A Client/Server Framework for Interactive Remote 3D Visualization of Histological Data

THESIS

in partial fulfilment of the requirements
for the degree of Master of Science
presented at Delft University of Technology,
Faculty of Electrical Engineering, Mathematics, and Computer Science,
department of Mediamatics, section Computer Graphics and CAD/CAM

by
Jeroen E. Verschuur
2009

Supervised by:
Dr. Charl Botha
Ir. Frits Post

TUDelft Delft University of Technology

PREFACE

This thesis is the report of my work on designing and developing a visualization framework for histological data aimed at research- and education applications. It sure took me a while to finish but finally my work is done! My work ends here with the completion and submission of this thesis but I really hope that the framework will be used by many people and be further improved and extended in the future, as this would give me the satisfaction of knowing that my work was more than ’just’ an academic assignment.

I would like to use this opportunity to thank several people that were important for my graduation. First I would like to thank Charl Botha for supervising me throughout this project and for finding time in his busy schedule. You should also know that he is gifted with a contagious enthusiasm: at certain times you may come to him for help on some specific problem and you usually go home with not only an answer to your problem but also with an irrepressible urge to explore a dozen additional, very interesting, ideas.

I also want to thank Huib Simonsz for allowing me to contribute to the Visible Orbit project and for providing me with a medical perspective on this project, based on his extensive knowledge of the medical world in general and the human eye in particular. In one of the meetings we had he also introduced me to Ben Willekens, who allowed me to look at the actual microscopic sections through a microscope at the Netherlands Institute for Neurosciences in Amsterdam and who spent countless hours digitizing the microscopic sections of the Orbita Collection. Even after his retirement he continued this laborious work, which to me is a sign of utmost dedication.

Thanks to Erik Jansen for carefully reading my pre-final thesis and providing me with valuable suggestions to improve the content and structure of my thesis. Thanks also to Frits Post and Alexandru Iosup, for joining the committee and thereby accepting the large amount of work involved.

I would like to thank my family, especially my parents, for their continued support in the years that I attended university and for regularly showing their interest in my project. Finally, I want to thank Myrthe for her patience during this longer-than-expected graduation and, more importantly, for her presence in my life which certainly makes my life more valuable.

Jeroen Verschuur

November 2009
Hoogvliet, The Netherlands

ABSTRACT

Title of thesis:	A Client/Server Framework for Interactive Remote 3D Visualization of Histological Data
Author:	Jeroen E. Verschuur
Date:	November 2009
Committee:	Prof. Dr. Ir. F.W. Jansen Dr. Ir. A. Iosup Prof. Dr. H.J. Simonsz Ir. F.H. Post Dr. C.P. Botha

The amount of data involved in medical volumes, especially in the case of volumes consisting of stacks of histological sections, tends to be very large. Despite rapid advancements in computer processing power and storage- and memory capacity in the past years these volumes are simply too large to be transferred to and visualized on regular workstations. At the same time Internet connectivity is becoming common and connections are getting faster every day. In this situation a client-server remote visualization can be used to hand off visualizations to (clusters of) dedicated, well-equipped, servers which can do the heavy work and send the result, or a part thereof, back to the client when ready.

In this project we developed a client-server framework for remote visualization of histological data. The framework is designed to be scalable and effectively utilize hardware resources from multiple servers for each visualization pipeline. The framework uses the concept of strategies to assign resources to each visualization and to distribute these resources over the available hardware according to predefined rules.

To manage the available data we designed a data scheme in which multiple objects belonging to a certain dataset, we will define these objects as modalities, can be stored in a relational database. A key feature of our solution is that the scheme allows appropriate visualizations of the modalities to be stored along with the data in the same relational database.

Additionally, as part of this project, we developed a reference client application which is able to use the framework to do remote visualization. This application was purposely designed to work on many platforms without the need for state-of-the-art hardware or software. The application currently offers support for visualizations that require viewing of images, viewing of planes extracted from volumes and viewing of three-dimensional scenes under control by a camera.

Both the framework and the reference client application are designed to be extensible and will be made available to the public domain so that the software can be used freely and adapted to future demands whenever needed.

Contents

Chapter 1

Introduction

Recently, a unique and valuable collection of histological sections (cross sections of the human orbit with thickness up to approximately 100 μm) has been discovered. These cross sections are now being digitized at high resolution and used for 3D reconstruction. To visually disclose this data for research and education, a standard web application would not suffice as it can not deal with the large amounts of data and processing that is needed to display 3D volumes.

In this thesis we will describe an architecture and design of a framework that allows remote rendering of the volumes on a server using parallel volume rendering. The framework will also provide the means for remote viewing of high-resolution slices that can be extracted from the volumes and which may be augmented with annotations if available. Eventually, all software written for this framework will be made available as open-source software to the public domain.

1.1 Visualization of Histological Data

While MRI [1] and CT [2], both in-vivo [3] techiques, are often used to examine biological tissue, the resolutions and color information they offer is rather limited. Despite these limitations, MRI and CT techniques are often very useful in helping radiologists determine the cause of medical conditions.

Our project focuses on visualizing histological data. For research purposes the technique of histological sectioning, in which tissue will be physically sliced, allows real-color (RGB) acquisitions and provides much more detail with resolutions that will only be limited by either the cutting process or by the optical equipment used for digitization. With special equipment resolutions of over 100000 dots-per-inch (dpi) can be reached (for example the BrainMaps project [1] currently offers samples at 55000 dpi, or 0.46 μm/pixel). More information on the process of histological sectioning is provided below. Digitization of histological sections at these resolutions obviously results in huge amounts of data. Because processing power and storage capacity have increased rapidly over time, better visualizations of ever-increasing amounts of data become feasible.

Within the field of computer graphics a substantial amount of research is done on medical visualization, a field of research that has the potential of supporting physicians and researchers in medicine worldwide. Our research on a framework capable of visualizing these large volumes, together with technological advancement, could open the way for use of histological volumes in research- and education environments.

[1]Magnetic Resonance Imaging, technique visualizing internal structures by measuring magnetic properties of hydrogen atoms. Especially useful for soft-tissue examination.

[2]Computed (Axial) Tomography, technique using a large series of X-Ray images with a single axis-of-rotation to visualize internal structures

[3]Literally translated ”within the living” and referring to experiments on living organisms as opposed to experiments on dead organisms or on tissue obtained from a biopsy.

1.2 Dataset Overview

Throughout this thesis two data collections are discussed, both of which are described here to justify their important role in this project. The datasets share the fact that they are (collections of) histological volumes, but they each have properties making them interesting subjects for this work of research. These histological volumes are digitizations of so-called histological (often microscopic) sections. These sections of organic material are usually obtained in five steps, explained shortly (see [7] and [8]):

1. Fixation: stop life-processes quickly and prevent deformation
2. Embedding: prepare the specimen for the sectioning process
3. Sectioning: the actual cutting of the specimen
4. Staining: colour the section with a ’dye’ or ’stain’ to enhance contrast between different types of tissue
5. Mounting: preserve the section and prepare it for microscope inspection and/or digitization

In several steps in the sectioning process decisions need to be made with regard to the technique applied in that stage, and these decisions will be based on the outcome that is aimed for with a particular type of research. For the embedding stage for example, several alternatives are available. Embedding the specimen in paraffin or cellulose and also freezing the specimen are often used techniques for this stage. In the staining process different dyes can be used to highlight different types of tissue in the sections. Also, one can choose to have the sections stained with a different dye for every other section (or so), to see local differences. The combination of the techniques applied will obviously yield different results having different properties, as the two collections described below will clearly show.

