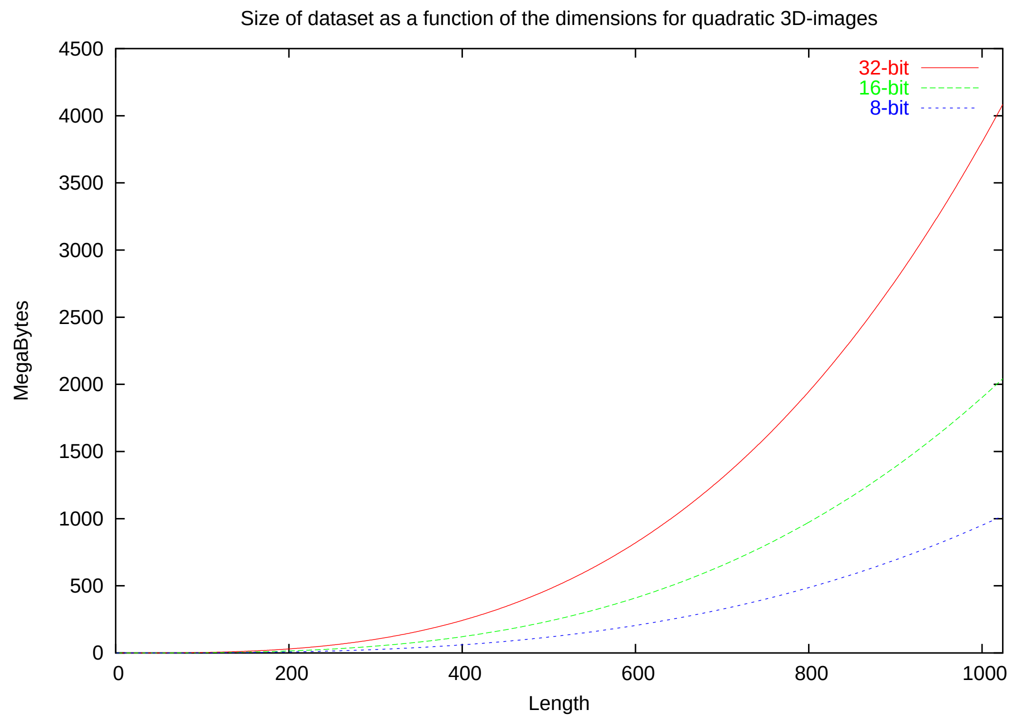


Figure 3.1: Plots of the function $f(l)_b = l^3 \cdot b,\ l \in (0, 1024), b = (8, 16, 32)$. The function show how the storage requirements increase exponentially with the image size.

process. Limiting the possible image size even further.[1] (The IA-32 instruction set is being used by the Intel Pentiums (I-IV) and clones. For an overview of the IA-32 architecture and memory management see the *IA-32 Software Development Manual* published by Intel Corp.[21].)

Since datasets can be (much) larger than the physical amount and addressability of the processor, many techniques exists for overcoming these limitations. The survey by Vitter[60] gives a good overview of such methods. Due to their complexity *Dr. Jekyll* does not use such techniques, and assumes a flat address space where the whole image can be kept in memory at all times. The maximum size of datasets which *Dr. Jekyll* can process is thus limited by the addressability of the processor and operation system. Thanks to the heavy use of templates, *Dr. Jekyll* should however scale to future 64-bits processors (and beyond) just by recompiling the program. (This does not however solve the problem of the data being swapped to disk, only more memory on the computer can remedy this situation.)

Reducing the storage requirements does not only allow for larger images sizes, it

[1]To complicate matters even further, many 32-bit computers (including *some* IA-32 processors), allow for 36-bit of addressing, raising the addressable limitation to 64GB – however the per process limitation still exists.

can also reduce the running time of an algorithm in the following two ways:

1. If a dataset is larger than the amount of physical memory in the computer, some of the data has to be swapped to disk. Reading and writing of swapped data is many orders of magnitude slower than accessing the data if they were in main memory.

   By using as few bits as possible for each pixel, the memory requirements are reduced accordingly.

2. Another aspect where the number of bits used for representing each pixel affects the datatype is with respect to the cache hierarchy.

   A caching hierarchy is implemented because the speed of a modern processor is roughly an order of magnitude faster than the speed of the main memory. Without caching the processor must wait a long time every time it reads from memory. With caching, used data are stored in the much faster cache-memory. However, not only data already read are cached, most caches implements a “read-ahead” (also called prefetching) strategy where data “close” to the accessed data are also stored into cache memory. The details of what is “close” varies with the processor model. During image traversals it is important to try to access memory such that memory already in the cache is accessed.

   Since the size of the cache is fixed (typically at 128Kb or 512Kb for the second level cache on PC class hardware), a small type for pixel data, allow for more pixels in the cache, and thus faster running time.

   For a discussion on the widening gap between processor speed and memory speed, see [67], details of various caching strategies (and computer architecture in general) is given in [20].

For effective implementations, a vital question is how to use the memory efficiently, both with regards to memory limitations and caching.

The template mechanism of C++ (see section 1.4.1) is a partial solution to limiting image sizes. The bitdepth of both input and output images can be deduced at compile time, and can be as small as necessary.

Another option to reduce the memory usage of images is to keep the images compressed in memory, and run algorithms on the compressed image. Bajaj et al[4] demonstrates such a technique for 3-dimensional RGB images by using wavelets. For segmented images even simple run-length-encoding is probably effective, since many pixels share the same value as their neighbor. See section 5.4.3 for some more notes about this.

To utilize the processor caches it is paramount to exploit the "read-ahead" strategy. It should be known that a multidimensional image can be stored as a continuous vector in memory, and a number of helper variables called strides are used to map from image-positions (like $\mathcal{I}(x, y, z)$) to memory positions. By traversing images consecutively (in the direction where the stride is one, the next data element is usually already stored in the cache. In *Dr. Jekyll* the image containers (which are based on the Blitz++ library) are stored depth-column-row, and all array traversals are made to utilize this. Technically, this is ensured by using C++'s macro facility to generate the loops, instead of handwriting them each time an image is traversed.

Much research is currently put into understanding how to better utilize caches. The TUNE project at University of North Carolina[37] tries to get a better theoretical understanding on how caching affects the formal running time of algorithms [45]. The ATLAS Project[70] aims to write a linear-algebra library which automatically tunes its algorithms and storage containers for a specific CPU. This tuning is done by executing empirical tests on the actual CPU before the library is built. The techniques applied by the ATLAS project is explained in [65].

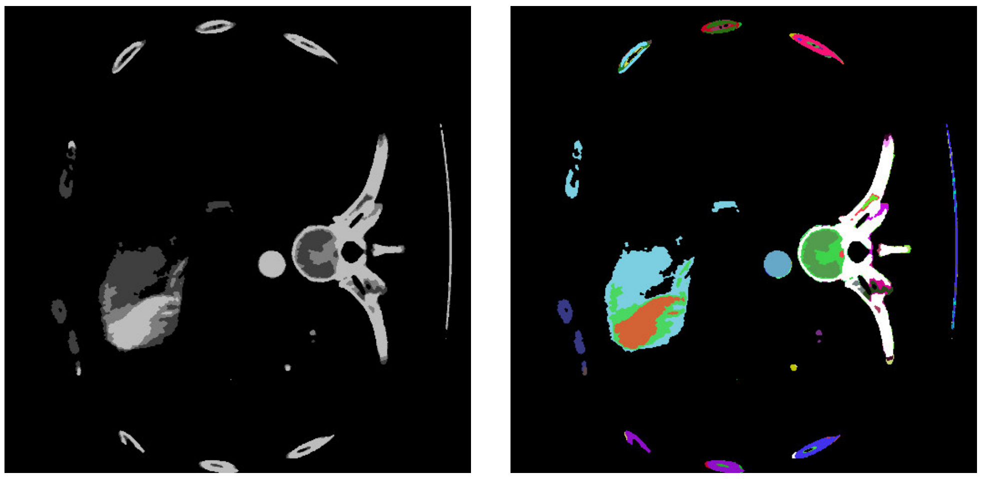
3.2 A note a about segmented versus binary images

When implementing algorithms for segmented images it is not obvious if they shall be seen as gray scale images or as a stack of binary images.

Usually gray scale images can be seen as a sampling of a continuous field, where the pixel value designates the luminosity of a pixel. Pixel values which are close to each other (for example expressed in L1-norm $|p_i - p_j|$) designates pixel which have almost the same luminosity. Thus pixels with a small difference in value have more in common than two pixels with a larger gap in value.

This does not hold true for a segmented image. Only if the difference between two pixel values are zero do they have anything more in common than two pixel values with a large distance between them.

Algorithms are typically defined differently for gray scale and binary images. The binary version of an algorithm is typically the one most suited for post-processing, since it already makes a clear distinction between two labels, namely foreground and background. However for our purposes, the algorithms must be extended to handle more than two distinct labels, with the traditional background label being just one of several other labels, and not a special case of its own. The details of these extensions are summarized in the following text.



(a) Segmented image (b) Components of segmented image

Figure 3.2: An segmented CT slice and its connected components. Notice that the connected components are calculated in 3D using 26-connectivity so connections between some of the components are via other slices than the one shown here. (Images courtsey of IVS, RH.)

3.3 Connected components analysis

An important concept, both for relabeling, analysis and visualization are the *connected-components* of the image. Pixels sharing the same value, and constituting a path in the image are said to belong a unique *component*. Figure 3.2 illustrates the components of a CT slice calculated using the presented algorithm.

The connected components of an image can be seen as the dual to the graph problem of finding all nodes which there is a path between.

3.3.1 Theoretical background and algorithm overview

The first known algorithm for calculating the connected-components of a binary image were originally proposed by Rosenfeld and Pfaltz in their 1966 paper *Sequential Operations on Digital Picture Processing* [39].

An optimized version was published in [57]. The latter version is the basis for the algorithm implemented in *Dr. Jekyll*. For *Dr. Jekyll* the algorithm was extended to handle segmented images as well as binary images by ensuring that the only probed neighbors of a pixel are those with the same label as the pixel in question.

The algorithm is clearly divided into three different steps, the first and third step are given formally as algorithm 1, while the second step is given as algorithm 2.

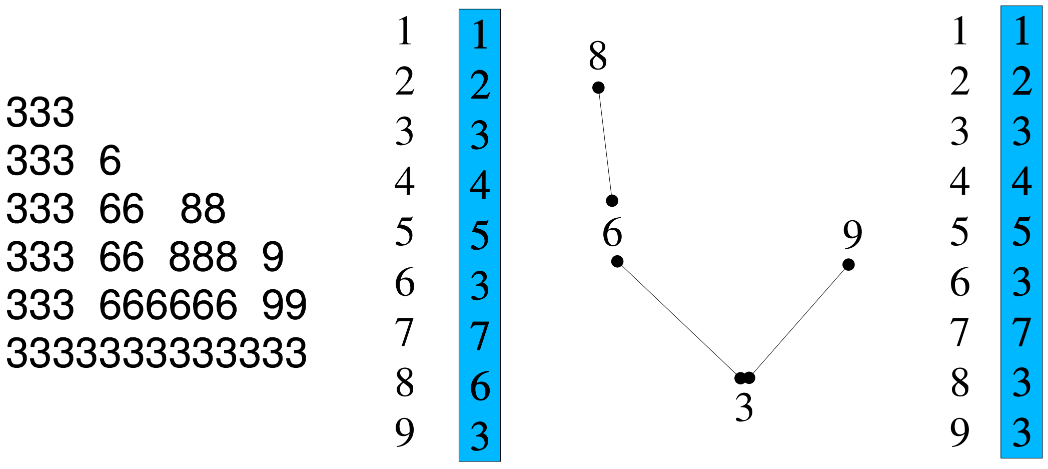


Figure 3.3: To the left is an example showing equivalence between different labels of a component. The next item is the resulting equivalence table, and the tree represented by by the table. To the right is the lookup table after the propagation step. Notice how the tree has been flattened, and value 8 in the table points directly to component 3. Only a part of the image is shown, containing one component. The other entries in the table is supposed to be found outside the image part to the left.

Sequential labeling algorithm

The algorithm is an image-to-image transform, taking as input an image, $\mathcal{I} = \{p_1, p_2, \ldots, p_N\}$ and generating as output a new image $\mathcal{I}' = \{p'_1, p'_2, \ldots, p'_N\}$. Both $\mathcal{I}$ and $\mathcal{I}'$ are used during processing.

Each pixel p is processed in order, and values of the neighbors in $\mathcal{I}$ are probed, those of the neighbors in $\mathcal{I}$ which have the same label as p are probed in the output image $\mathcal{I}'$ and their values are looked up in the translation table. The pixel p' is then assigned the minima of the values of the neighbors (with the same label) from the lookup table. Notice that there is no distinction between background and foreground pixels. It should be noted that it is only necessary to probe "backwards" in the image, only those pixels which already has a component contributes to the component of the current pixel. The first pixel in the image will therefore automatically be assigned a new component.

The lookup table is also updated, each of the neighbors that was found in $\mathcal{I}'$ is updated with the minima that was found, in this way regions which were thought to be different components but were later found to be different regions of the same component can merged in a second pass through the image.

If p has no neighbors with the same label, it is assigned a new region number i, and the new region number is stored in the translation table so that $\mathcal{T}[i] = i$.

Algorithm 1 Calculating connected components

```
Require: An input image of labels, $\mathcal{I} = \{p_1, p_2, \ldots, p_N\}$
Require: An output image of components, $\mathcal{I}' = \{p'_1, p'_2, \ldots, p'_N\}$
Require: A lookup table $\mathcal{T}$, initialized so that $\mathcal{T}[i] = i$
  for all $p \in \mathcal{I}$ do
    if $p \neq$ all neighbors in $\mathcal{I}$ then
      $p' \leftarrow$ new unique label
    else if $p$ = one or more neighbors in $\mathcal{I}$ then
      $m \leftarrow \infty$
      for all $n \in \mathcal{I}'$ {$n$ is neighbors with same label} do
        $m \leftarrow \min(m, \mathcal{T}[n])$
      end for
      $p' \leftarrow m$
      for all $n \in \mathcal{I}'$ {$n$ is neighbors with same label} do
        $\mathcal{T}[n] = m$
      end for
    end if
  end for
  propagate $\mathcal{T}$ to root-nodes.
  for all $p \in \mathcal{I}$ do
    $p' \leftarrow \mathcal{T}[p']$
  end for
```

The resulting translation table can be seen as a tree storing the equivalence relationship between regions which are found to be merged after they have been initially processed. An illustration of such a tree and its representation in the translation table can be seen in figure 3.3.