1.2.1 The Orbita Collection

Recently a collection of microscopic sections was discovered at the Netherlands Ophthalmic Research Institute (which has become part of the Netherlands Institute for Neuroscience) and the Department of Anatomy and Embryology of the Academic Medical Centre (AMC) in Amsterdam. These sections were obtained between 1972 and 1986 as part of several studies of the late professors J.A. Los, L. Koornneef, Dr. M.P. Bergen and Dr. A.B. de Haan. The complete collection (from here on called ”The Orbita Collection”) consists of approximately 3000 sections, from five human adult orbits, and 30 foetal heads and bodies [9].

In the acquisition process several different stains were applied to the sections. However, in the past thirty to forty years, these stains have started to fade and in a few decades not much will be left of them, rendering the collection useless.

A consortium consisting of the NIN, AMC and TU Delft has agreed that this valuable and unique collection should be preserved for further research and therefore the digitization of the sections has started a few years ago. Obviously, digitizing 3000 sections is an enormous amount of work, most of which is currently carried out by Ben Willekens at the Netherlands Institute for Neuroscience (NIN). Details about this project can be found at the project’s website [10].

1.2.1.1 Digitization

As mentioned above, the collection consists of approximately 3000 sections, cut at various thicknesses. They have been scanned at a resolution of 2500 dpi. The images are currently stored in the PNG format, which uses a lossless compression method to achieve relatively high compression rates as well as high quality. The final size of an image depends on the physical size of the section and the compressibility of the resulting image, but files up to 100MB are no exception. Actually, the largest slice available (pixel-wise) is approximately 12000x7000 pixels. With 200-700 sections per subject it is easy to see that for a single specimen multiple gigabytes[4] of data are available.

[4]Throughout this thesis we will refer to either gigabytes (GB) or megabytes (MB), where 1GB = 1024MB and 1MB = 1024x1024 = 1048576 bytes of data.

1.2.1.2 Reconstruction

In the histological sectioning process, slices of tissue have been fixed between two transparent objects (glass for example). However, the orientation of the tissue between the transparent objects is completely dependent upon the person fixating the section. Studying the randomly orientated sections is no problem, but stacking the sections to get a 3D model will not work well, as the sections are not aligned properly. Also, artefacts in the sections (such as small cuts, overlapping tissue, etc.) as well as the use of different stains (colourings) in the sectioning make it difficult to use the data.

To get a coherent 3D dataset of a subject, we will have to use techniques taking care of three dimensional reconstructions. Applicable methods for reconstruction have been studied in [11], and a suitable method for reconstruction of the sectional data in the Orbita collection has been proposed and implemented by Van Zwieten [8]. Prior to the actual reconstruction three pre-processing steps have to be taken: down sampling (to speed up the reconstruction), segmentation (to separate the object from the background, which speeds things up as a side effect), and a conversion from RGB data to scalar data (needed because the reconstruction and its metrics are based on scalar values). After that, an iterative process of image registration starts. In [8] this is defined as ”the search for a transformation that puts two images of the same scene, the reference and the floating image, in a common coordinate system such that all corresponding points are aligned”. So, the aim of process is to maximize the similarity between consecutive sections, resulting in a dataset that will (hopefully) closely resemble the original three dimensional specimen, or try to get as close as possible.

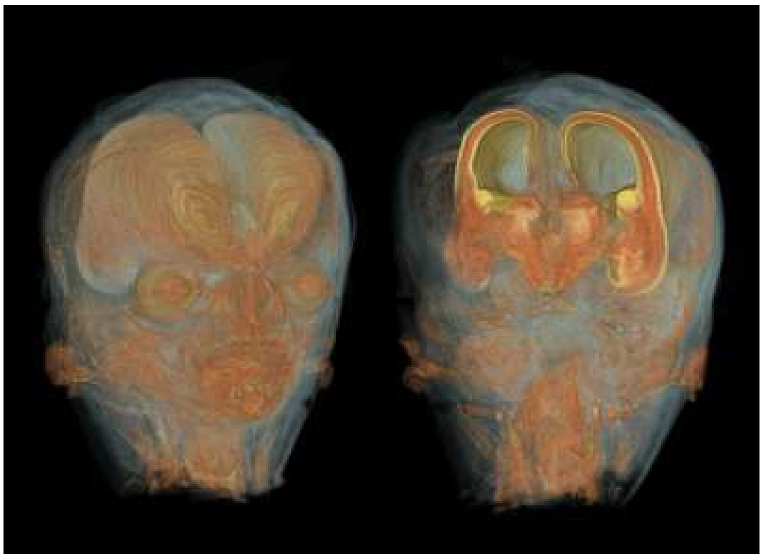


Figure 1.1: Rendering of an embryo from the Orbita collection after reconstruction from the digitized sections. Image courtesy of [8].

1.2.2 The Pelvic Dataset

The human pelvic dataset used as source of data for this project was acquired in 2007 as part of research by the Leids Universitair Medisch Centrum (LUMC) using a technique called frozen-cadaver histological sectioning, a technique also applied for the well-known Visible Human Project datasets of the National Library of Medicine (NLM).

1.2.2.1 Acquisition and digitization

The acquisition technique used in the pelvic dataset is fundamentally different from the technique used with the Orbita collection. First of all, the specimen was embedded using freezing, instead of treating the tissue with formaldehyde and embedding in cellulose. An important difference is that the obtained sections will preserve most of their real-color information instead of the color resulting after staining with some given dye. A second major difference with this technique is that the sections are not placed between thin sheets of glass like the Orbita sections (which were required for examination under a microscope). Instead of mounting the sections they have been photographed from above using a professional digital camera. The sections have been cut at a thickness of 25 μm and every third section has been photographed, yielding a slice-thickness of 75 μm.

In total 2052 images of 3008x1960 pixels are available for the dataset, and each image is just under 18 megabytes (3008x1960 pixels at 24 bits/pixel), stored in the lossless TIFF file format. What makes this dataset interesting (besides the fact that its color information is conserved) is that the sections are well-aligned. As a consequence of the acquisition process issues with alignment are already minimal. However, for the pelvic dataset two frameshifts have occurred, which have been properly documented and corrected manually after the acquisition.

1.3 Objective and Motivation

The objective of this project is to design and implement a framework for remote visualization of histological volumes. The system should be able to deal with the large amounts of data involved in histological volumes and should therefore offer a scalable visualization solution. The framework should also be able to efficiently utilize and combine resources from several computer systems in a distributed and parallel way, to accomplish its resource-intensive task. Supported visualization techniques for these large datasets should include Direct Volume Rendering (DVR), surface extraction and rendering, and Multi-Planar Reconstruction (MPR).

We aim to keep the data in a central location, for three main reasons. In many cases application-data can be distributed along with the application itself (for example on portable media like CD's or DVD's, or using an internet connection) but for the huge amount of data involved in histological volumes this is usually infeasible. In case this data would be distributed along with the application anyway every user would be forced to have a powerful computer system available to be able to process the data. A third reason is the delicate matter of protecting the data (data which is often very expensive), a client-server system layout provides better means for protection of the data.

To allow for remote visualization, the framework will be client/server-based and two separate components will be designed. A server component will be designed that is responsible for providing the functionality described above, where we will use the ParaView framework as parallel rendering backend to our server component. The framework should be able to perform user authentication and will therefore require some user management capabilities. In addition to the server component, a reference client application will be implemented (reference, in the sense that implementers of the framework are not limited to this reference client application). Clients will be able to run the provided visualizations from any location (allowed by the administrator of the system) and the aim is to create a client application requiring very little installation effort, allowing practically any user to run the application. The client application will support several elemental histological volume visualizations such as (arbitrary) plane reconstruction, with support for annotations, and volume rendering.

The idea is to design both the server and the client using open-source components and -protocols to allow further development of this framework after its initial release (after this thesis), and to enable others to easily create a custom client application to use the framework.

1.4 Contributions

The work described in this thesis makes the following contributions to visualization frameworks:

- We developed a fully-functional remote visualization framework that is designed to be highly-scalable and extensible and which can be deployed on a large number of currently common operating systems (Chapter 3). We built our framework around ParaView, which itself is built on the Visualization Toolkit (VTK) and offers a large collection of visualization algorithms. Specific aspects of a remote framework, such as work-distribution strategies, incremental result transfer and authorization were incorporated (Chapter 3 and 4). A reference client application was developed to allow use of the default visualization functionality (Chapter 2).

- To organize the available data we developed an elegant data encapsulation scheme (Section 3.5.3). Information on available datasets and possible relationships between them is captured in a relational database. Additionally the scheme will allow definition of visualization routines appropriate for the data and store these routines along with the data.
- The visualization algorithms are provided by a highly-efficient and proven visualization framework called ParaView. In our work we show an effective and robust solution to access ParaView's visualization capabilities in a way that allows visualization pipeline concurrency (Section 3.4).

1.5 Structure

The remaining part of this thesis starts with a discussion of the client application that we developed as part of this project and we show the basic functionality offered to its users. After that Chapter 3 provides a rather high-level, conceptual overview, discussing the architecture and design of our framework without going into details. In Chapter 4 we provide more details and discuss implementational aspects of the framework. The last part of this thesis, Chapter 5 and 6, contains our results and conclusions respectively.

Chapter 2

The Remote Visualization Client Application

In this chapter we show what users of the client application written for our project will be able to do and see. By doing so we anticipate on the discussion of the design and implementation of a framework needed for remote visualization of large amounts of data. The conceptual and detailed discussion of our framework, including the client application, can be found in Chapter 3 and Chapter 4 respectively.

2.1 User Audience and Existing Software

Common design rules in Man Machine Interaction (MMI) state that a user interface for an application should always consider the target audience that will be using the application (for example in [2]). An interface that is too complex will scare users away, while an interface that is too simple may not give users the full potential of the application. Therefore, we made an effort to define which kind of users would be using our client application. After defining the target audience we will also review an existing (medical) application with a slightly similar purpose.

The Orbita project, from which this thesis project has emerged, has one main goal. That goal is to make the Orbita dataset, which is unique for several reasons, accessible to people from around the world and allowing them to do research on the data with their own computers, while still keeping control over the data. An example of research functionality for this kind of data was brought forward by attendees of the ARVO [1] annual meeting that were interested in this software being able to comparatively visualize an MRI and histological modality of a recently digitized specimen.

The Pelvis dataset described earlier serves a different purpose. This dataset is used mainly for educational purposes and in this situation the client application will be used in a classroom-like scenario where multiple students will be running the same application, using identical, or at least very similar workstations.

An existing application that we had the liberty of experimenting with is the Image Viewer of the EasyVision system developed by Philips Medical Systems. This viewer is used for viewing CT and MRI data and patient documents. The interface of the application is very basic. Few buttons can be found in the interface, and more advanced features are turned off by default. Available datasets are shown on the left side as icons, and on the right side the data display area is found. This display area can be divided into a 1x1, 2x1 or 2x2 layout. Importing a dataset into (one of) the display area(s) is done by either double-clicking or dragging-and-dropping the dataset icon. Each display section has a navigation map to give the user an indication of the current position within the dataset, and to enable fast traversal of the data. The navigation map in this case is a topogram image, a parallel projection of all slices, that acts as an overview of the complete dataset.

For our application, two things are worth mentioning here. First, for our application the main user base will be made up of medical specialists, researchers, and medical students, and for this group of users com-

[1] The Association for Research in Vision and Opthalmology, `http://www.arvo.org`

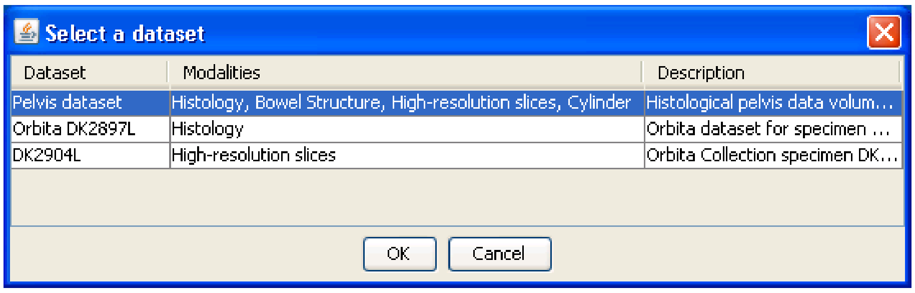
puter fluency should not be taken for granted. The second point is that hospitals nowadays are depending on computer science and software has become an essential tool supporting the medical field. Tools for visualization of X-Ray, MRI and CT images are already in use, so it is likely that expectations regarding the behavior of such applications are already set.

Note that this thesis is not focused on (graphical) user interface design but on the architecture making remote visualization of this data possible. However, we acknowledge the fact that application usability is often a key factor in the acceptance of this kind of applications. Applying an interface philosophy which is similar to that of existing visualization applications seemed like the best way to go.

2.2 General Design Philosophy

One of the main design goals while creating the client application was to keep the thresholds for users as low as possible (for running the application as well as for working with the application). With the user audience mentioned above we decided to try to keep the application clean, simple and intuitive. Slightly lower on the priority list, but nevertheless an important item, was to give the user an interactive experience. In an ideal situation we would like to provide interactivity even within visualization requests so that users can request the server to start on a new visualization when a previous visualization task is still running. Unfortunately, this would require that visualizations can be aborted and for the approach we have chosen this is not possible. In our solution interactivity is achieved by using client-side representations of the remote object (a bounding box of a volume for example) to alter visualization parameters on the client-side without much delay. Examples of this functionality will be illustrated below.

Another decision made to keep things simpler was to allow only one dataset to be visualized at a time, and with ’dataset’ we mean the collection of data for one object. For each dataset multiple modalities with multiple visualizations may be available, which we will explain later. The restriction of only allowing (simultaneous) visualization of modalities of one dataset should keep the user from getting disoriented, for example when similar-named visualizations are available for different datasets. We chose to let the user first select any of the datasets available on the server, before visualizations for that specific dataset may be started (Figure 2.1).