A special step, called the propagation step is made through the lookup table, and ensures that regions of the same component which originally were given different value can be merged. In other words, the tree represented by the translation table is flattened, and each node is assigned to it's root. The propagation step is given in algorithm 2.

Finally a second pass through the image is executed, during this pass a pixels final value is determined from the lookup table, based on the value the pixel was assigned during the first pass.

Propagation step

The goal of the propagation step is to merge all the different regions of the same

Algorithm 2 Propagation step of the connected components algorithm

```
Require: A lookup_table, $\mathcal{T}$
Require: A stack, $\mathcal{S}$
  for $i = \mathcal{T}_{\text{end}}$ downto $\mathcal{T}_{\text{begin}}$ do
    $c \leftarrow \mathcal{T}[i]$
    if $\mathcal{T}[c] \neq c$ then
      while $\mathcal{T}[c] \neq c$ do
        $\mathcal{S}$.push($c$)
        $c \leftarrow \mathcal{T}[c]$
      end while
      while $\neg\mathcal{S}$.empty do
        $\mathcal{T}[\mathcal{S}$.pop$] \leftarrow c$
      end while
      $\mathcal{T}[i] = c$
    end if
  end for
```

component. The regions were originally thought to be different components but were later found to be part of the same, this is reflected in the lookup table generated in the labeling step. Figure 3.3 demonstrates the contents of a lookup table before and after the propagation step. The output of the GUI dialog for creating the rule thus returns just on `BPFC` functor, which is an `Angroup` or an `Orgroup`. This group then includes all the other rules.

The merging is achieved by iterating over the lookup in reverse order, starting from the end. If the value at an index is different from the index value (it is then always lower than the index value), the value is used as index for a new lookup. This is repeated until the value at an index is the index itself. Ie. $\mathcal{T}[i] = i$, this is called a terminal node for the rest of this section.

For complex images the translation table can be several levels deep, like in figure 3.3. In order to avoid traversing the same path several times (like the 8-6-3 path in the figure), a stack is utilized to remember which nodes have been visited. When a node is found not to be a terminal node, it is pushed on the stack before traversing further down the tree. (Returning to figure 3.3 the resulting stack will be 8 when the terminal node is found.)

After a terminal node is found, the stack is popped and each entry popped is assigned to the current terminal node, ensuring the same path up the tree will only be traversed once.

```
template<class Label, class Component>
class Thurfjell {
public:
    \\ Constructors, destructor
    blitz :: Array<Component, 3>
    operator()( const blitz :: Array<Label, 3> input );
    void attachstructel ( Structural_element e );
}
```

Figure 3.4: Signature for the connected components algorithm. The `operator()` method returns an image of the type `Component`, but the input image is of the type `Label`. The algorithm is realized as a class (or more correctly, a functor), and not as a function, since it can later be queried for information about the connected-components image. This information includes sizes and boundingboxes of the components.

3.3.2 Implementation

In order to have a generic implementation of the algorithm, which was not tied to a particular bitdepth, and which can support multispectral images the algorithm was realized as a template class taking two template parameters. The first template parameter is the type of the dataelements in the input image, the second element it the type of the dataelements in the output image. This distinction between the input type and the output type was found to be necessary since the number of components in a large image can be several orders of magnitudes larger than the number of distinct labels in the input image.

As a worst case scenario for the return type, consider a volumetric image were every pixel is a separate component. The maximum number of components (and hence the resolution) of such an image is linear with the number of bits used for storage. For a cube shaped volumetric image with equal length in all dimensions, the resolution along each axis is given by the function $f(b) = \sqrt[3]{2^b}$, were b designates the bitdepth of the (unsigned) return type.

For a return type which is 32-bit wide this results in a maximum image size of 1625^3. While still a large image to handle interactively on PC-style hardware today, it is not completely out of the league for todays 32-bit computers.

Increasing the return type to 64-bits gives an image size of $(2.64 \cdot 10^6)^3$, while 128 bits yields $(6.98 \cdot 10^{12})^3$. 64-bit ought to be enough for any sequential algorithm, larger datasets would probably benefit from parallel algorithms running on a cluster of computers, and few computers will be equipped with anything close to this amount of memory, so disk swapping will be a real bottleneck.

It should be noted that the worst case scenario is extremely unlikely to arise in any real application *when the images presented to the algorithm are segmented!*

All interesting segmented images have at least some structure spanning multiple pixels, if not the segmentation algorithm has failed completely.

By decoupling the return type of the image from the implementation, the user can choose what return type is best suited for the problem at hand, while at the same time not introducing any runtime penalty.

There are multiple advantages to letting the input type of the algorithm be a template parameter as well. Primarily the algorithm is not tied to a particular bitdepth for input images, but the algorithm can also support multispectral image were the data elements are not a basic datatype (like an integer or a float), but can be any object implementing the equality (`operator==`) and assignment (`operator=`) operators.

3.3.3 Usage in Dr. Jekyll

The connected components of a segmented images constitutes view of the data at a lower level than the segmented image, since we are able to differentiate between different components which still share the same label. To be able to modify different components efficiently is the key to efficient postprocessing.

The image generated by this algorithm is an extremely versatile image which can be used for many different tasks during post-processing. Indeed it forms the basis of all further postprocessing which is based on a selection of components, and not acting globally on the whole image or directly manipulating pixels.

A summary of the uses of the algorithm within *Dr. Jekyll* is given below:

- Component selection

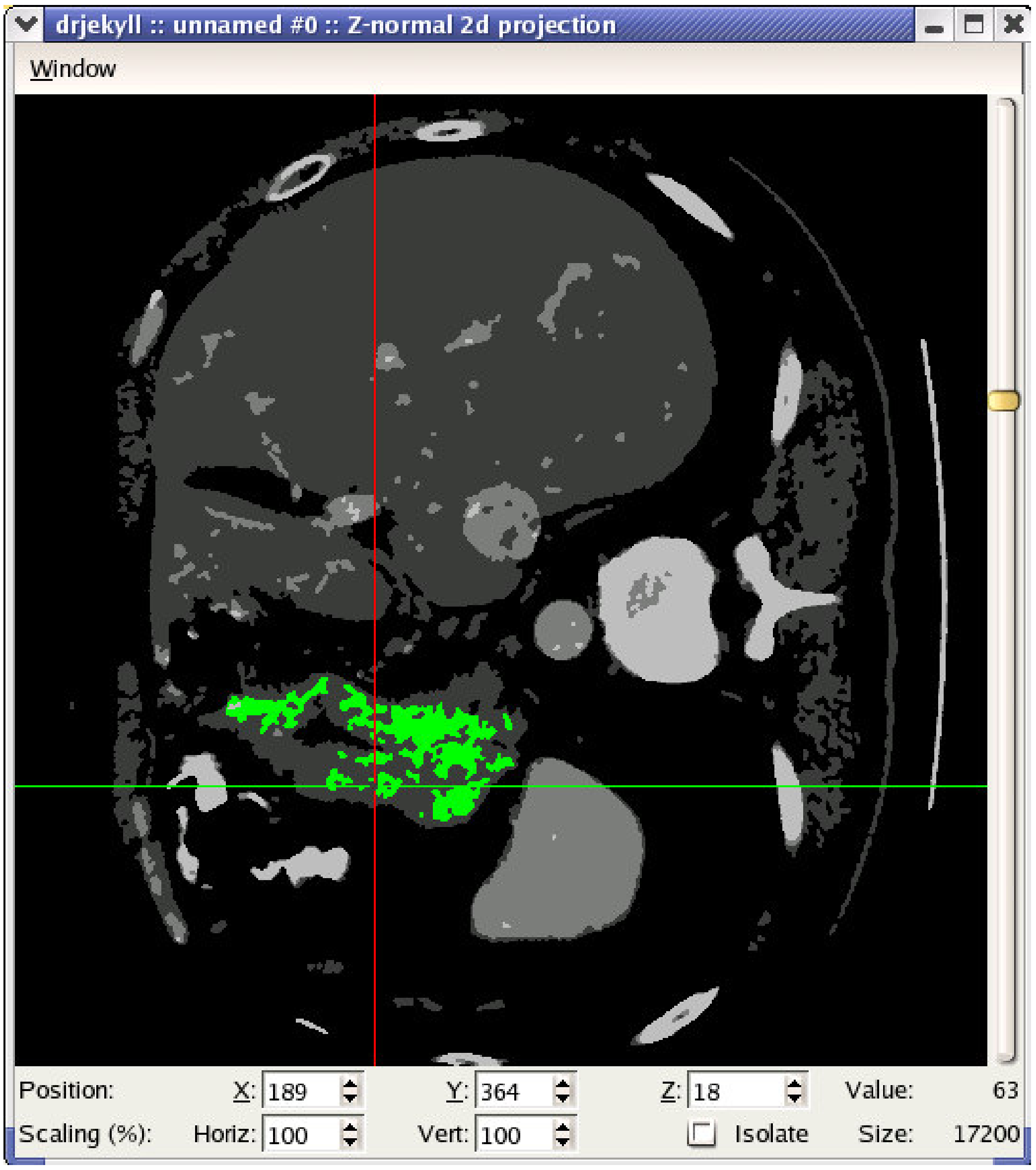
  For interactive postprocessing based on components, being able to select a number of components and apply algorithms to the selection. Figure 3.5 shows how the interface of *Dr. Jekyll* highlights a selection of components.

- Number of pixels and volumes

  The number of pixels in each component can be used further by other algorithms. The rule based relabeling algorithm is highly dependent on this information.

  Determining the volumes of components can also be the end product of the entire post-processing. For example, in medical analysis determining the volumes of organs are crucial both for diagnosis and surgical planning and it can be an end result itself of the entire post-processing process.

- Bounding boxes



Position: X: 189 Y: 364 Z: 18 Value: 63
Scaling (%): Horiz: 100 Vert: 100 Isolate Size: 17200

Figure 3.5: An illustration of how a selection of components is highlighted in *Dr. Jekyll*. Further operations can be applied to only the selection. The connected components algorithm described in this section was used for calculating the components.

While not an end in itself, algorithms which are to be applied to a selection of components can get a speed increase by knowing the bounding boxes of each of the component. The bounding boxes effectively reduces the image which must be search to find the selected components.

➥ Adjacency graph

During the calculation of the components it is possible to build an adjacency-graph for each component, storing what labels each component has a border to. This graph is the also used extensively by the rule based relabeling algorithm (section 3.5).

➥ Visualization

For 3-dimensional visualization, each component designates an unique object which has to be rendered. If surfaces are to be calculated for 3D visualization, each component designates a unique surface (section 3.6.2).

For 2-dimensional slices, highlighting the contours of a selected component allows for identifying the spatial structures of an image, since connections in other slices than the one present can be shown by highlighting the contours of the "active" component.

3.3.4 Alternative algorithms

Labeling algorithms for images have been an intense field of study since the original Rosenfeld and Pfaltz paper [39]. Much effort has been devoted to parallelizing the algorithm, an overview of such algorithms can be found in [19].

While dual and quad CPU machines are becoming more common, and technology like Intels HyperThreading (which lets one CPU appear as multiple CPU's) [32] are emerging, most desktop computers will continue to be single CPU-based for the foreseeable future. Such algorithms were therefore not evaluated for use in *Dr. Jekyll*.

The Achilles heel of the sequential algorithm (from [39]) is the updating of the lookup table which has to be done for every pixel. The solution presented above demonstrates one way to reduce the running time of the original algorithm, another solution based on a divide and conquer approach is presented by Park et al [38], however it works only for regularly shaped structuring elements, and it is presented only for 2D images, but the extension to 3D images is possible.

3.4 Mathematical morphology

Mathematical morphology can be defined as a theory for analyzing spatial structures. It is called morphology because it deals with the analysis of shapes and forms using a mathematical foundation. The origin of mathematical morphology comes from the study of the geometry of porous media in France in the mid-sixties, the classical text outlining the theory is *Elèments pour une thèorie des* *milieux poreux* by G. Matheron [33]. A modern introduction is given in *Morpological Image Analysis* by P. Soille[47].

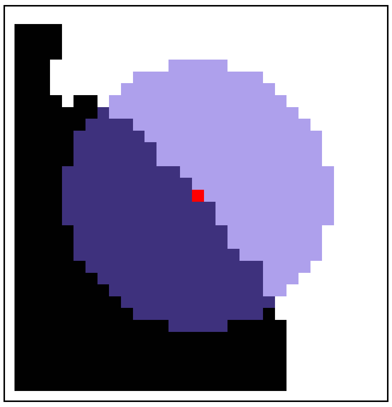


Figure 3.6: A structuring element both "hits" and "misses" the underlying image. The origin is the single red pixel in the center of the blue SE. The value of the origin pixel in the output image after being probed by the SE is dependent on the operator. (Note that this is a very large circular SE, made for illustration purposes.)

Mathematical morphology has become an important subfield of image analysis, and it is used in fields as diverse as geosciences, materials science and inspection systems, as well as in medical imaging.

The following is an concise introduction to morphological operators. The field of mathematical morphology is wide and exciting, and many interesting properties can be derived. This text gives only enough information for a discussion of the uses and implementation of the operators. Mathematical morphology is founded on classical set theory. For the following discussion a basic understanding of set theory is assumed.

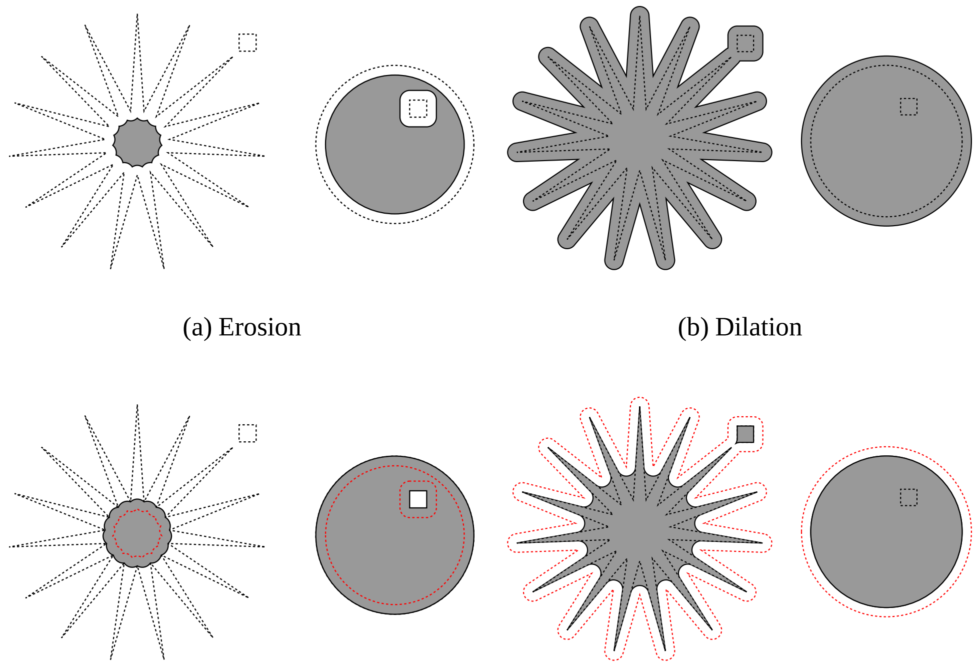
Morphological operators are operators that aim at extracting relevant structures of an image, considered through its subgraph representation. This is achieved by probing an image with another small image called the *structuring element* (hereafter called SE). Depending on the application, various SE's can be utilized. The SE is a small image, with a pixel defined as the origin of the element. The notion of an origin allows us to place the SE over a unique pixel in the input image. For each pixel in an image we can probe with the SE and assign a new pixel value to the same position in a new image. Figure 3.6 illustrates how a structuring element is placed on a single pixel, and allows for probing a unique subset of all the pixels in the image for every pixel. Morphological operators are thus image-to-image transforms.

3.4.1 Morphological operators on binary images

The following text summarizes the four basic morphological operators for a binary image. All the definitions are from *Morphological Image Analysis*[47]. Figure 3.7 illustrates the effect of the various morphological operators on the same input image, using the same SE.

➥ Erosion

When probing an image with a SE, a basic question is *does the structuring element fit the set?*. If the pixel, and all the other pixels which the SE "hits" are foreground, then the pixel is foreground in the output image, else it is



(a) Erosion (b) Dilation

(c) Opening (d) Closing

Figure 3.7: The effect of the various morphological on the same input image. The black stippled lines indicate the original image. In the bottom row, the red stipples indicate the input for the dilation and erosion step of opening and closing. Notice the image consist of both the star-shaped structure with a box on to the top-right and a circular structure with a box-shaped hole. Notice how the erosion (and thus the opening) preserves the hole in the circle, but removes the star shape. The dilation (and therefore the closing) on the other hand preserves the star shape, but removes the hole.

assigned background.

Definition 8 *Erosion*

$$\varepsilon_B(X) = \{x \mid B_x \subseteq X\}$$

Thus an erosion will make elements of the picture smaller, and increase the number of background pixels in an image.

➥ Dilation

Another basic question when probing is, *does the structural element hit the set?* If a pixel within the SE's "hits" (not necessarily at the origin), then the probed pixel will be assigned foreground in the new image.

Defi nition 9 *Dilation*

$$\delta_B(X) = \{x \mid B_x \cap X \neq \emptyset\}$$

A dilation will make subgraphs/elements thicker, the number of background pixels will decrease.

➥ Opening

The combination of an erosion followed by a dilation with the transposed SE. (The transposition of a set B corresponds to its symmetric set with respect to the origin: $\check{B} = \{-b \mid b \in B\}$ Symmetric SE's are their own transposed).

Defi nition 10 *Opening*

$$\gamma_B(X) = \delta_{\check{B}}[\varepsilon_B(X)]$$

The goal of morphological opening is to recover as much as possible of an eroded image. Small structures are removed, but large structures retain their size and their outline are smoothed.

Opening can be viewed as a "rolling-ball" moving around the inside of the shape, and filling any gap on the outside it touches.

➥ Closing

The dual of opening is closing. An image is first dilated and then eroded with the transposed SE. The end result are structures which keep their initial shape.

Defi nition 11 *Closing*

$$\phi_B(X) = \varepsilon_{\check{B}}[\delta_B(X)]$$

Closing can be thought of as a "rolling-ball" on the outside of the shape, and gap it fails to fill into is filled.

Notice that the definitions of morphological operators are dimensionless, so extending them to arbitrary spatial dimensions are trivial. However these definitions do not tell us how to apply them to gray scale and segmented images.

For gray scale images the normal case is to let an erosion assign to the pixel being probed the value of the minima of the image overlapped by the SE (see figure 3.6), for a dilation the maxima of the pixels within the SE are used. This approach works for gray scale images where the gray levels are continuous.

This approach does not however work for segmented images, where there is no ordering of the labels. However, if the value of foreground and background are label not hardcoded into the operators, but instead given as parameters the operator, the result is an operator which can be used in labeled images.

Another question which arises is what to do when part of the SE is outside of the definition domain of the picture? One possibility is to let the outside be defined as background, effectively adding a background border around the image. Another possibility, which is the one implemented for the morphological operators in *Dr. Jekyll* is to only consider the part of the SE which is inside the definition domain.

3.4.2 Algorithms for mathematical morphology

For an image on a discrete and regular grid, algorithm 3 and 4 summarizes how erosion and dilation can be implemented. Opening and closing can be be realized by letting the output of an erosion be the input of a dilation (and visa-versa).

The formal running time of a simple, image traversal based morphological operator is $\mathcal{O}(N(s-1))$ where N is the number of pixels in the image, and s is the number of pixels in the structural element[47].

Algorithm 3 Erosion

```
Require: An input image $\mathcal{I} = \{p_1, p_2, \ldots, p_N\}$
Require: An output image $\mathcal{I}' = \{p'_1, p'_2, \ldots, p'_N\}$, initially set to a copy of the
  input image.
Require: A Structural, $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$
Require: A function $f$ which calculates the position of a pixel in the input image
  based on an origin pixel and an delta point from the SE.
  for all $p \in \mathcal{I}$ do
    for all $s \in \mathcal{S}$ do
      if $f(p, s) \neq$ label then
        $p' =$ background
        break
      end if
    end for
  end for
```

If erosion and dilation is implemented as a sequential algorithm, like algorithm 3 and 4 suggests, it is not always necessary to probe all the pixels within the SE in order to determine the final value of the origin pixel. For erosion it is enough to

Algorithm 4 Dilation

```
Require: An input image $\mathcal{I} = \{p_1, p_2, \ldots, p_N\}$
Require: An output image $\mathcal{I}' = \{p'_1, p'_2, \ldots, p'_N\}$, initially set to a copy of the
   input image.
Require: A Structural, $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$
Require: A function $f$ which calculates the position of a pixel in the input image
   based on an origin pixel and an delta point from the SE.
   for all $p \in \mathcal{I}$ do
      for all $s \in \mathcal{S}$ do
         if $f(p, s) =$ label then
            $p' =$ label
            break
         end if
      end for
   end for
```

find one pixel where the SE “misses” the set. If it misses, the loop can be exited and the pixel set to the background label. We can then move on to processing the next pixel. Dilation allows for a dual technique. It is enough to find one pixel where the SE “hits” the set. Then we know that the pixel shall be set to the foreground label and can move on. Notice that if this strategy is to be used, the output image must initally be set to a copy of the input image.

Alternative algorithms

The formal running time of the morphological operators ($\mathcal{O}(N(s-1))$), can be reduced dramatically on some occasions. Unfortunately these techniques put constraints on either the dimension of the input image or on the shape of the structural element, and are therfore not as general as the algoritms above.

One possible technique is the “Moving histogram technique” where an histogram is updated for the pixels leaving and entering the structural element. This algorithm was presented in [11] and claims to have lower complexity and higher efficiency than comparable methods. However, thich approach is mainly effective when operating on gray scale images when using large SE. The algorithm is also only presented for 2-dimensional images, an extension to 3D should be possible however.

Experiments show that the classical loop based algorithms described above are fast enough for interactive usage, even on large volumes.

Algorithms3 and 4 is both simple to understand, and requires few lines of code to

```
template<typename VT>
 blitz ::Array<VT, 3>
 dilation ( const blitz ::Array<VT, 3>& old, const Structel& element,
            const VT& fromlabel, const VT& tolabel )
{
   blitz ::Array<VT, 3> img;
   Structel :: const_iterator  it ;
   // Add a border, initialize img to old etc.
   FOR_EACH(img) { // macro which brings x, y and z into scope.
      it = element.begin();
      while ( it != element.end() ) {
         if ( old( x+it->x(), y+it->y(), z+it->z() ) == fromlabel ) {
           img(x,y,z) = tolabel;
           break; }
         ++it;
      }
   }
   return img;
}


struct Dilation {
     template<typename VT>
      blitz ::Array<VT, 3>
     operator()( const blitz ::Array<VT, 3>& old, const Structel& element,
                 const VT& fromlabel, const VT& tolabel ) const {
          return dilation ( old, element, fromlabel, tolabel );
     }
}
```

Figure 3.8: The signature for the dilation operator, and it's corresponding functor object (notice the difference in casing). By use of the template parameter `VT` (for voxel type) the algorithm is completely independent of the type of data stored in the `Blitz++` containers. The functor object is necessary to be able to utilize the algorithm in the strategy pattern.

implement, thus less is the chance of an error in the implementation.

3.4.3 Implementation of morphological operators

In order to describe how the morphological operators were implemented, a quick summary of some if the goals of the *Dr. Jekyll* application is in order. The goal is to have both a useful interactive application, as well as to provide an extendible framework for new algorithms.

Such a framework consists of both working algorithms which can be called for other purposes than their original intention, as well as reusable components to implement new algorithms, or to create a graphical user interface for controlling them.

In order to be as generic as possible, all the morphological operators were implemented with the same function signature. Namely returning an new image, and taking as input the image which will be calculated on, and SE, as well as the label the operation will be going from and to. Figure 3.8 includes almost the entire body of the dilation operator which is used in *Dr. Jekyll*.

By returning an image object, it is possible to implement "chaining" of operators. This allows for easy implementation of opening and closing, as well as other morphological operators like the half-edge gradient. Returning the resulting image object directly, could introduce a lot of memory overhead, since all return values are passed by value in C++. The `Blitz++` object returned is however only an encapsulated pointer to the actual image data. The overhead of returning by value is thus negligible, thanks to this feature of the `Blitz++` library.

The Structural Element

In order to write generic algorithms which can support specialized SE's, the SE was separated from the algorithms and implemented as it's own class. The only thing that is stored in this class is an array containing point objects. These points store the offset from the origin of the structural element, and can be added to the current position to give a point within the SE.

The SE class is implemented as a subclass of the STL vector-container. The usage of the SE class is illustrated in figure 3.8; an iterator is created which iterates over the SE (independent of its shape and size) and these points are added to the current point and processed, to give the position of pixel to read.

This mechanism of storing delta points in a container is also used when calculationg a neigborhood for the connected components algorithm. This technique demonstrates how generalizing a concept as it's own class increases flexibility and encourages code reuse.

Graphical user interface

The morphological operators, combined with the connected components of an image allows for highly flexible interactive post-processing. However a good graphical user interface (GUI) to the algorithms is necessary to fully exploit this.

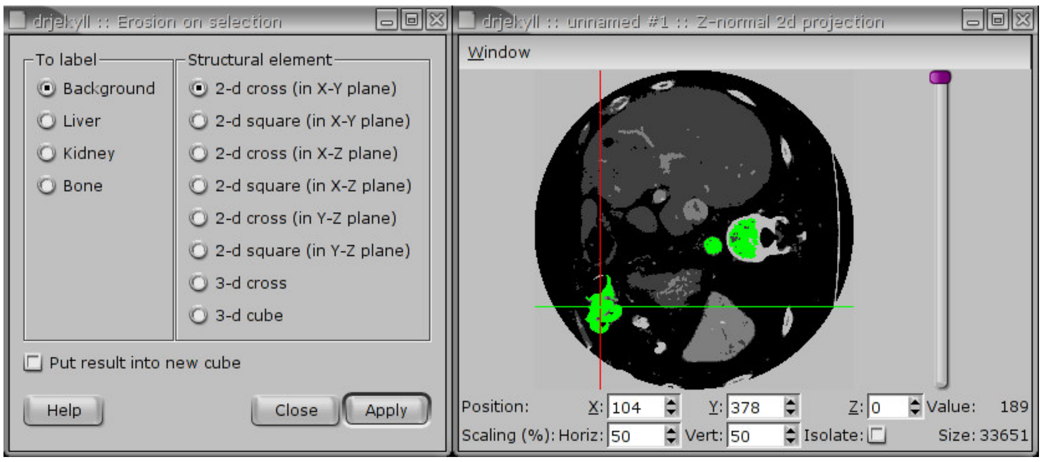
Since the technique outlined above allows for a large number of the algorithms, with the same signature, it should be possible to write a generic GUI control,

```
template<typename VT, typename STRATEGY>
class Selection : public Plugin {
  // Constructors (which build the GUI) and destructor.
  void apply () { // Activated when the Apply button is clicked.
    STRATEGY S; // Instantiate a concrete algorithm
    // Generate image from selection.
    blitz :: Array<VT, 3> out = S(image , ...);    // Call the concrete algorithm.
    // Map out into the original image.
  }
}
// Different Selection objects can be made like this:
Selection<uint8_t, Dilation > Dilationplugin ;
Selection<uint16_t, Opening> Openingplugin;
```

Figure 3.9: Code example of how templates can be used to implement the *strategy* design pattern, promoting loose coupling between the algorithms and the user interface class. Notice that the objects created at the end uses the uppercase version of the algorithm name, which is the functor object encapsulating the actual algorithm. The first template parameter is for different voxel types.

which can be use for all algorithms with the same parameter list.

The implementation of the dialogs for morphology in *Dr. Jekyll* uses type parameterization to achieve this goal. Figure 3.9 illustrates how this can be done. The `Selection` class, which has methods to the generate the GUI, and for applying an operation only to the selected components, takes two template parameters. The first one is for the data type of the underlying image, the second can take any object matching the return value and parameter types where the object is used in the


drjekyll :: Erosion on selection

To label: [✓ Background] Liver Kidney Bone

Structural element: [✓ 2-d cross (in X-Y plane)] 2-d square (in X-Y plane) 2-d cross (in X-Z plane) 2-d square (in X-Z plane) 2-d cross (in Y-Z plane) 2-d square (in Y-Z plane) 3-d cross 3-d cube

☐ Put result into new cube

drjekyll :: unnamed #1 :: Z-normal 2d projection

Position: X: 104 Y: 378 Z: 0 Value: 189
Scaling (%): Horiz: 50 Vert: 50 Isolate: ☐ Size: 33651

Figure 3.10: The dialog for the selection based erosion operator. Several labels can be eroded into the tolabel, and a variety of structural elements are available.

`apply()` method.

In order to be able to instantiate the `Selection` class with different algorithms, the algorithm must be available as a *type* and not only as a callable function. The functor dilation object figure 3.8 is an example of how such types can be made.