Dataset	Modalities	Description
Pelvis dataset	Histology, Bowel Structure, High-resolution slices, Cylinder	Histological pelvis data volum...
Orbita DK2897L	Histology	Orbita dataset for specimen ...
DK2904L	High-resolution slices	Orbita Collection specimen DK...

Figure 2.1: Before any visualization can be started the user is prompted to select a specific dataset from the list of datasets available on the server.

In order to prevent users from having to make too many context-switches mentally we tried to minimize the number of screens needed to start visualizations and interact with them, and we think that three sources of information are needed at any given moment. Obviously, the most important is the actual visualization. We expected that users will need to be able to access the available visualizations quickly and we did this by arranging the dataset’s modalities and visualizations in a tree structure. Also, an orientation view was thought to be essential, to help the user determining the current viewpoint, location in the data, or whatever is suitable for the visualization at hand (see Figure 2.2 and its caption for an explanation of the views in the interface, and Figure 2.5 and Figure 2.6 for ’live’ examples of the orientation view).

Based on already available tools for medical visualization (like the EasyVision tool described above) we decided to support multiple simultaneous views, for which the default behavior is to tile horizontally in case the visualization area is wider than it is high, or vertically in any other case. The EasyVision tools

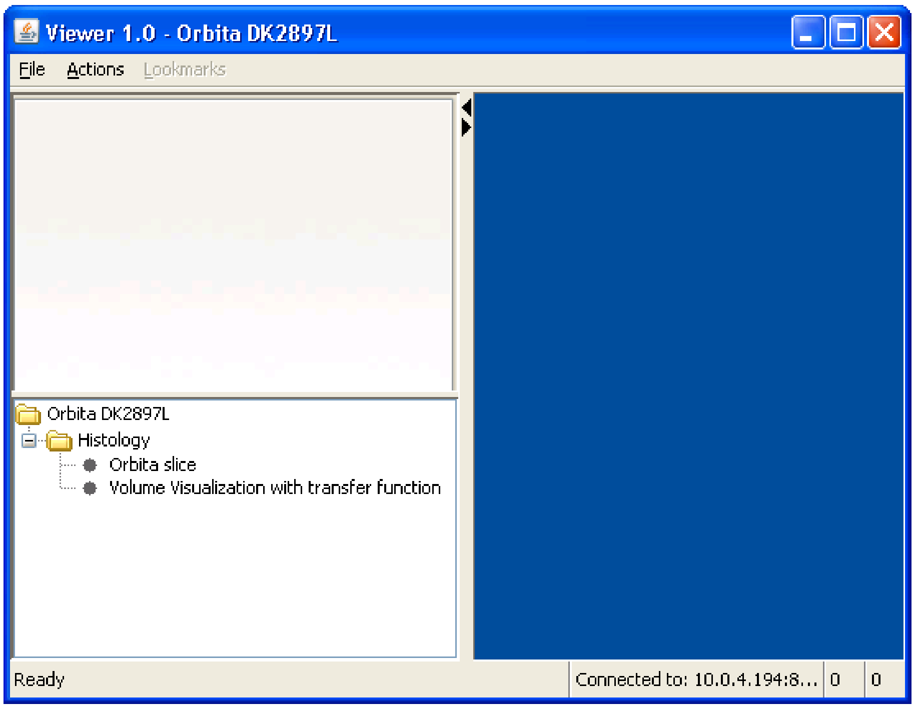

Viewer 1.0 - Orbita DK2897L
File
Actions
Lookmarks
Orbita DK2897L
Histology
Orbita slice
Volume Visualization with transfer function
Ready
Connected to: 10.0.4.194:8...
0
0

Figure 2.2: The client interface showing the three basic areas of the main screen: the orientation area (top-left), the modality- and visualization-selection area (bottom-left) and the visualization area (right).

limits the layout of the visualization to be 2x2 but we decided to give the user the freedom of choosing how many simultaneous visualizations are opened and how they are arranged (note that the maximum number of simultaneous visualizations can be limited by the server).

2.3 Functionality

Although the client application we developed should be seen as a proof-of-concept or reference client implementation, we tried to create an application containing support for common (types of) visualizations, making the reference application usable to a substantial number of users. In this section we will describe the functionality that is offered by the client application at this time.

2.3.1 Modality-Visualization Selection

After the user has connected to a server and chosen a dataset to work with, the application will look largely similar to Figure 2.2. The root node of the tree in the bottom-left panel will show which dataset the user has selected, and sub-nodes will show which modalities are available for the dataset. Modality nodes may be expanded which will then show the available visualizations for each of the modalities as leaves in the tree. Clients can easily start a visualization for a given modality either by double-clicking the visualization node in the tree or by dragging-and-dropping the tree item to the visualization area on the right. It is also possible to select multiple visualizations in the tree, and drag all of them to the visualization area to start all the selected visualizations in one action. In case all visualizations of a modality (or even all visualizations for all modalities) are needed a user can drag-and-drop the modality tree node, or the dataset (root) tree node respectively. Note that starting many visualizations simultaneously may cause a heavy load on the server. Figure 2.5 shows a situation in which a single visualization of a single modality is activated.

2.3.2 Supported Data Types

Results of visualizations are retrieved (entirely or partially) from the server in a certain format after rendering has finished. The client application currently accepts two types of data from the server: image data (more specifically: image data in JPEG, PNG, GIF and BMP format) and Zoomify data [2], the data format also used in the well-known BrainMaps project [3]. The support for Zoomify images was implemented to experiment with an option to provide users with a more interactive experience. These Zoomify images can be seen as pyramids of tiles of the original image with resolution increasing per layer. The application can start with retrieving only the top layer of the pyramid, to show the client a low-resolution version of the result as soon as possible. More tiles can be fetched when the client remains idle to increase the resolution of the result. The result is that we can view very large images and allow interactive panning and zooming much like Google Maps. For more detailed information we refer to Section 4.5.4.

2.3.3 Supported Visualization Types

For our client application we have defined and implemented three types of visualizations. Because many different visualizations can be defined it is simply infeasible to implement viewing algorithms and interaction models for each of those visualizations. In our framework we have tried to group visualizations into visualization types and implemented the necessary logic for each type. We have implemented three types of visualizations which will cover many of the possible visualizations, as you can see below, but certainly not all. When needed support for additional visualization types could easily be implemented at a later time.

The three visualization types we have defined and implemented are: image visualizations, plane visualizations and view visualizations. The main input interaction model (i.e. the rules for behavior of mouse- and keyboard actions) is based on the VTK interaction model (more on VTK can be found in Chapter 3). In the client application, all views except the parameter views are built using OpenGL functionality. OpenGL stands for Open Graphics Library and is a cross-platform interface for writing applications that produce 2D or 3D graphics, basically a software interface to graphics hardware [4]. Advantage of using this library is that views built on OpenGL can use hardware-accelerated drawing. Currently we use this functionality to provide hardware-supported image panning and -zooming and drawing of geometrical primitives for the visualizations (overlaying interactive bounding boxes and cutting planes for example, as mentioned in the next paragraph).