Such encapsulation of the inner workings of an algorithm is an example of the *strategy* design pattern[15]. It is not a variant of the *decorator* pattern, even if the trick of passing a “decorating” object as a template parameter has been demonstrated[61, 62]. In the decorator pattern, it is not *required* to decorate all instances of the class, the dialog above however cannot work without a functor type being passed.

It should be noted that this example could alternatively be realized by using subclassing, as illustrated in figure 3.11. The `Selection` class has a pure virtual function (a function which all subclasses must implement). For every algorithm which is to be called a new subclass has to be derived, the subclass implement the pure virtual function and forward the function call to the actual algorithm. As can be seen from figures 3.9 and 3.11, the example using subclassing is longer and introduces a virtual function call (which has a slight runtime overhead). The main disadvantage of using subclassing instead of static typing is the tighter coupling between the inheritance hierarchy and algorithms. If the algorithms change parameter list, every subclass acting as a forwarding function has to be updated to reflect this, by using static typing, the actual call to the algorithm is isolated to only one line of code.

A third option, embodying the actual algorithm in a subclass of `Selection`, also deserves mention. To achieve any code reuse at all, it would have to be implemented in form of a virtual function call like figure 3.11. However, it would be impossible to call the algorithms without instantiating a complete graphical dialog as well, making implementation of opening and closing much troublesome than they need to, since calling the code for opening and dilation would also call all the code for the dialog.

Finally, an illustration of the GUI for a selection based erosion is shown in figure 3.10. This dialog-algorithm combination was made using the technique explained in this section. However the actual inheritance hierarchy for the dialog shown is more complex than this example suggest. This was done partly to overcome Qt’s limitation with regards to templates (see section 2.2.2), and also to add different help texts for the the different morphological operators.

Alternative implementations At the time *Dr. Jekyll* was started, no generic library for doing mathematical morphology on volumetric datasets were available. However in spring 2002 the Olena library was announced by the LRDE

```
template<class VT>
class Selection : public Plugin {
  // Constructors, destructors etc.
  // Pure virtual forwarding function:
  virtual void blitz :: Array<VT, 3> algo(image , ...) = 0;
  void apply () { // Activated when the Apply button is clicked.
    // Generate image from selection etc.
    blitz :: Array<VT, 3> out = algo(image , ...); // Call to virtual function
    // Map out into the original image.
  }
}

template<class VT>
class OpeningSelection : public Selection<VT> {
  void blitz :: Array<VT, 3> algo(image , ...) {
    return opening(image , ...);
  }
}

template<class VT>
class ErosionSelection : public Selection<VT> {
  void blitz :: Array<VT, 3> algo(image , ...) {
    return erosion(image , ...);
  }
}

// Concrete instances can be made like this:
ErosionSelection<uint8_t> Erosionplugin;
OpeningSelection<uint16_t> Openingplugin;
```

Figure 3.11: An alternative way figure 3.9 could have been implemnted. This example uses subclassing and virtual functions instead static typing to achieve the same effect. The net result is more code lines, and tighter coupling between code, since all subclasses acting as forwarding functions has to be altered if the parameter list of the morphological operators change.

Lab at EPITA (École pour L'Informatique et les Techniques Avancées) in Paris, France[9]. This library provides a complete framework for doing mathematical morphology in C++ and it is available under an open source license. While the OLENA is more polished than *Dr. Jekyll* for doing mathematical morphology, they share many design philosophies, like the use of templates to support various bitdepths of images. As well as heavy use iterators to support arbitrary SE's, and the chaining of operators.

3.4.4 Applications of morphological operators

For doing interactive postprocessing of segmented images, the morphological operators provide a unifying theory for altering the shape and form of either the entire image, or a subset of it (based on the connected components analysis).

The morphological operators are highly adapted for manipulating shapes. By providing an interactive GUI they allow for complex, non-linear filtering of images or components of images..

Furthermore, two other uses of morphology deserves special mention:

1. Surface extraction can be done by the half-edge gradient (also called the internal gradient). If all labels are eroded, and the eroded image is subtracted (pixel by pixel) from the original image, only the contours of each component are left;

   $$\rho_B^- = I - \epsilon_B. \tag{3.1}$$

   Here I denotes the original image.

   For a segmented image, where a change in pixel value defines a component boundary this is a faster way of extracting the surface pixel than the marching cubes algorithm, since the aim of the later it to extract isosurfaces.

   This techniques is used by the three dimensional visualization technique to highlight comoponent selection (see section 3.6.3).

2. While not implemented in *Dr. Jekyll* the watershed transformation for image segmentation is based on mathematical morphology. If *Dr. Jekyll* is to be extened to be an segmentation tool as well, the watershed transform is a prime candidate for the segmentation algorithm, and it can utilize the framework already in place for doing morphology.

3.5 Component based relabeling

The most important task during postprocessing is to relabel pixels. Morphological operators are one way to achieve relabeling based on local neighborhood operations, by nature the morphological operators operate on single pixels in order. However another approach is possible, namely relabeling based on *components*. These components arise from the connected components algorithm described in section 3.3.

This tool do not use any sophisticated algorithms, but nevertheless the implementation requires some trickery to get right in order to be flexible, so an overview of how it was implemented is in order.

3.5.1 Interactive Component based relabeling

A very simple way to reclassify a component is to manually select the component (which is highlighted in green by the *Dr. Jekyll* GUI), and then select the label the selection will be converted to.

The actual relabeling is also trivial. Every pixel in the selected components bounding box is traversed, and if they belong to the component, the label is updated accordingly in the pixel data for the underlying image.

3.5.2 Rule based

Rule based relabeling is a much more interesting and complex task. The connected components analysis calculate information which can be used for rule based relabeling:

- Size of component
- Neighbors of component

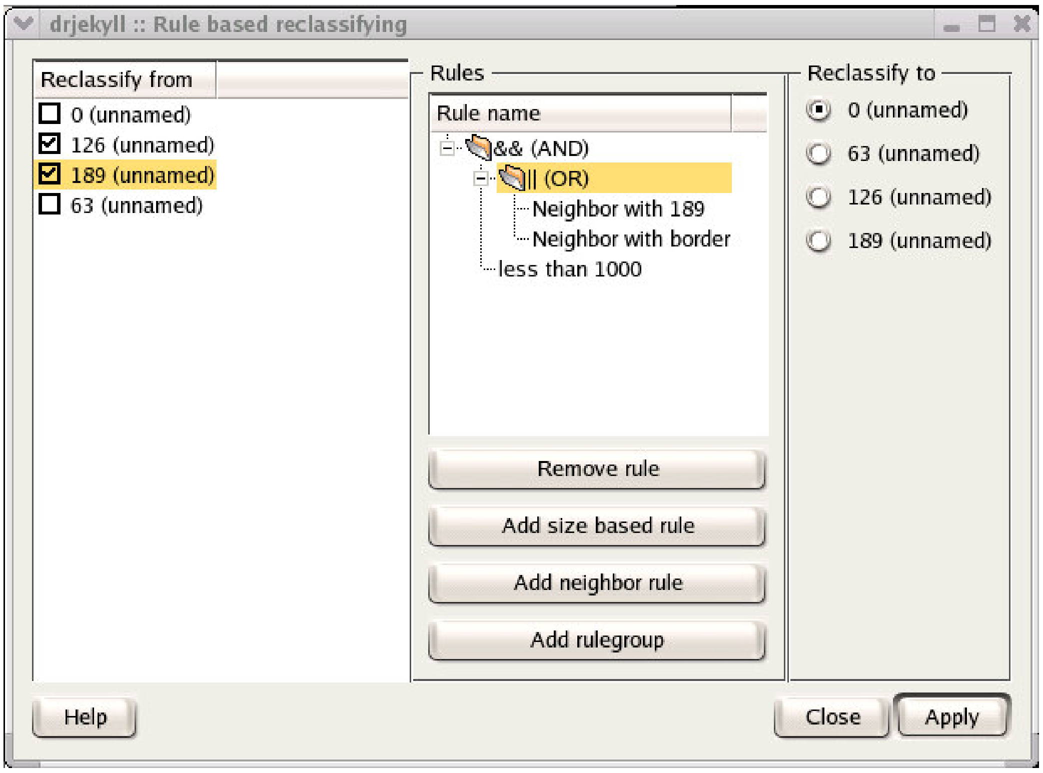
While not itself an impressive list, the logical rules that can be built from this information can create complex transforms of the image. An example of such a rule could be *reclassify components (with size less than* 250 *pixels AND connection to the liver label) OR size greater than* 500 *pixel OR connection to the border to the background label*.

Doing such a reclassification by hand would be almost impossible, however the connected components analysis provides exactly the information needed to find the components which shall be reclassified. A screenshot of the GUI for creating such rules are presented in figure 3.12.

3.5.3 Implementation of the rulechains

The programming of such “chains” of rules of arbitrary length and depth requires some trickery to get it right, but it was made easier by one of C++ generic programming mechanisms, namely *functors*. Since the implementation is highly tied to the STL and the *C++-way* of doing things, some vocabulary must be defined:

- A *functor (function object)* is simply an object which can be called as if it is a function (with the `f(...)` syntax). Any class which implements the `operator()` method is thus a functor. Like any other class a functor object preserves states throughout the lifetime of the object.


drjekyll :: Rule based reclassifying

Reclassify from
- [ ] 0 (unnamed)
- [✓] 126 (unnamed)
- [✓] 189 (unnamed)
- [ ] 63 (unnamed)

Rules

Rule name
```
&& (AND)
  || (OR)
    Neighbor with 189
    Neighbor with border
  less than 1000
```

Reclassify to
- [✓ 0 (unnamed)]
- 63 (unnamed)
- 126 (unnamed)
- 189 (unnamed)

Figure 3.12: The GUI for creating arbitrary chains of rules for relabeling of components.

- A *predicate class* is a functor returning either true or false for its *operator()*. A predicate taking as input exactly one parameter is called an *unary predicate*. (If it takes two inputs it is called a *binary* predicate etc.)
- An *adaptable functor* is a functor which provides typedefs for its return and parameter type(s). Adaptable functors may be used by *functor adapters*, functors that transform or manipulate other functors. The STL provides pre-declared base classes (called `unary_function` and `binary_function`) to simplify the declaration of adaptable functors.

The goal of the implementation is to create a boolean functor which can be queried for each component in the in the image. The result value of the functor decides if the actual component is to be relabeled. The functors corresponds to either an actual rule (is the size of component x larger than n pixels), or they can be a logical group. For a group of logical “ANDs”, all the members of the group has to return true for the group to return true.

The bottom row of figure 3.13 shows the concrete rules which are included in *Dr. Jekyll*. However the inheritance hierarchy for the rules are quite complicated, due

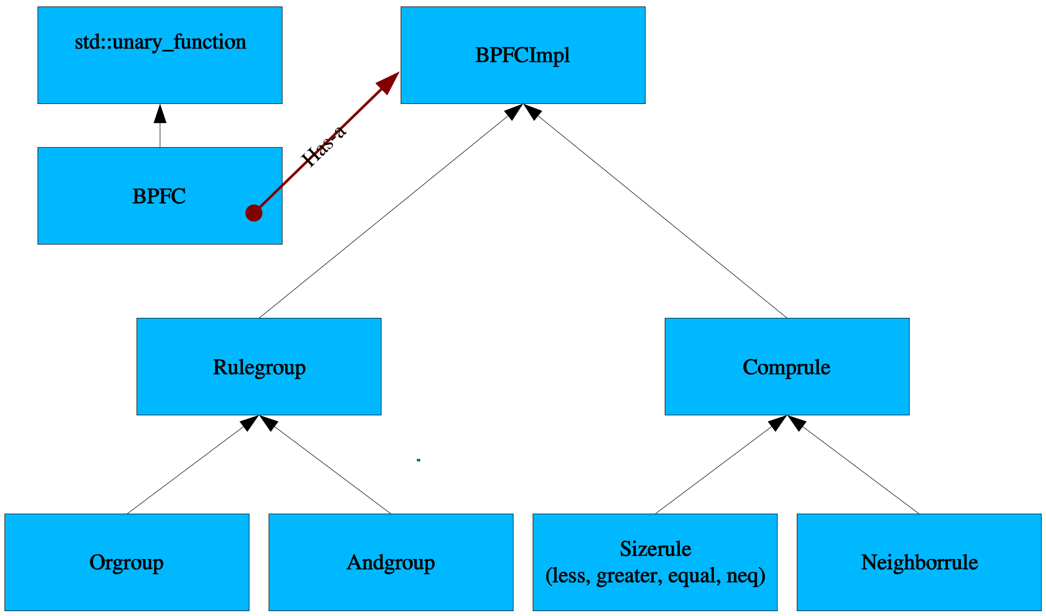


Figure 3.13: Inheritance hierarchy for the implementation of the rule chains used in the rule-based relabeling algorithm. The red line indicates a pointer, the black lines indicate inheritance. The functor class `BPFC` (Big Polymorphic Functor Class) is in reality a very small monomorphic class, but it holds a pointer to its implementation part, called `BPFCImpl`.

to a limitation of the STL; functors must designed so that they can be passed by value (and not by reference). This constrains the functors to be monomorphic (that is not polymorphic), ie. they must not contain virtual functions. To overcome this limitation, and allow for code reuse between the rules, a proxy class, called `BPFC` (Big Polymorphic Functor Class) was created. This class has no virtual functions, but it holds a pointer to an implementation class (`BPFCImpl`), which has the "real" implementation. A call to `operator()` of a `BPFC` instance, is forwarded to the `BPFCImpl`'s `operator()` method. This is illustrated in figure 3.14. (It should be noted that the implementation in *Dr. Jekyll* is a bit more complicated than this, since it uses smart pointers to avoid memory leaks when the last instance of a `BPFC` goes out of scope.)

The class hierarchy above is inspired by Meyers solution to overcoming the monomorphic functor limitation if [36]. The technique of hiding the implementation of a class behind a pointer is called the "Bridge Pattern" by Gamma in *Design Patterns* [15], and the "Pimpl Idiom" by Sutter in *Exceptional C++* [52].

```
template<class T>
class BPFC
{
private:
    T∗ pImpl;
public:
    BPFC( T∗ impl ) : pImpl( impl  ) {};

    bool operator ()( ... )  const
    {
        return pImpl−>operator ()( ... );
    }
};

template<class T>
class BPFCimpl
{
    friend class BPFC<T>;
public:
    virtual ~BPFCimpl() {};
protected:
    // = 0 at end means pure virtual.
    // It has to be reimplemented in subclasses.
    virtual bool operator ()( ... )  const = 0;
};
```

Figure 3.14: The implementation of the Pimpl idiom or bridge pattern. A small monomorphic (without virtual functions) delegates the call to `operator()` to an implementation class, which can have virtual functions. (Notice that the implementation sketched here is prone to memory-leaks, since functors can be copied, and there is no mechanism to delete the pointer when the last `BFPC` instance goes out of scope.)

3.6 Visualization

The preceding sections has given an overview of the algorithms implemented in *Dr. Jekyll*. For an interactive application, and for deciding what algorithms to apply (and to what parts of the image) it is necessary to have efficient visualizations of the image as well.

This section discuses the benefits and disadvantages of various established visualization techniques, and does not aim to give a detailed account of the techniques themselves. However to form a basis for discussion, a short summary of the various techniques work are given.

The reader will notice that none of the algorithms described in this section was actually implemented in *Dr. Jekyll* by us, but were found in libraries. Implement-

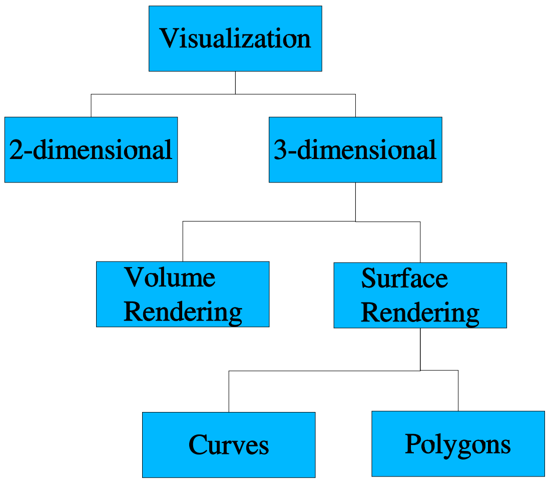


Figure 3.15: Hierarchical overview of the different visualization techniques discussed.

ing high-quality visualization algorithms from scratch is a very time consuming process, and were beyond the scope of the *Dr. Jekyll* project.

The classic text *Computer Graphics: Principles and Practice* by Foley, Van Dam et Al[14] gives a detailed introduction to the field of computer graphics, but it is becoming a dated. Still it includes a good overview lightning and shading models, polygonal representations etc. An up to date, modern text, specially dealing with how to utilize modern graphics hardware to achieve interactive framerates for 3-dimensional models is *Real-Time Rendering* by Akenine-Moller/Haines[1]. It scope is a bit narrower than [14] though.

For visualizing volumetric 3-dimensional images on a 2-dimensional computer screen, two different approaches can be taken. Namely sliced based 2-dimensional visualization or 3-dimensional visualization. Figure 3.15 contains a hierarchical overview of the different visualization techniques which were evaluated for use in *Dr. Jekyll*, and which are discussed below.

3.6.1 Slice based 2-dimensional visualization

A simple, but yet very effective way to visualize a volumetric dataset is to extract slices from the volume. For a dataset on a regular grid, it is possible to extract slices along perpendicular axes, allowing for multiple views of the same data. Images arising from CT and MRI scanners has traditionally been diagnosed this way, by printing out the slices on film, and using backlight whiteboards to see each slice. Radiologist and doctors are therefore used to seeing 3-dimensional images

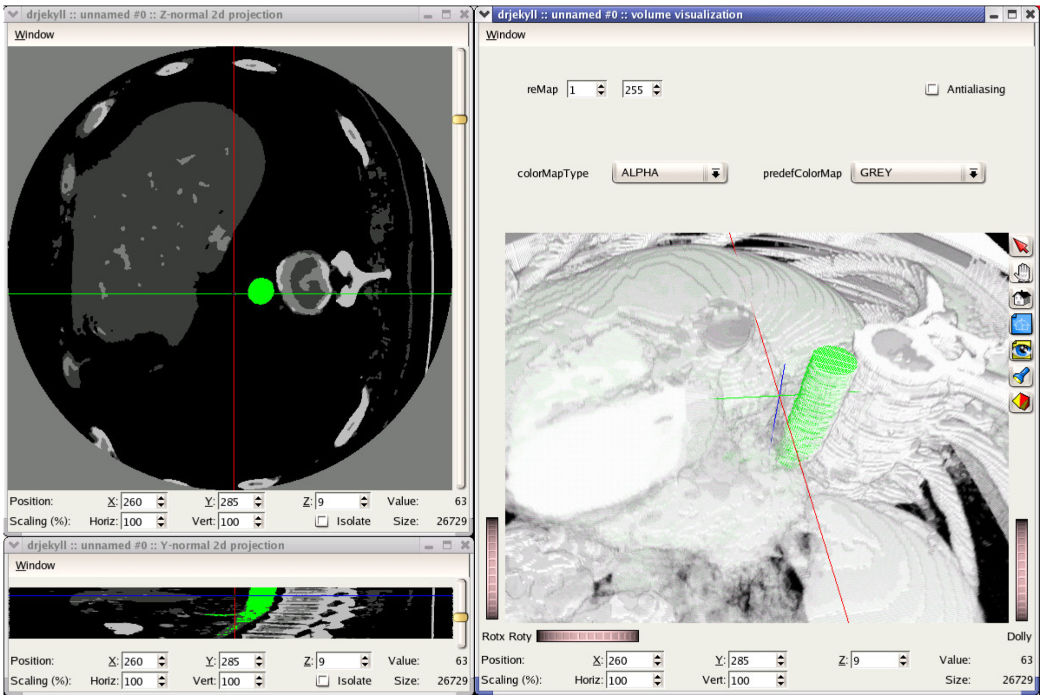

drjekyll :: unnamed #0 :: Z-normal 2d projection
Window
Position: X: 260 Y: 285 Z: 9 Value: 63
Scaling (%): Horiz: 100 Vert: 100 Isolate Size: 26729
drjekyll :: unnamed #0 :: Y-normal 2d projection
Window
Position: X: 260 Y: 285 Z: 9 Value: 63
Scaling (%): Horiz: 100 Vert: 100 Isolate Size: 26729
drjekyll :: unnamed #0 :: volume visualization
Window
reMap 1 255
Antialiasing
colorMapType ALPHA
predefColorMap GREY
Rotx Roty
Dolly
Position: X: 260 Y: 285 Z: 9 Value: 63
Scaling (%): Horiz: 100 Vert: 100 Size: 26729

Figure 3.16: This screenshot of shows both kinds of visualization techniques which *Dr. Jekyll* provide. Slice based 2D visualization is available along perpendicular axes in the datasets (to the left). A volume visualization provides a 3D overview of the entire image. A moveable cross is synchronized between the different visualizations, allowing for easy navigation throughout the volume. A selection (based on the connected component analysis) is highlighted in green.

this way. The left part of figure 3.16 shows a CT image being visualized along two perpendicular axes, with a cross highlighting the same pixel in both views.

There are other advantages to utilizing 2-dimensional slice based visualizations as well. Primarily, it is the only visualization technique which allows for a direct correspondence between image pixels and screen pixels. No perspective is calculated, and thus pixels which are neighbors on the screen are really neighbors in the underlying image dataset as well.

Furthermore, since only a small subset of the image is be visualized, it can be very fast. This is partly because very few calculations has to be made to visualize the slice. But also, the number of pixels which has to be considered is much less than the entire image, effectively reducing the running time.

In theory it is also possible to extract slices along axes which are not perpendicular to the dataset. This approach would lose all the benefits stated above, since in is no longer possible to extract “exact” slices of the data. This removes to one-to-one correspondence between image and screen pixels, as well as introducing more complex calculations for the visualization. Arbitrary slices is therefore not supported in *Dr. Jekyll*. This functionality would probably be better to provide as cutplanes through the 3-dimensional visualization.

3.6.2 3-dimensional visualization

As humans we are accustomed to living in a world with three spatial dimensions, and our vision systems reflects this. The human eye is remarkable in its ability to "see" depth in the real world. Much of this information is gained by looking at the shadows and reflections of the objects we are looking at. Even if a computer has a representation of a 3-dimension world in it's memory, it can not directly output this data to a 2-dimensional computer screen. Projection, lightning and shading has to be calculated in order to give a human observer looking at a computer screen the illusion that he is looking into a 3-dimensional world.

Such calculations involve floating point operations, and can be very processor intensive, so dedicated graphics chips were developed in the late 1980's. Initially such dedicated graphics hardware were the domain of high-end and high-cost hardware, specially from SGI (previously Silicon Graphics). However in the late 1990's dedicated hardware for 3-dimensional acceleration started to become common on desktop computers, thanks to the gaming industry. Today almost every computer sold, including laptops, is equipped with dedicated graphics hardware. Such hardware is often called 3D- accelerators or GPU's (Graphical Processing Units). The speed and complexity of a modern 3D-accelerator is quite amazing, and the latest incarnation of the graphic chips from Nvidia and ATI includes more transistors than top of the line processors from Intel[2]

While commodity GPU's are primarily designed for games, they can also be used for scientific visualization, making high performance and large dataset visualization affordable on commodity hardware.

3D-accelerators work by implementing very effective, brute force algorithms in hardware. These algorithms include algorithms for geometry (like transformation, scaling and rotation), lightning and shading as well as the final projection on the computer monitor. Thus allowing for interactive framerates of even highly complex models.

Even if a computer is equipped with a GPU, it is a non-trivial task to convert a 3-dimensional image into a model which can be rendered by the GPU. One possibility is the generate a surface of the various objects in the image (if such exists), another is to try to take every pixel into account when visualizing the

[2]ATI's R300 chip contains 110 million transistors, while the Intel Pentium IV contains 55 million. However the R300 has eight parallel pipelines, greatly increasing the transistor count.

image on the screen. These techniques are called surface and volume rendering respectively, and are explored below:

Surface visualization

Most GPUs are traditionally very well suited to rendering surfaces, since they are primarily designed for rendering models where a geometric representation already exists. However, how to generate the geometric model, and what form the geometric representation shall take, is not obvious.

The geometric representation can basically take two forms:

1. Parametric surfaces

   A parametric surface, is typically a mapping from a rectangular domain Ω into space: $f : \Omega \in \mathbb{R}^2 \rightarrow \mathbb{R}^3$.

   The mapping f can take several forms, including Bezier patches and tensor product spline surfaces. The exact definition of f can vary, but for the sake of the discussion it is enough to know that there exists several ways to generate f from the surface points of an object, and that the level of mathematical continuity can be as high as necessary. (But usually C^2 or C^3.)

   The advantage of surfaces is that relatively few points (often called control points) has to be stored for a complex geometric model, as well they guarantee (mathematical) continuity, simplifying lighting calculations. Furthermore, by moving these points interactively it is possible to alter the shape of an object, making parametric surfaces a good tool for modeling. An implementation using Catmull-Rom splines for editing medical volume images is demonstrated in [10].

   It should however be emphasized that a parametric surface seldom represents the object exactly, but represents an arbitrary close approximation to the surface.

   The disadvantage of parametric surfaces is that there is no generic way to parameterize an arbitrary collection of surface points. To generate a correct model, more information has to be known about the object being represented, including it's genus (number of holes) and if it is an open or closed surface.

   Another way to generate a parametric, continuous approximation of a surface, is to first generate a representation of the surface using triangles, and thereafter calculate a parametric surface over the resulting triangulation. This is of course even more computationally expensive, requiring both a triangulation step and thereafter a “surface” step.

When rendering parametric surfaces, it is usual to convert the parametric surface to polygons (see below). Algorithms for doing this is computationally and conceptually simple, and allows for utilizing the direct hardware support for rendering polygons, while at the same time keeping an accurate, continuous representation.

2. Polygons

   When raw rendering speed is crucial, the fastest way to describe the surface is by polygons, and very often the triangle is the preferred polygon. Since, by definition, the three point spanning a triangle is planar. In addition to storing the vertexes, also the normals for each polygon has to be stored, in order to generate correct lighting.

   The disadvantages of the polygonal representation is that interactive editing becomes an almost impossible task, since the number of triangles is very high. Furthermore, a polygonal representation is also an approximation to the underlying data, and if several objects close to each are visualized, they could have overlapping representations.

   For volumes consisting of a scalar field, the standard algorithm for generating a surface based on triangles is the *marching cubes* algorithm[31]. It is designed to extract ISO-surfaces of a volume, and is therefore not the best algorithm when the object boundaries are clearly defined as in a segmented image. The marching cubes algorithm is also patented, restricting its use in a commercial setting, thus making the legality of an open source implementation dubious.

   For segmented images, where a connected components analysis already exists, it should not be necessary to go via the "marching" step. However, as of yet there exists no standard, well proven technique to avoid the "marching" step. Though it is a field of active research.

The main disadvantage of surface visualization is the high cost associated with calculating the surface. The running time of calculating the surface is acceptable if they were only to be run at program startup. However since the algorithms of *Dr. Jekyll* may change the shape and form of the dataset, in can be necessary to recalculate the surface many time throughout the programs lifetime.

Another cost associated with generating a geometric model, is that one has in effect to store two representations of the image. Both the raw image data, and a geometric model (and possibly even the normals of the geometric model as well).

However on possible end-product of postprocessing, and thus of *Dr. Jekyll* itself is to have a refined dataset which is suitable for surface extraction. As a way to

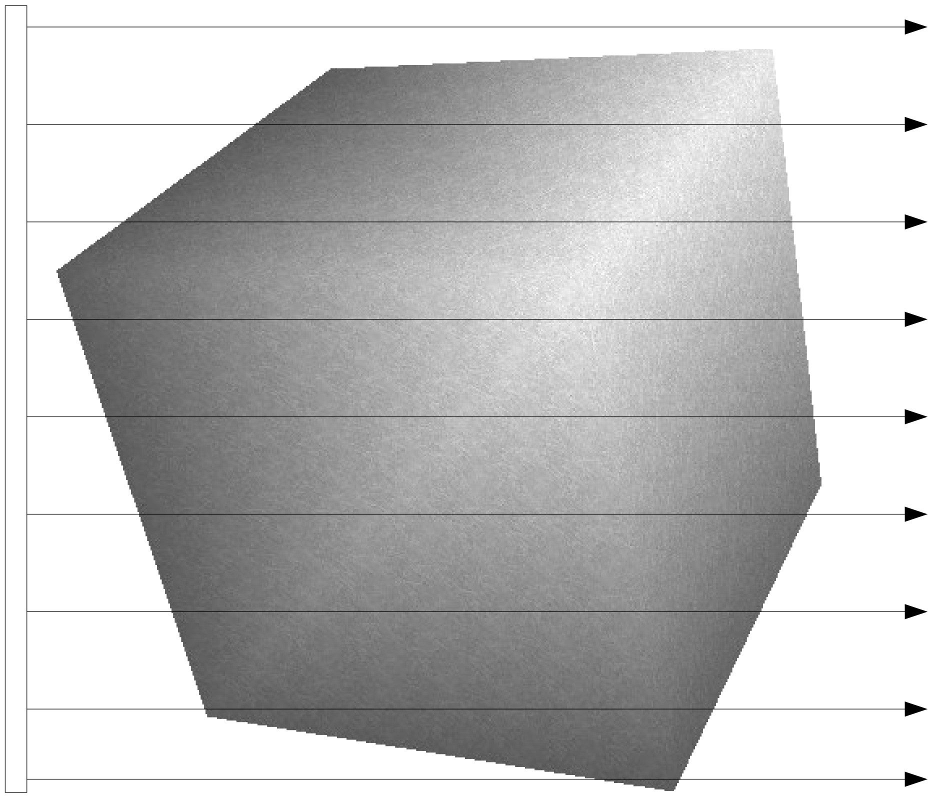


Figure 3.17: The basic principle of volume rendering. A “ray” is cast from each pixel (perpendicular to the screen) through the volume data. Each pixel the ray touches accumulates to the final pixel value.

visualize the images while they are being processed, generating the model is to computationally expensive.