Image visualization The image visualization type is rather basic in the sense that the number of ways to interact with the data is limited. This type is functionality-wise comparable with common image viewers and supports interactions like zooming (right-mouse) and panning (left mouse). This type is accompanied by an orientation view that shows the part of the image that is currently visible in the visualization view. An example scenario for this type could be the visualization of high-resolution slices. Combining the image visualization type with the Zoomify data type will provide a smoothly panning and zoomable view on large images, while the user can keep an overview using the orientation view provided.

Plane visualization Plane visualization types can be used for visualizations in which users need to be able to specify planes through a volume. Visualizations offering functionality to define arbitrary slices can be created. The plane type has two ways of interaction: one for navigating the result of the visualization, e.g. a reconstructed plane (where the interaction is the same as with the image visualization type), and one for manipulating the orientation and position of the plane (shift+left-mouse for rotating the plane normal, shift+right-mouse for moving the plane position along the normal). To keep an interactive user experience we implemented the interaction model in such a way that the effect of manipulations will be visible at interactive rates on the client-side. Only after the mouse-buttons are released the altered plane parameters will be exchanged with the server. After a plane manipulation the orientation view belonging to this visualization will show the selected plane through a bounding box of the volume. In Figure 2.3 an example is shown of what a user might see when manipulating a plane through a volume.

[2]http://www.zoomify.com
[3]http://www.brainmaps.org

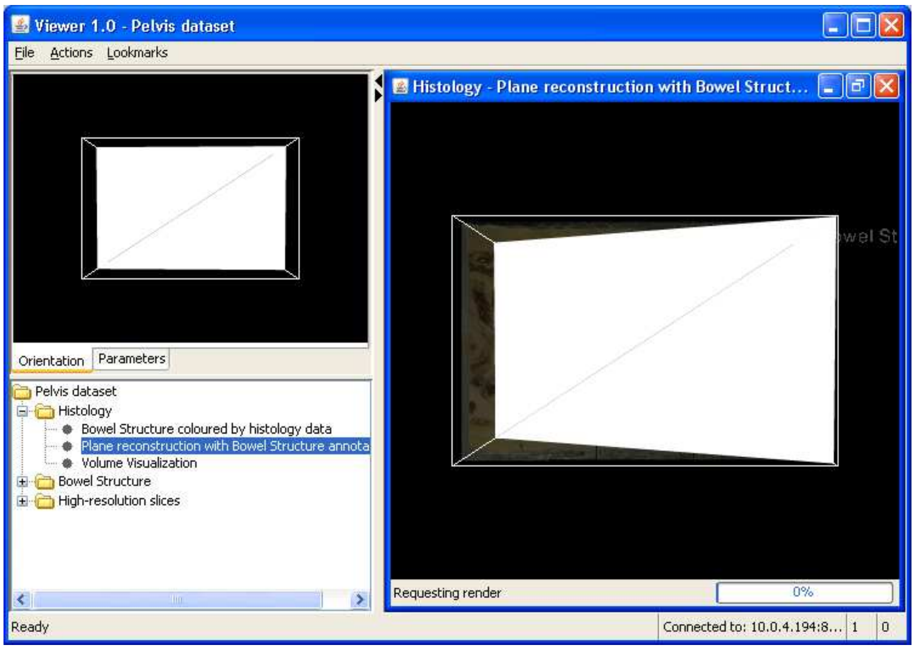

Viewer 1.0 - Pelvis dataset
File Actions Lookmarks
Histology - Plane reconstruction with Bowel Struct...
Orientation
Parameters
Pelvis dataset
Histology
Bowel Structure coloured by histology data
Plane reconstruction with Bowel Structure annota
Volume Visualization
Bowel Structure
High-resolution slices
Requesting render
0%
Ready
Connected to: 10.0.4.194:8...
1
0

Figure 2.3: An example of interaction on a plane visualization. When a plane manipulation interaction begins the old visualization result is greyed-out and an interactive, client-side representation of the new plane orientation is projected on top. While the user-interaction continues the visualization will not be updated.

3DView visualization The third visualization type supported is used for camera-aware visualizations in which a three-dimensional view is rendered. The same 'lazy' interaction as with the plane visualization is used, where the user will manipulate (rotate, zoom, etc) the scene by only manipulating a bounding-box of the subject on the client-side. After releasing the mouse button the altered camera settings are sent to the server and a new visualization is requested. The orientation view of the view visualization type will show the current orientation of the volume in space. Figure 2.4 shows a screenshot of the manipulation of the three-dimensional scene.

2.3.4 Annotations

Particularly useful in the case of medical applications or medical educational environments is the support for annotations. Annotations can provide additional information on images by showing texts about and/or shapes of interesting objects in the data as an overlay on the data itself. For our reference application we have added basic client-side support for overlays of annotations using the plane visualization type described above. In Figure 2.5 an example is shown of what annotations may look like. Our reference application will currently only outline the annotation objects on the image data, along with the object name placed top-right, relative to the outline. Without going into details on how the support for annotations is accomplished on the server-side (see Section 4.5.2) it may be useful to know that these visualizations are actually done in two steps: first, the image data for the selected plane is extracted from the volume. Second, the annotation is requested from the server and drawn over the image data.

2.3.5 Linked Views

In comparative studies it can be important to be able to visualize different modalities of a dataset using the same visualization parameters. Consider an example where both MRI and histology volumes are available for a certain object. In this case comparative visualization can be used for example to navigate to a certain slice in the histology volume, and see what the slice looks like on an MRI.

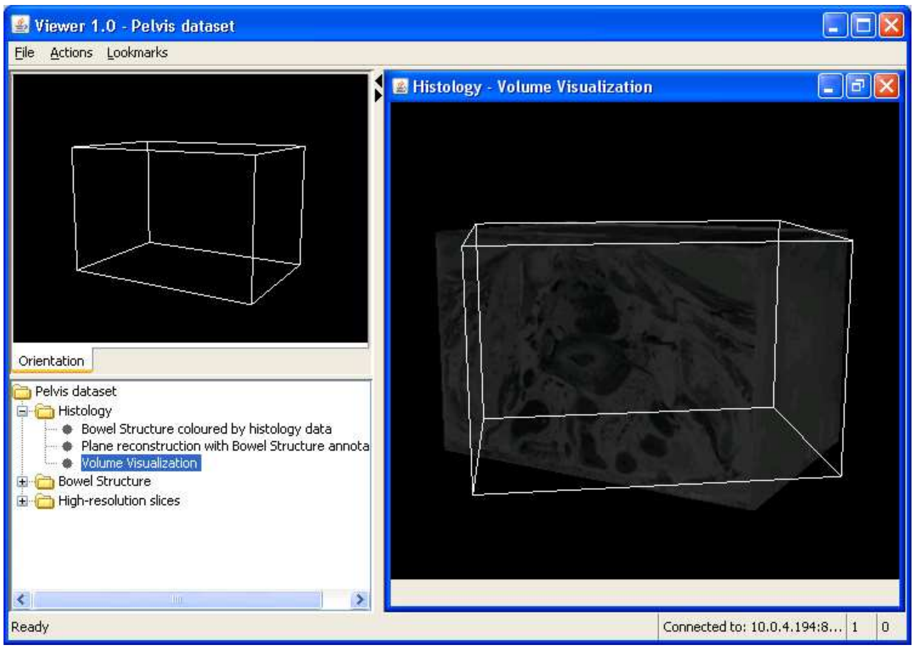

Viewer 1.0 - Pelvis dataset
File
Actions
Lookmarks
Histology - Volume Visualization
Orientation
Pelvis dataset
Histology
Bowel Structure coloured by histology data
Plane reconstruction with Bowel Structure annota
Volume Visualization
Bowel Structure
High-resolution slices
Ready
Connected to: 10.0.4.194:8...
1
0

Figure 2.4: When a (three dimensional) view visualization is manipulated the old result is greyed-out like in the plane example above, but an interactive representation of the new orientation of the volume (using a bounding box), is displayed over the visualization area. Like the screenshot above the visualization will not be updated as long as the user is interacting.