Direct volume rendering

Direct volume rendering is conceptually very different from surface rendering. Instead of extracting and representing surfaces of a volume, each pixel (or data point) is taken into consideration when generating the final image, this is done by a technique known as ray-casting.

Figure 3.17 illustrates how a ray is cast perpendicular to the screen from each screen pixel throughout the data volume. The final intensity value of each pixel is given by the volume rendering integral:

$$I(a,b) = \int_a^b g(s) e^{\int_a^s \tau(x)\, dx}\, dx \tag{3.2}$$

Here $g(s)$ describes the lighting model used in the ray casting. It can be a very simple lighting model, or a more sophisticated lighting model like the Phong model. The other function, $\tau(x)$ defines the transparency of each voxel.

On modern GPUs this calculation can be done in hardware, by loading the pixels into the memory of the GPU, but CPU based techniques also exists. Both methods rely on using numerical integration to calculate the volume rendering integral.

Using GPUs to do the volume rendering however has one serious limit. In order to do the calculations the pixel data has to be loaded into the GPUs, and like a the main processor the GPU has also has a limited amount of memory.

Volume rendering is still a young field, and both techniques and applications is a subject of rapid research. An overview of techniques and applications on general hardware is given in [43]. The mathematical foundations and early techniques is given in [29], though it is becoming a bit dated and was written before GPUs became mainstream on commodity hardware.

3.6.3 Implementing the visualizations

As stated above, implementing effective visualization algorithms, to take advantage of dedicated graphic hardware, is not an easy task. Therefore libraries were used to generate the 3-dimensional visualizations.

Segmented volumes should be very well suited for surface visualizations, since object boundaries can be generated directly, and the connected components analysis group pixels into a unique object to which they belong. Furthermore the half-edge gradient (see equation (3.1)) can be used to extract the surface.

Since Open Inventor was chosen as the library for developing the visualizations, it was experimented with using an implementation of the marching cubes algorithm[31] written by Josh Grant[18] for Open Inventor.

While this implementation provided satisfactory visual results, the running time for generating the visualization was to slow for interactive usage. If this processing only was necessary to do when loading a new dataset, the delay would have been acceptable. However since the various tools provided by *Dr. Jekyll* alters the shape and form (and thus the object boundaries) of the dataset. It is required to recalculate the visualization each time the data is updated, rendering the application to slow for interactive usage.

Since the marching cubes algorithm is patented, the legality of distributing an implementation of it is dubious. Licensing issues regarding the implementation provided Josh Grant was also clouded by this. Even reimplementing the algorithm from scratch will have these limitations, since it can not be published under an Open Source license and thus not legally linked with the Qt and Coin libraries of *Dr. Jekyll*.

Volume rendering has the benefit that all the processing can be done in hardware, provided the GPU has a copy of the image data. If the image data is stored contiguously in memory (like *Dr. Jekyll* do), it is an extremely fast operation to load the image data into memory, resulting in near realtime feedback for algorithms which alter the objects.

A change of the shape and connectivity of the dataset, still requires the image data to be reread into the GPU and recalculated, but this happens in a fraction of a second on a modern workstation, in contrast to surface based methods which takes several seconds.

Systems in Motion[79], developers of a freely available Open Inventor clone named Coin, kindly provided us with a prerelease version of their volume rendering extensions to Coin, called SimVoleon. Using this library, volume visualization of images in *Dr. Jekyll* could easily be developed, and was as easy as giving a pointer to the image data to the right function. This is so since the library plugs easily into the scene-graph based architecture of the Coin library.

Highlighting a selection

It has previously (see 3.3 been emphasized how the connected components analysis is a valuable tool for image navigation in 3-dimensional images, since it allows for visualizing spatial structures with connections in all parts of the volume.

The volume visualization of *Dr. Jekyll* uses the half-edge gradient by erosion (see equation (3.1)), using a 3-dimensional structural element to calculate the surface of a selection. The selection is therefore the 3-dimensional contour of each component. The surface is visualized as points, allowing the color from the objects inside of the selection to be seen through the highlighting. The right window in figure 3.16 shows a volume rendering with a component highlighted using this technique.

3.6.4 Conclusion for visualizations

High-quality and high-speed visualization for arbitrary images is a field of its own, and could easily constitute the focus of entire research projects. The timeframe of the *Dr. Jekyll* project has not allowed for finding optimal visualizations for the image data, even if we have relied on libraries and not implemented the algorithms ourselves.

The volume rendering approach which is the only 3-dimensional visualization which has been completely implemented, has the advantage of not requiring extra storage space, nor calculations of the image data. However a volume rendering is not a good base for developing three-dimensional editing tools.

The present state of *Dr. Jekyll* which combines 2D slices and volume rendering seems to be a good compromise for visualizing volumetric images, since the raw data is visualized, the interactive feedback on the result of algorithms is quite good.

However, there is no one-size-fits-all solution to visualizing volumetric images, and it would be a nice addition to *Dr. Jekyll* to provide more visualization front ends.

3.7 Conclusion

This chapter has provided an overview of the central algorithms which forms the computational base of the *Dr. Jekyll* application. Though none of the algorithms are novel, the image processing algorithms have had to be extended from an algorithm operating on binary images to an algorithm which can be applied to labeled, segmented images.

Furthermore is has been demonstrated how the algorithms are realized in modern C++, with the aim of being generic and supporting different structural elements without sacrificing running time.

Chapter 4

Results and discussion

The most important result of this work is the *Dr. Jekyll* application itself. It compiles and runs, and is available to everyone who wishes to use it or extend it under an open source license. The following chapter summarizes the project from bottom to top, starting with the algorithms implemented, continuing with a description of the user interface and all the plugins implemented, and ending with its usefulness for manual post-processing.

4.1 The algorithms

Early in the process of writing *Dr. Jekyll*, the algorithms detailed in chapter 3 were identified as potentially useful algorithms for volumetric post-processing. Whether it was possible to achieve interactive performance of these algorithms for large images, however, was an open question. if the licenses are legally compatible is just as important to verify as if they are technically compatible. A lot of effort was put into finding an effective algorithm for connected components analysis, and the Thurfjell[57] algorithm implemented is much faster than the original algorithm by Rosenfeld and Pfaltz[39]. The difference in running time between these two algorithms is crucial to the interactivity of the application.

Implementing the morphological operators was much easier than implementing the connected components analysis. The algorithms are much simpler and the straightforward implementation is fast enough for interactive performance, even for large images.

Finding effective visualization techniques for 3D images proved to be more challenging than originally assumed. Calculating geometry for each component is too costly since they have to be updated so often. The best results were achieved by techniques which could visualize the image directly without costly calculations, utilizing modern graphics hardware to ease the burden of the CPU.

4.1.1 Algorithm optimizations

Donald Knuth is quoted as saying *Early optimization is the root of all evils*[26]. However, this does not mean that one should never optimize the implementation of an algorithm. One should wait until the algorithm is correct and stable before optimizations take place. Furthermore blindly optimizing an algorithm is not a good option, since modern compilers do a lot of optimizing on their own. Many optimizations are better let off to the compiler than to the programmer.

However, the two points presentif the licenses are legally compatible is just as important to verify as if they are technically compatible.ed below show that manual optimizations also has a great effect on the running time. Actual timings for the benchmarks are included in appendix A.

1. Avoid cache misses:

   The importance of traversing images the way they are stored in memory has previously (section 3.1) been discussed. Benchmarks on the connected components analysis shows a performance penalty of between 20% and 60% on different CPU's when traversing in the wrong direction.

2. Avoid object allocation in tight loops:

   The Thurfjell algorithm uses two STL vectors to merge different parts of a component found to belong to the same component. By allocating these vectors before the tight loop and resetting them inside it (as illustrated in figure 4.1), instead of allocating them for every iteration, the running time was reduced by nearly an order of magnitude on the Intel Pentium IV CPU! This is by no means a new finding, several texts, including [52, 35, 2] documents how expensive object allocation can be. However since the results are dramatic, and it is quite an easy error to make, it is worth emphasizing. Specially since modern CPUs probably wiif the licenses are legally compatible is just as important to verify as if they are technically compatible.ll be more prone to such issues.

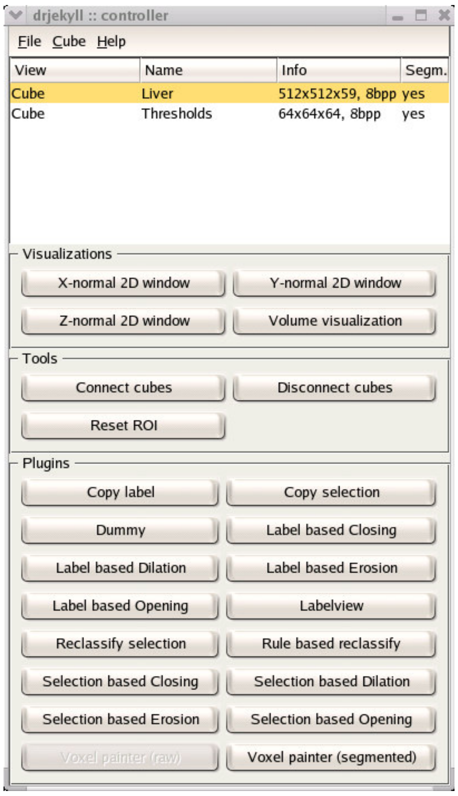
4.2 The application

The previous chapter has shown several screenshots of the *Dr. Jekyll* application in action, with special focus on the connected components analysis. Here follows

```
for_each_pixel(image) {
   std :: vector  V;  //  Creates a new object.
   //  inner  loop ,  which uses V.
}
```

```
std :: vector  V;  //  Creates new object.
for_each_pixel(image) {
   V.clear ();  //  Resets old object.
   //  inner  loop ,  which uses V.
}
```

Figure 4.1: Two code examples which illustrates how unnecessary object allocation inside tight loops can be avoided, if the object supports resetting. For the Thurfjell connected-components algorithm this technique reduced the running time by an order of magnitude.



View	Name	Info	Segm.
Cube	Liver	512x512x59, 8bpp	yes
Cube	Thresholds	64x64x64, 8bpp	yes

Figure 4.2: The main controller dialog of the *Dr. Jekyll* application. The top level menu bar can be used to open, close and save cubes. Thereafter follows a list of all opened images. The buttons at the bottom are divided into three different groups: The first one controls the different visualizations, the one in the middle has tools for synchronizing visualizations between different images. The last button group contains a button for each plugin, doing the actual image modification.

a description of the capabilities of the application, with special focus on what the different plugins do.

An image of the top-level menu widget of the application is given in figure 4.2. The top level menu bar can be used to open, close and save cubes. Next follows a list of all images which is opened by the application. Thereafter follows three groups of buttons.

1. The first group of buttons can be used for opening different visualizations of the images, both slice based and the volume rendering.

2. The next group contains buttons for connecting and disconnecting cubes. Cubes that are connected will have their visualizations “follow” each other. This is useful for data exploration, both raw and segmented images can be studied, and one can see what a point in the original image has been segmented to.

   There is also a button for resetting the current ‘region of interest”.

3. The last group contains all the plugins. These described separately below:

   - Copy label: Is used to copy all components of a label into a new cube. The new cube will be a binary image, consisting only of background and the extracted component.
   - Copy selection: Is used for copying a selection of components into a new cube. Multiple components may be selected, and they can can belong to different labels.
   - Dummy: This is simply a plugin doing nothing, it displays all signals it receives and is useful for testing purposes. Furthermore since it is not cluttered by any functionality, its source code should be easy to read by new aspiring plugin writers.
   - Label based closing, dilation, erosion and opening: These plugins allows for applying a morphological operator to all pixels of the given label.
   - Labelview: Displays all labels present in the current image and what value they are assigned to. It can also be used to introduce new labels into an image. Components may later be relabeled to the new label.
   - Reclassify selection: This plugin is used to reclassify the selected components into a new label.

- Rule based reclassify: Allows the user to generate complicated queries based on information generated by the connected components analysis. All components for which the query is true can be relabeled to another component.
- Selection based closing, dilation, erosion and opening: Like their label based cousins above, these let the user specify a number of components to apply a morphologic operator to.
- Voxel painter (raw and segmented): These let the user directly manipulate voxels in the image, by freehand drawing in the 2D slice visualizations. The label and shape of brush to draw with can be selected from a dialog.

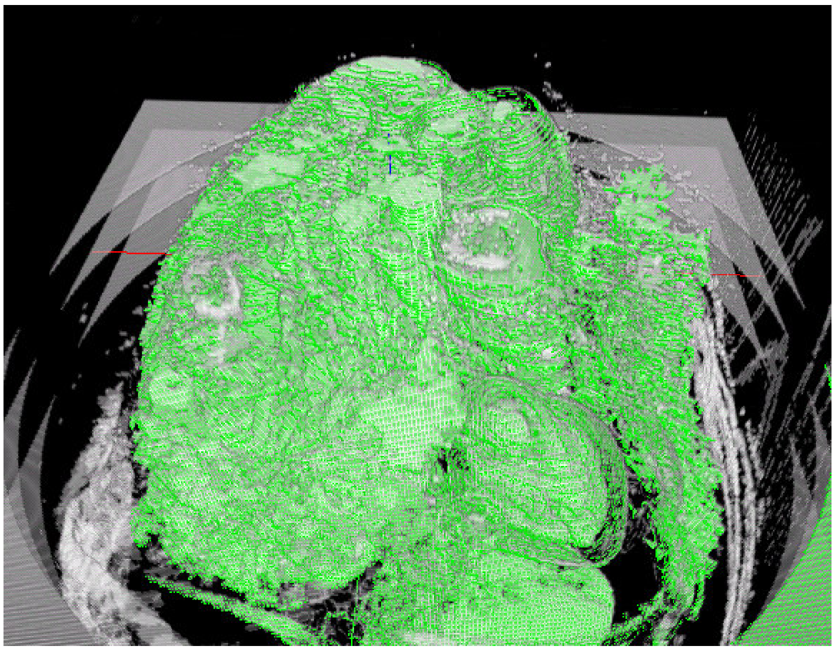
Together these plugins provide for large possibilities for post-processing. However, it takes some time get an intuitive feeling for how they work, since everything is processed in 3D.

4.3 Using Dr. Jekyll for manual post-processing

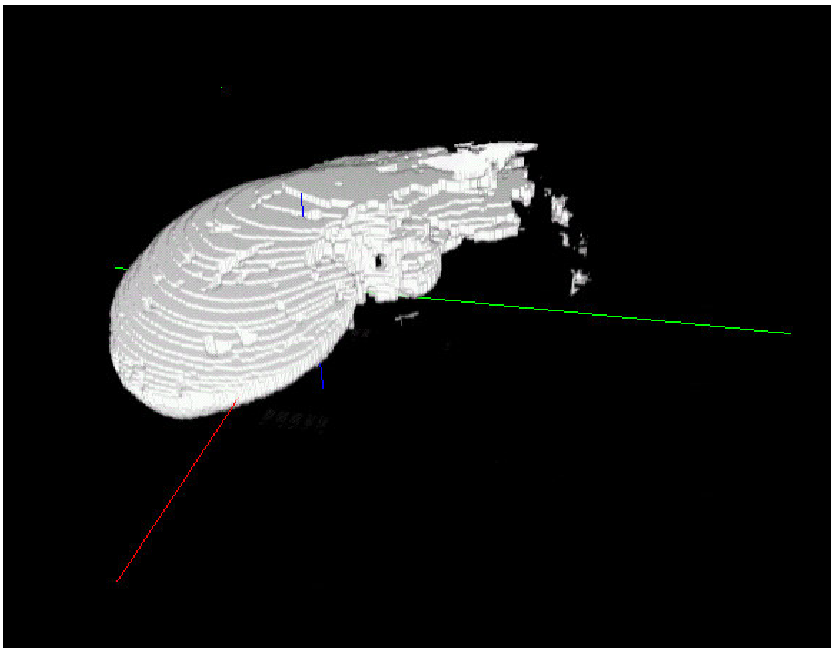
We have not, due to limited resources, conducted any empirical studies to assess if the *Dr. Jekyll* application allows for better and faster post-processing of segmented volumes. However, based on user feedback and our own experiences, the applications fulfills it goals, and it is possible to effectively post-process and classify segmented images using it.

Figure 4.3 shows a typical input image to *Dr. Jekyll*. The “input” image is very chaotic, and everything is connected to everything else. After a few minutes of massaging in *Dr. Jekyll*, the liver component shown in the bottom image was extracted. Doing the same in a 2D image manipulating tool like Gimp or Adobe Photoshop would take considerably longer.

The interactivity of the application is acceptable, but performance drops dramatically if the computer running the program has too little memory.



(a) Original segmented dataset. The highlighting shows the very large liver component.



(b) The extracted liver component

Figure 4.3: The upper image show a typical "input" image to *Dr. Jekyll*. It is very chaotic and more or less everything is connected to everything else. The bottom image shows the extracted liver component, it was made by applying several morphological operators to a selection of components, and manually relabeling those. The entire processing took less than five minutes in *Dr. Jekyll*.

Chapter 5

Conclusions and further work

We have spent one and a half year designing and developing the *Dr. Jekyll* application, and it is a useful tool for manual post-processing of volumetric images. Until “perfect” automatic segmentation algorithms arise, it will be necessary to refine segmentations, and manual post-processing is one way to accomplish this.

The time we have spent developing *Dr. Jekyll* has given us intimate knowledge about the tools we have used. It is time to evaluate if the chosen selection of languages and libraries were sound.

Thanks to its open source license it is possible for others to follow where we have left off. There is certainly more work to do within all the fields of computer science we have touched upon during this project and we hope to be able to contribute to some of them. Possible directions for future work will end this report.

5.1 At a computational threshold?

From the advent of computer hardware and until today, the speed and efficiency has been increasing at a tremendous rate. An application like *Dr. Jekyll*, dealing interactively with large 3D images would not have been possible just a few years ago, even on expensive “big-iron” machines.

Specially cheap 3D visualizations on commodity hardware opens up new and exciting possibilities, but the processing power and memory capability which just recently has become cheaply available allows for manipulating real-world datasets in new ways.

A clear indicator of this is the number of applications and projects doing related work to our own which have turned up during the period we worked on *Dr. Jekyll*. (See [17] for a list of some of these.)

Datasets will continue to increase, but so will processor and graphics performance, it will be very interesting to see what is possible just a few years from now.

However, better hardware also opens up new possibilities besides running old algorithms "larger" and "faster". New programming paradigms may arise, some might aim to hide complexity, like using scripting languages to avoid the code-compile-test cycle, others might expose more of it, as CPU's becomes more vulnerable to pipeline stalls and cache misses.

When implementing speed critical code, it will always be necessary to operate at several abstraction levels. Understanding how the effects of each abstraction layer (from choosing the best algorithm to the compiler optimization setting) affect the end product will always be important. However, running time is not everything. The time spent developing must weighted up at the importance of running speed of application and the cost of maintaining the software.

5.2 Did we accomplish the technical design goals?

A summary of how well we succeed in reaching the technical design goals defined in section 2.1 is given below:

➥ The interactive performance of the application is acceptable, even for large images, however, as images grow larger, calculating the connected components analysis requires several seconds, giving the illusion that the application has frozen. Computers running the application definitively needs large amounts of memory.

➥ We have succeeded in avoiding to hard code limitations on the size of the datasets. We have extensively used templates to avoid hard coding the types of data into the algorithms of the application. However, to overcome Qt's limitation with templates, is has been necessary to cast between types for communications via the signals-and-slots mechanism.

➥ We have not had access to any 64-bit computers, so we have not been able to test if the application can utilize a 64-bit CPU. Neither have we had access to the MS Windows version of Qt, so the portability of *Dr. Jekyll* to MS Windows is unknown. We do not however know of any reason that is should *not* fulfill these goals either.

➥ All the existing plugins of *Dr. Jekyll* uses the same interface for communicating with the rest of the application. New plugins should not need to alter any other parts of the program, and we believe the present interface to be sound.

However, no person outside our team has written any plugins, so if it is possible to write plugins without internal knowledge of the program is unknown.

➥ We have successfully combined several libraries into the application, providing us with a lot of functionality which we could not have produced ourselves. Combining libraries has not posed any technical problems, but a careful study of the licenses of each library was necessary to finalize the license of the application

5.3 How wise was the selection of languages and libraries?

Section 2.2 contained an overview of why the different libraries, and the C++ language was chosen as the tools for implementing *Dr. Jekyll*.

This section discusses the strengths and weaknesses that was found when using them extensively for more than a year. Each library is given a “verdict” which, which basically answers the question, would we use the same library again for a new project?

The following text focuses on how well suited the libraries were for implementing the algorithms. Not every library is covered here, but the companion work by Gleditsch[17] contains discussion of those left out.

5.3.1 Language

The choice of using C++ as the only implementation language has been a subject of debate throughout the entire implementation process. C++ is a very complex language, something which is both a blessing and a curse. Its a blessing since it is possible to implement nearly every programming paradigm in the language. Its a curse, since as the level of knowledge about C++ increases, you constantly want to rewrite old code to take advantage of newer and more elegant methods. However, advanced C++ might be unreadable by other developers, since they don’t understand the mechanisms involved.

Bjarne Stroustrup claims that becoming comfortable with all the major features of C++ takes from one to two years[51]. This estimates assumes basic programming skills from beforehand, but not detailed knowledge about the intrinsics of C++. This claim coincides with our experience from writing *Dr. Jekyll*.

The template mechanism of C++ has been used quite extensively. Basically intended to primarily allow for type independent containers, many new uses has been found for it, and the C++ community does not yet understands its full power.

It very easy to be fascinated by the possibilities of writing truly generic algorithms. If however generic programming provides any real world benefits, like shorter development times or less bugs is dubious. Advocates of the *Extreme Programming* methodology of programming[5] which has gained popularity recently, calls for the simplest design that gets the job done, and to do not try to solve tomorrows problems. (However it can be argued that extreme programming does not apply to project as small as *Dr. Jekyll*.)

Our experience is that the classes which were designed with genericity in mind could more easily met new unexpected requirements. This coincides with the claim from [41] where empirical studies show that a generic solution is easier to maintain in the long run.

It has previously (section 2.2.1) been hinted at the possibility of using a scripting language to develop the GUI, and just use C++ or even classical C or Fortran for the speed critical algorithms.

For realistically sized datasets, writing the entire application in a scripting language is not an option, since both memory usage and algorithm running time would be considerably higher than for a compiled language.

If starting anew with the project, such a “dual-language” approach would have evaluated to a larger degree before committing to the solution. That said, C++ has a proven track record for large projects, has mature compilers and tools and it is very well documented. We do not rule out the option that we would end up using C++ again for a similar project.

5.3.2 Mixing libraries

It is solely thanks to the number of high-quality and freely available on the Internet it has been possible to write an application as complex as *Dr. Jekyll*. Even if all the libraries we have used were mature, and well thought out, they are designed with different philosophies in mind.

While different philosophies in the libraries makes the resulting source code less aesthetically pleasing than it could have been, it has not been a problem in practice. Using several libraries results in more time poking through documentation, but it is still a lot faster than implementing the same functionality from scratch. In the future we would certainly like to continue using libraries, if the licenses are compatible. However, verifying if the licenses are compatible is just as important as verifying if the libraries are technically compatible.

5.3.3 Qt

The choice of using Qt for GUI and message passing (see section 2.2.2) turned out to be more problematic than was originally foreseen. The problem is the usage of a the `moc` preprocessor to support the signals and slots mechanism, but as stated earlier, a class which has signals and slots can not be a template class.

To a certain degree this can be worked around by having a non-template base class with signals and slots, and let the template class inherit those, something which leads to unnatural and poorly designed class hierarchies. However this strategy can only be used once in each inheritance hierarchy since all subclasses of a template class must also be template classes.

Another annoyance with Qt it the use of to broad datatypes in some classes. Methods typically return singed datatypes, when the unsigned variant is enough. (Ie. for the height and width of widgets, the number of children for a node in a tree etc.) A study by Prechelt[40] concludes that libraries should strive to use as strict type checking as possible the reduce the number of possible bugs.

As experience was gained using the template mechanism of C++ it became apparent that templates are a powerful mechanism, and it would be very interesting to have templated signals and slots as well as templated classes.

However as a library for writing GUI components Qt is as good as they get. The documentation is excellent, and there is a good selection widgets available. More than enough for the relatively simple dialogs which are present in *Dr. Jekyll*.

When developing a cross platform application, it is the only library promising source code compatibility between various platforms.

Before using Qt again for a new project, we would evaluate if cross-platform compatibility between MS Windows, MacOS X and UNIX platforms is an objective. If it important, Qt is really the only way, even though Qt also has it shortcomings.

We would however hesitate to use the signal-and-slots mechanism for inter process communications, since it lacks support for templates. The ideas of signals and slots is excellent, but a library like `GNU libsigc++`[74] manages to provide this functionality, without restricting its users to use a subset of the language.

5.3.4 Blitz++

The `Blitz++` library which was used to store the image data, has been a very pleasant library to work with. Is has been very stable, and the documentation is excellent. We have also received helpful support on the Blitz mailing lists from time to time. We have also submitted a small patch back to the Blitz project,

A feature we would like to highlight, is its excellent support for generating slices and subvolumes of the data in the container. It is very easy to extract a region of interest from the data, a nice feature was that is was possible to decide if the subvolume should keep the coordinate system of the original volume or if it should have its own. This made it very easy to map a boundingbox out of an image, process it and map it back into main image.

For a new project we would not hesitate to use Blitz++ again, at least until a comparable container becomes a part of the official C++ standard.

5.3.5 OpenGL and Open Inventor

For the visualizations, we ended up using mostly raw OpenGL. OpenGL delivers fast performance on different hardware. Being a de facto industry standard, documentation was excellent, both in the official texts[66, 25] and on various Internet sites.

However the abstraction level when writing OpenGL code is very low, directly dealing with the lowermost geometric primitives. When programming OpenGL one must keep in mind that one is programming a state machine, which is not directly tied to ones applications program flow.

Open Inventor ended up only being used for the volume rendering, but thanks to its intuitive scene-graph model it was extremely easy to create the wanted functionality. The entire volume rendering functionality was written in a few hours, demonstrating how powerful the scene-graph paradigm is.

Again the documentation, both in form of the official documents[63, 64] and the online documentation available from Coin's homepage[73] was excellent. So was the support on the Coin mailing list.

We would definitely use both OpenGL and Open Inventor again in a new project. Possibly relying even more on Open Inventors scene graph, and only resort to low-level OpenGL if the performance of using Open Inventor is not good enough.

5.4 Further work

This work has touched upon several different fields within computer science. Possible directions for future work are therefore numerous. The application itself, is of course never completely finished, and if time permits it would be very interesting to extend it further. This section aims to summarize directions the author finds the most interesting.

5.4.1 Empirical studies to assess the benefi ts of manual post-processing

While this work has been very technical, the goal has been to provide an application which makes it easier to refine segmentations of volumes. It would be very interesting to assess the real world benefits of *Dr. Jekyll*. Both the time used, and the quality of the resulting classification should be measured.

Such a study would have to decide what the target group of the program is. Is it technicians, who are used to using different software, or is it professionals in the field of study where the images arise?

5.4.2 Applications of mathematical morphology in geometric modelling

Mathematical morphology is a relatively unknown field, yet it provides intuitive operators for manipulating shapes. Assessing it usefulness in combinations with other fields would be interesting.

A candidate for such work could be to see if erosion and opening can be used for decimating (optimizing) the surface representations of geometric objects. Conversely, closing can have its uses in subdivision (smoothing) algorithms.

As a method for extracting the surface of objects they also show promise. Using morphology for gray scale images, they could possible provide an alternative to using "marching" for extracting iso surfaces as well. If the neighborhood information calculated by the morphologic operator could be used to directly extract a triangle representation of the surface, they would be a very attractive tool on some settings.

5.4.3 Other datastructures and representations

Due to their size, images requires a lot of memory. It would be interesting to experiment with other image representations that reduce the memory requirements.