To support this we implemented the concept of ’linked views’ or ’synchronization’. In [5] linked views are defined as an interaction mechanism between views of a dataset where interaction with one view will modify the display of data in the linked views. Also, several ways of linking between views are discussed. For information on the implications linked views may have, see [6]. The general idea of the application of linked views in our framework is that a user can start two visualizations, say A and B, select A and then synchronize the visualization to B, or vice versa (synchronizing A to B and then B to A at the same time is not possible). When two visualizations are synchronized each manipulation of the parameters of the primary visualization will be propagated to the secondary. The result can be seen in Figure 2.6.

2.3.6 Lookmarking

When browsing on the Internet the term ’bookmarking’ is often used to refer to storing an Internet address for later use. The term ’lookmarking’ is used much in the same way, like a bookmark of what one is looking at. The term lookmarking was actually borrowed from ParaView [3].

The first step in using the lookmarking functionality is to create a lookmark for a visualization that is displaying some interesting features for example. In Figure 2.6 the action menu shows the menu item that is needed to create a lookmark. For each defined lookmark the server will explicitly record the used dataset, modality, visualization and parameters. After entering a description the lookmark will be added by the server (by default available only to the user), and thereby be made available for retrieval later on. Opening a lookmark can be done by selecting the appropriate lookmark from the ’Lookmarks’ menu, after which the application will instruct the server to restore the session to the way it was at the time of lookmark creation.

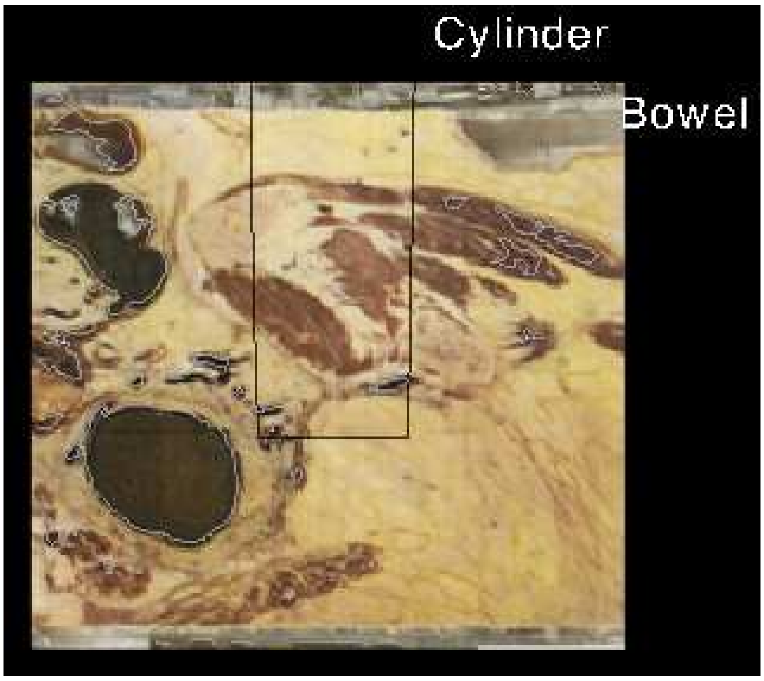


Figure 2.5: A sagittal plane reconstructed from a volume, with the outlines of two objects-of-interest as overlay on the extracted plane. First object is the rectangular shape on the plane, resulting after cutting a sagittal plane through a cylinder, and second is bowel structure, showing as white outlines of the dark-brown areas of the slice.

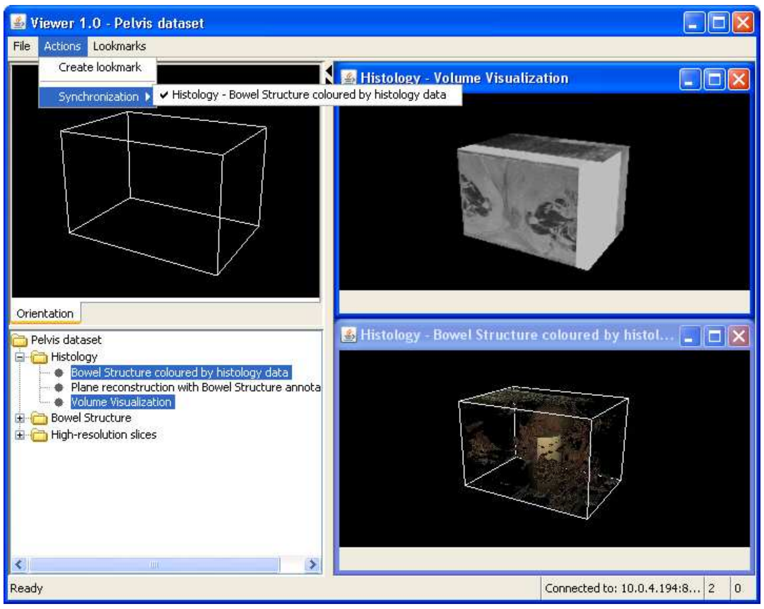

Viewer 1.0 - Pelvis dataset
File
Actions
Lookmarks
Create lookmark
Synchronization
✓ Histology - Bowel Structure coloured by histology data
Histology - Volume Visualization
Orientation
Pelvis dataset
Histology
Bowel Structure coloured by histology data
Plane reconstruction with Bowel Structure annota
Volume Visualization
Bowel Structure
High-resolution slices
Histology - Bowel Structure coloured by histol...
Ready
Connected to: 10.0.4.194:8...
2
0

Figure 2.6: An example of two simultaneous visualizations where the lower visualization is synchronized to the upper visualization. As can be seen from the screenshot, the functionality is available from the 'Action' menu.

Chapter 3

Architecture and Design of a Remote Visualization Framework

In this section we provide a high-level overview of the basic elements that our framework consists of. Prior to this thesis research [12] was done on existing visualization systems. We also suggested an architecture for which we expected that it would meet our objectives and many of the design decisions made in this section are based on this suggested design.

3.1 Architecture Overview

From a high-level point of view our framework contains two types of components: server- and client components. Although a client component was developed for the purpose of this project, the idea was to have the communication between client and server done using well-supported protocols, which ensures that our reference implementation can be easily replaced with a custom implementation by anyone who wishes to do so. As the reference application is open-source, custom implementations may extend or alter this implementation instead of completely rewriting the application. Obviously, choosing a different programming language for implementation will require a rewrite of the client application.

Most important for this thesis and most relevant for this chapter is the architecture of the server components. The Client component will therefore be discussed in more detail in the implementation chapter, Chapter 4.

To aid the reader in getting an idea of the design of the framework we will provide a system design overview, showing all architectural components, along with the relations between them. In Figure 3.1 one can see that the framework consists of four components: a client-, middleware-, visualization- and database component.

In the remaining part of this chapter we discuss the server components on an architectural level while not addressing implementation details yet. The RDBMS component, the relational database that is used to store data, is a standard third-party relational database and will not be discussed here.

3.2 Requirements

Before the actual development of the (custom) components we assembled a list of requirements to see what the server components should be capable of in order to be useful to us [12] (chapter 7). We briefly describe these requirements here and afterwards discuss major design decisions, which are largely based on those requirements.

Globally reachable As we want to provide a public domain service, the service should be globally reachable and allow access to everyone that has an internet connection and is allowed access to the service.

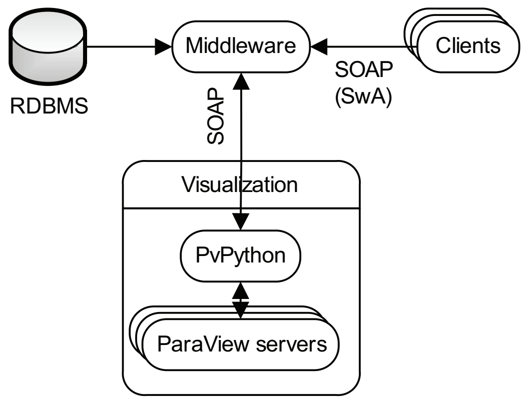


Figure 3.1: High-level system diagram showing the main components in the framework.

Support for Iso-surfacing, Direct Volume Rendering (DVR) and Multi Planar Reconstruction (MPR) These three types of visualizations are considered requisites for the framework and should therefore be supported.

Parallel and distributed rendering Given the large amount of data and the calculation-intensive visualizations possible it is essential to use as many server resources as possible as efficient as possible and the support for parallel rendering is a key factor in coping with this amount of data. Supporting distributed rendering would create an additional benefit as this means that work can also be divided among multiple physical servers.

Besides the requirements above we also proposed secondary requirements in our earlier research: the support for watermarked or fingerprinted transmission of image data, support for annotations and operating system independency for the server-side components. Meeting the first requirement means that security risks may be further reduced. Annotations are commonly used in medicine as a way of clarifying raw images by adding information like shapes and textual explanations for certain structures in the data. Therefore they can be useful tools in educational environments. Being operating system independent, together with support for distributed rendering, would mean that a more versatile collection of servers (versatile in the sense of computer architectures and/or operating systems) could be integrated in the chain of computers systems available for rendering tasks.

3.2.1 Discussion

Fairly early in the project the choice was made to go for remote visualization (as opposed to visualization on a local workstation) and to provide the required functionality as a service over the Internet. Although this way the framework requires an Internet connection on both the server and the client side, this can hardly be seen as an issue nowadays. Added benefit of this solution is the central storage of the data and of easily providing means for user authentication. To circumvent firewall issues we looked for a solution that would use default network ports instead of arbitrary ports which are often blocked by default firewalls, in company and institute networks for example.

For the actual visualization work we chose to use an existing visualization system instead of developing our own solution, for which would then need to incorporate visualization routines for at least the basic visualizations mentioned in the requirements above. Apart from the huge amount of development work, this custom visualization solution would probably not be flexible and would have to be actively maintained. Existing solutions are far more flexible and are updated regularly for bugfixes and new functionality. In order to meet our requirements we had to find a visualization system having parallel- and distributed rendering capabilities, which would be capable of providing the aforementioned visualizations and a system that would allow other applications to control the system.

3.3 The Visualization Components

Choosing a visualization system is important in this project as this choice will affect many of the properties of the resulting framework. Prior research [12] on appropriate visualization systems available for this project has shown that ParaView [13, 14], a parallel visualization framework developed by Kitware Inc. and Los Alamos National Laboratory, is a very suitable candidate to be used as foundation for the visualization component of this project.

3.3.1 VTK

The ParaView framework is built on top of VTK (the Visualization ToolKit) [15] which is also under development by Kitware and supported by many people in the open-source society, Los Alamos National Labs, Sandia National Labs and many more. VTK has become one of the most popular toolkits for all kinds of visualization-related tasks such as scientific visualization, image processing, volume rendering and many more. The toolkit is open-source and written in the highly efficient programming language C++, allowing for fast graphical processing, and VTK is designed to run on multiple operating system platforms.

3.3.2 ParaView

In the design of ParaView three major requirements were considered, as explained in [14]. The first of them is usability, as the VTK framework which is used as the basis of ParaView is merely a toolkit, and is not exactly easy to use. The second is scalability. As the name of the framework already explains, work should be done in a parallel way, to improve performance by utilizing more resources concurrently. Providing scalability is therefore essential to the success of ParaView. The last requirement considered is extensibility: although ParaView already has a lot of VTK modules incorporated, it is possible to register additional components after specifying their interface. Although the top layer of the application is written in C++ most other things (from the widgets to inter-process pipeline synchronization) are done in Tcl/Tk, which is a scripting language which makes extending the application easier.

Summarizing, we can say that the aim of ParaView is to take the extensive and efficient VTK visualization functionality, add a layer for parallel processing of visualization pipelines on top of this functionality and make the system more accessible and extendible for the user.

Several properties of ParaView, besides the ones already mentioned above, should be mentioned here in order to further justify the use of ParaView for our project. Just like VTK, Paraview is open-source software and under active development. Actually, at the start of this project we used ParaView version 3.2.1. During this project two major product updates (versions 3.3 and 3.4 respectively) were released and successfully migrated into our project, while version 3.6.1 was released very recently.

By far the most important feature of ParaView over VTK is the support for both distributed and parallel processing. In this sense, Parallel processing is the term for using multiple processors in a single computer system, where distributed processing is used to define the use of multiple systems. The difference between the two may not seem that big, but with parallel processing you know beforehand which architecture (32-bit or 64-bit, Big Endian or Little Endian[1], etc.) you will be running on because all processors in a single system are obviously based on the same architecture. In contrast, distributed computing may include systems with various architectures and Operating Systems, including uncommon supercomputer architectures like in the Cray XT3 supercomputer[17]. For parallel processing in ParaView the Message Passing Interface (MPI) is used, which is a high-performance protocol for message exchange in parallel computing.

ParaView also has some limitations and unfortunately it has one specific limitation that may adversely affect our framework. Where VTK supports the concept of streaming, i.e. the chunk-wise reading and processing of data to handle arbitrarily large data structures in a limited memory footprint, ParaView is, at this time, not capable of supporting this. This basically means that the size of the data to be visualized should never exceed the combined total amount of memory (both physical and virtual) available to all of the computers cooperating in the visualization.

[1]Big Endian refers to the way of storing numbers in memory starting with the lower bytes of the number, where Little Endian stores them the from high to low bytes

Two remarks should be made on this limitation. The first is that the current state of technology allows a single server to be equipped with hundreds of gigabytes of memory. Combining several of these servers will allow for volumes of several hundred gigabytes in size, which is very large for a single volume or even a collection of volumes. Besides this practical argument of volume size, KitWare has also recently released an updated version of ParaView (3.6.1) that is said to contain an application called StreamingParaView, which should support the streaming processing of volumes. Because ParaView 3.6.1 and StreamingParaview were announced at the time of writing of this thesis and because of the experimental phase the StreamingParaView project seems to be in (there is hardly any documentation available, there are no pre-built executables) we have decided not to try to incorporate these in our framework at this time.