One possibility is to keep the image compressed in memory, but still keep the representation as being based on pixels. Such a container has been prototyped as part of the *Dr. Jekyll* project, based on simple run length encoding. However its running time $\mathcal{O}(\log n)$ for both lookups and alterations, something which was to slow for interactive usage.

Using other representation methods, possibly based on OCTrees would be interesting as well.

5.4.4 Patterns for algorithms

Implementing algorithms in *Dr. Jekyll* has demonstrated how different language constructs can results to wildly varying running times for algorithms.

A complete survey on how different language construct, *with the same formal running time* affect the running time of an *implementation* of an algorithm. Such a survey is complicated by the fact that various compilers might optimize the code in various ways; constructs which are fast on on compiler, might not be fast on others. Also different memory technologies and caching strategies must be considered.

Such an effort would probably benefit from the findings of the Atlas[70, 65] and Tune[37] projects, for finding cache efficient implementations of algorithms for linear algebra. However, caching is not the only factor which must be considered, how to best utilize the long pipelines and semi-parallelism in a modern CPU must also be explored.

One possible way for such an effort could be to identify common building blocks which are common to different algorithms. A kind of “patterns” for algorithms. These patterns could be implemented generically, and possibly optimized empirically for different compilers and computer architectures.

5.5 Further work in Dr. Jekyll

An application the size of *Dr. Jekyll* will always have potential for improvements, and is in that sense never finished. If possibilities arise to do more work on it, the following lists some good starting points for further work. Some of them are good candidates for other project work by students, other are more of an “engineering” exercise.

➥ Threading: The largest problem of *Dr. Jekyll* as it stands today, is that the application apparently freezes when the connected components analysis is calculated on large datasets. Making the application threaded would make the user experience much better. Qt provides cross platform classes for threading, and locking of the image data is already implemented, so this should not be an enormous task. However multi-threaded applications are harder to debug than single threaded ones, and nasty deadlock situations might occur.

➥ 3D editing tools: To be able to edit the surface of the components within an image interactively would be a great plus. The most intuitive way to do the actual editing would probably be to have a parametric representation of the surface and modify the control points. However calculating such a representation for objects of arbitrary shape and topology is very difficult.

An interesting approach to generate such a parametric representation, would be to first generate a triangle representation of the component surface, and then simplify the triangulation as much as possible[6]. Thereafter splines could be calculated over the simplified triangulation, and these splines could be interactively edited by hand.

➥ More visualizations: The current implementation is lacking a good surface visualization of the images. One possibility, which would make it relatively easy to add new visualizations is to use the algorithms already present in the VTK[44, 82] library. VTK itself does not provide any mechanism for integrating the rendering with Qt. The geometry which VTK outputs can however be used by Coin, and Coin provides excellent incorporation with Qt.

This functionality has already been prototyped, however it is not included in *Dr. Jekyll* today.

➥ Editor for structural elements: Since the structural element is decoupled from the algorithms which uses them, it would be nice to provide a GUI editor for adding more SE's. Implementing such an editor is not an enormous task, and would make the morphological operators more useful, since custom SE's could be created for datasets which special peculiarities.

➥ Compress undo information: It has been stressed that the images which *Dr. Jekyll* process are potentially very large, and uses large amount of memory. While it probably would incur a to high runtime penalty to use any compression of the image dataset which is worked on. The application also stores undo information in a uncompressed format. These undo data are a prime candidate for the usage of compression. Implementing such functionality should be local to just one class of the program, and should therefore be quite easy to incorporate. One must however find a good compression algorithm, which doesn't use to much time to uncompress.

➥ Port to MS Windows: One of the main design goals of *Dr. Jekyll* has been to write portable, cross platform and standards compliant C++. However, since the Qt library is not available under the GPL license on MS Windows, it has not been possible for us to test it there. With the Windows version of Qt (and a recent compiler) it should in theory compile out of the box. Realistically, there has been small glitches in the implementation, so some updates of the code is necessary. Yet, it should not be more than a days work to get it up and running on MS Windows.

- Connection to Olena: The Olena[9] library provides many image processing algorithms written in the same spirit as the morphological operators in *Dr. Jekyll*. Incorporating them should be relatively easy, and it would add segmentation functionality to *Dr. Jekyll*.

Annexes

Appendix A

Benchmarks

A few benchmarks were run to test various language constructs. They are summarized here. All the timings are in seconds, and were gathered using the standard Unix `clock()` function. The numbers in the tables below are averaged running times for 10 consecutive calls to the connected components algorithm.

All the benchmarks were run with the GCC C++ compiler, version 3.2 (this is the stock compiler on RedHat 8). The compiler was set to use aggressive optimizations (-O3), as well as using all extensions the target platform supports.

The timings were run on two different machines, using different CPU architectures, running at different speeds.

- Bardou is a dual 1.2GHz AMD Athlon MP processor. Each CPU is equipped with 256KB of cache memory. It has a front side bus speed at 266MHz. The code for Bardou was compiled with the following compiler settings: `-O3 -march=athlon-mp -mfpmath=sse -mmmx -msse -m3dnow`

- Belves is a 2.54 GHz Intel Pentium IV. The CPU is equipped with 512KB of cache memory. It has a front side bus speed of 533MHz. For Belves, the compiler settings were: `-O3 -march=pentium4 -mfpmath=sse -mmmx -msse -msse2`

A.1 Traversal direction for connected components analysis

The ordering of the loops in the Connected Components algorithm were run for both X-major and Z-major on two machines. The memory is stored consecutively along the Z-axis, which is reflected in the measurements in table A.1. Bardou has

Dataset size	Bardou $X-Y-Z$	Bardou $Z-Y-X$	Belves $X-Y-Z$	Belves $Z-Y-X$
$512^2 \cdot 59$	16.255	21.108	8.87	14.21
64^3	0.295	0.324	0.15	0.184

Table A.1: Timings (in seconds) for the connected components algorithm when changing the order of traversal. Using the Z direction in the innermost loop traverses the memory consecutively, leading to faster execution times.

a difference of only 29% for the large dataset, while Belves has a staggering 60% difference. This is not so surprising, since the Intel computer has a larger cache, and will have more cache hits in the "normal" situation. For the small dataset, the difference is not that large.

A.2 Object allocation inside tight loops

This test is based on the code illustrated in figure 4.1. The "New object" row (in table A.2) shows the timings when a new STL vector is allocated for each iteration of the loop. The "Object resetting" row is when the vector is reseted by a call to `clear()`. The algorithm is again the connected components algorithm, using the large dataset also used above.

Belves uses 8.75 times longer when the object has to be allocated for each iteration. Bardou on the other hand uses only 4.19 times. However, such factors alone show the importance of knowing what language constructs are expensive to use. Even though both object resetting and allocation is an $\mathcal{O}(1)$ operation theoretically.

It is interesting to notice that Belves uses *longer* time than Bardou when the object is allocated inside the loop, even if the CPU runs at more than a GHz more. The reason for this is not fully understood, but a educated guess is that the Pentium IV's extremely long pipeline must be flushed each time new memory is allocated (since object allocation requires the operating system to intervent). The Athlons

Dataset size	Bardou	Belves
New object	68.251	77.623
Object resetting	16.255	8.87

Table A.2: Timings for the connected components algorithm when the two helper vectors are allocated anew for each iteration, vs. the case where they are only reseted.

pipeline is not that long, so the cost of a pipeline flush is not that extreme. The recently announced AMD Opteron CPU, which is a hybrid 32 and 64 bit processor has a longer pipeline, and is probably suspect to the same weakness as the Pentium IV.

Bibliography

[1] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A. K. Peters, second edition edition, 2002.

[2] Andrei Alexandrescu. *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley, 2001.

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

[4] Chandrajit Bajaj, Insung Ihm, and Sanghun Park. 3d rgb image compression for interactive applications. *ACM Transactions on Graphics (TOG)*, 20(1):10–38, 2001.

[5] John Brewer. Extreme programming FAQ. `http://www.jera.com/techinfo/xpfaq.html`, 2001.

[6] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Efficient decimation fo complex triangle meshes. Technical Report 3/98, Computer Graphics Group at University Erlangen-Nurnberg, 1998.

[7] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V. W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 10(5):458–494, 1997.

[8] Matthias Kalle Dalheimer. Design patterns in Qt. *The O'Reilly Network*, October 2002. `http://www.onlamp.com/pub/a/onlamp/2002/01/10/designqt.html`.

[9] Jèrôme Darbon, Thierry Gèraud, and Alexadnre Duret-Lutz. Generic implementation of morphological image operators. In *Proceedings of the 7th International Symposium on Mathematical Morphology*, 2002.

[10] R. Deklerck, A. Salomie, and J. Cornelis. An editor for 3d medical volume images. In *TASK-Quarterly*, volume 1, pages 155–162, October 1997.

[11] M. Van Droogenbroeck and H. Talbot. Fast computation of morphological operations with arbitrary structuring elements. *Pattern Recognition Letters*, 17(14):1451–1560, 1996.

[12] George Eckel and Ken Jones. *OpenGL Performer Programmers Guide*. Sgi, 002 edition, November 2000.

[13] ECMA. *ECMA-334 C# Language Specification*, 2001.

[14] James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer graphics: principles and practice*. The Systems Programming Series. Addison-Wesley Publishing Company, 2nd. edition, 1996.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, October 1994.

[16] Ronald Garcia. The boost multidimensional array library. `http://www.boost.org/libs/multi_array/doc/index.html`.

[17] Kristoffer Gleditsch. Interactive manipulation of three-dimensional images. Cand. scient. thesis, Department of informatics, University of Oslo, Norway, May 2003.

[18] Josh Grant. Projects – marching cubes. http://www.wheatchex.com/, November 2001.

[19] John Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[20] John L. Hennesy, David Goldberg, and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[21] Intel Corporation. *IA-32 Intel Architechture Software Developer's Manual*.

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

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

[24] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[25] Renate Kempf and Chris Frazier, editors. *OpenGL Reference Manual*. Addison-Wesley, second edition, 1996.

[26] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[27] Hans Petter Langtangen. *Computational Partial Differential Equations*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, 1999.

[28] Hans Petter Langtangen. *Scripting Tools for Scientific Computing*. Textbook in Computational Science and Engineering. Springer-Verlag. In preparation.

[29] Barthold Lictenbelt, Randy Crane, and Shaz Naqvi. *Introduction to Volume Rendering*. Hewlett-Packard professional books. Prentice Hall, 1998.

[30] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34. ACM Press, 1987.

[31] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[32] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture: A hypertext history. *Intel Technology Journal*, 6(1), February 2002.

[33] G. Matheron. *Elèments pour une thèorie des milieux poreux*. Masson, 1967.

[34] Scott Meyers. *More Effective C++*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1996.

[35] Scott Meyers. *Effective C++*. Addison-Wesley Professional Computing Series. Addison-Wesley, second edition, 1997.

[36] Scott Meyers. *Effective STL*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2001.

[37] Univerisity of North Carolina. Tune – mathematical models, transformations, and system support for memory-friendly programming. `http://www.cs.unc.edu/Research/TUNE/`.

[38] Jung-Me Park, Carl G. Looney, and Hui-Chan Chen. Fast connected component labeling algorithm using a divide and conquer techniqu. In *Proceedings of the ISCA 15th International Conference on Computers and their Applications*, 2000.

[39] John L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

[40] Lutz Prechelt and Walter F. Tichy. A controlled experiment to asses the benefits of procedure argument type checking. *IEEE Transactions on Software Engineering*, 24(4), April 1998.

[41] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. A controlled experiment in maintainance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 21(12), December 2001.

[42] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[43] C Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware*. PhD thesis, University of Erlangen-Nurnberg, 2002.

[44] Will Schroeder, Ken Martin, and Bill Lorensen, editors. *The Visualization Toolkit*. Prentice Hall, 2nd. edition, 1997.

[45] Sandeep Sen and Siddhartha Chaterjee. Towards a theory of cache-efficient algorithms. Available at `ftp://ftp.cs.unc.edu/pub/users/sc/papers/soda00.pdf`.

[46] Sgi. Sgi white paper – opengl performer. `http://www.sgi.com/software/performer/whitepapers.html`.

[47] Pierre Soille. *Morphological Image Analysis*. Springer-Verlag, 1998.

[48] Bjarne Stroustrup. Why C++ isn't just an object-oriented programming language. In *OOPS Messenger*, volume Addendum to OOPSLA'95 Proceedings, October 1995.

[49] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[50] Bjarne Stroustrup. An overview of the C++ programming language. In Saba Zamir, editor, *Handbook of Object Oriented Technology*. CRC Press, 1999.

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

[52] Herb Sutter. *Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 1999.

[53] Herb Sutter. *More Exceptional C++*. C++ In-Depth Series. Addison-Wesley, 2002.

[54] G. Székely and G. Gerig. Model-based segmentation of radiological images. *Künstliche Intelligenz*, (3):18–23, 2000.

[55] The Open Source Initiative. The open source definition. `http://www.opensource.org/docs/definition.php`.

[56] The Open Source Initiative. The open source initiatives homepage. `http://www.opensource.org/`.

[57] Lennart Thurfjell, Ewert Bengtsson, and Bo Nordin. A new three-dimensional connected components labeling algorithm with simultaneous object feature extraction capability. *CVGIP: Graphical Models and Image Processing*, 54(4):357–364, 1992.

[58] Trolltech. Using the meta object compiler. `http://doc.trolltech.com/3.1/moc.html`.

[59] Todd L. Veldhuizen. Scientific computing: C++ versus Fortran: C++ has more than caught up. *Dr. Dobb's Journal of Software Tools*, 22(11):34, 36–38, 91, November 1997.

[60] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001.

[61] John Vlissides and Andrei Alexandrescu. To code or not to code, part i. *C++ Report*, March 2000.

[62] John Vlissides and Andrei Alexandrescu. To code or not to code, part ii. *C++ Report*, June 2000.

[63] Josie Wernecke. *The Inventor Mentor – Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.

[64] Josie Wernecke. *The Inventor Toolmaker – Extending Open Inventor, Release 2*. Addison-Wesley, 1994.

[65] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimization of software and the atlas project. Technical report, Netlib Repository, `http://www.netlib.org/lapack/lawns/lawn147.ps`, September 2000.

[66] Mason Woo, Jackie Neider, Tom Davis, and Dace Shreiner. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1997.

[67] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

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

[69] 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.

[70] Atlas homepage. `http://math-atlas.sourceforge.net/`.

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

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

[73] Coin 3d homepage. `http://www.coin3d.org/`.

[74] Libsigc++ homepage. `http://libsigc.sourceforge.net/`.

[75] Magick++. `http://www.imagemagick.org/www/Magick++/`.

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

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

[78] Sourceforge.net. `http://www.sourceforge.net/`.

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

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

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

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

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