As ParaView is specifically designed to run on a distributed, parallel set of computers it supports several modes of operation [18]. The basic components of the ParaView framework are the dataserver, the renderserver and the client, which can be combined for several set-ups. In [12] we described which renderings set-ups are possible with ParaView, but only three modes will apply for our project: the client-server, client-distributed server, and the client-distributed data-distributed render modes. For this project the second option was chosen, which we will explain in Chapter 4. However, any of the other rendering modes could be supported and will only require small modifications to the framework.

3.4 Interfacing with Paraview

To be able to use the functionality of ParaView we need to enable communication between ParaView and our middleware. Although this section has a large implementational aspect it is also a vital element of the design of the framework and would leave a gap if not discussed. For this reason we placed this section in the current chapter instead of in the implementation chapter.

Interfacing the two components did not turn out to be trivial, although the final solution is relatively simple and flexible. We mentioned earlier that ParaView is an open-source framework, therefore there are many entry points to interface with ParaView, and several options have been tried before a final solution was chosen. The solutions that were tried can be divided into three directions and they were examined in the following order:

- Implementing the ParaView protocol
- Creating a shared object for loading by the middleware
- Exploiting ParaView's Python support

Implementing the ParaView protocol

The basic idea behind the first approach is simple: if we are able to implement the underlying protocol that ParaView uses (called 'CSS' or Client/Server Streaming [18]) we can basically do what ParaView does, with minor overhead. If our middleware would have been implemented using C++ this issue could be solved by linking our project to parts of the ParaView source, however, for reasons stated above, we are in a different position. Implementation of the protocol in Java was the only way to accomplish our task, but several things make this solution infeasible. First of all, the protocol is not well-documented, and extracting the protocol from the source code would be very time-consuming. A second, and even larger, objection to this approach is that it is very low-level and has its limitations. The CSS protocol will only enable remote invocation of methods implemented in ParaView, and do that platform independently. In [18], section 4.2 the limitations of only using the CSS protocol are clearly summed up and they will be paraphrased here.

Complexity of use The work required to be done in order to invoke just a single method is rather tedious: prepare a network stream, gather method name, arguments, choose the target server and node and send the stream.

Lack of state As the CSS protocol keeps no state information, clients requiring information on the state of server-side objects will need additional method invocations to get up-to-date. Besides this inconveniency, the authors state that this can easily lead to high network loads.

Lack of data gathering A stream can only send and receive data from a single node, but the protocol itself cannot gather information from all nodes working together on a specific visualization operation.

To cope with these limitations of the CSS protocol, ParaView has implemented a ServerManager module providing a higher-level interface to the functionality. However, we would have to implement this module in Java too, making this implementation a project of its own and clearly an unfeasibly approach for our project.

Creating a ParaView shared object

Another attempt at interfacing with ParaView was to create a library (or shared object), compiled against the ParaView source, which can be loaded into the Java process space. All methods provided by the library can then be invoked from the middleware using the Java Native Interface (JNI). For detailed information on JNI have a look at [19]. In short JNI allows Java programs to execute native, platform-dependent code. To accomplish the task of creating a library we have used a tool called SWIG (which stands for Simplified Wrapper and Interface Generator). SWIG is capable of connecting C/C++ projects with a large variety of other high-level programming languages (PERL, PHP, Python, but also Java for example). To use SWIG one has to define an interface file describing the native methods to expose to the target programming language. After running SWIG with Java as target language, SWIG will generate a C/C++ wrapper (to be compiled with your native application), along with some Java files containing the JNI calls and the definitions of the objects referenced. Using SWIG we were able to both create a library compiled against ParaView and to load that library in a Java project.

Unfortunately, several issues of this approach became apparent. First, all functionality available in ParaView must be exposed somehow by the library we were using, making this solution inflexible. Also, loading a library (or basically using JNI in general) in an application container environment like Tomcat is rather difficult, in providing paths to shared objects for example, but also because of the on-demand nature of deploying and using web services. Last but certainly not least, we ran into a general issue of parallelization. ParaView uses several managing and controlling objects that are referred to in a static way, following the singleton pattern. These objects are responsible for keeping the state of a pipeline but are not designed to keep the states of multiple pipelines simultaneously. Basically, if you would like to have multiple pipelines running next to eachother in a given system (for example one for each client accessing the middleware), you would need to have multiple ParaView instances running. But with the approach taken, loading the self-compiled ParaView library in the Java process space, it is not possible to load this library multiple times, making it impossible to have multiple pipelines running concurrently. Even this last issue alone made this approach useless, because it violates the idea of our framework to service multiple clients at the same time.

Exploiting ParaView’s Python Support

While VTK provides wrapping of its functionality in three other programming languages (Tcl, Python and Java to be precise) ParaView offers only Python wrapping [17]. Python support is available after enabling the wrapping with a parameter in the ParaView building process. The by-product of this build consists of two additional command-line tools: pvpython and pvbatch, which both offer a fully functional Python interpreter, with the difference that pvpython can work interactively. In the process of wrapping ParaView for Python, the ServerManager module mentioned earlier is also implemented in Python by the ParaView developers. The ServerManager module applies the proxy design pattern [18] where each (distributed) server-side object is represented by a proxy object on the client-side which is responsible for managing the server-side object’s lifecycle and properties. As this ServerManager module and much of the logic for accessing the ParaView functionality is already wrapped for us, using these Python-enabled ParaView tools will provide us with a simple and clean solution to access ParaView’s functionality from our middleware. As an added benefit, these tools are also MPI-enabled, meaning that they can be started concurrently using the MPI protocol (as described above in Section 3.3.2) as communication between them.

The command-line tools can be executed by starting one of the Python-enabled executables, with a tool (called ’mpirun’) that will start one or more MPI-enabled processes concurrently. Starting the execution from our middleware will create a subprocess which has its own process space and therefore its own singleton objects, avoiding the limitations of the earlier approach. Summarizing what this solution means for our project:

- We can have multiple pipelines running simultaneously.
- We can have fine-tuned control over the number of ParaView nodes working together on a given pipeline.
- All processes are in their own process space, introducing the robustness that if a process crashes for some reason, it will not crash the parent process (which is our middleware).
- We get all default Python functionality and the functionality of third-party Python libraries for free.

As this turned out to be a suitable solution for our project we decided to implement our framework using the Python interface provided with ParaView. To be able to set up ParaView pipelines from Python we have developed a Python script (a 'glue' component) that will be run by the 'pvbatch' executable and that is responsible for providing the ParaView functionality to the middleware. To enable two-way communication between this Python component and the middleware we will have the Python component publish its functionality through a SOAP service. The result is that the middleware will access the Python components' SOAP service to access ParaView functionality and the Python component will access the middleware's SOAP service to provide updates on pipeline progress.

Additional advantages of this solution are rapid development (each client connection will run the Python script so changes to the script will only require a reconnect), and providing a common location to add re-usable pieces of Python code to be used from various visualization pipelines. In Figure 3.2 we zoom in on the part of the system diagram that is showing the connection between our middleware and the ParaView visualization system.

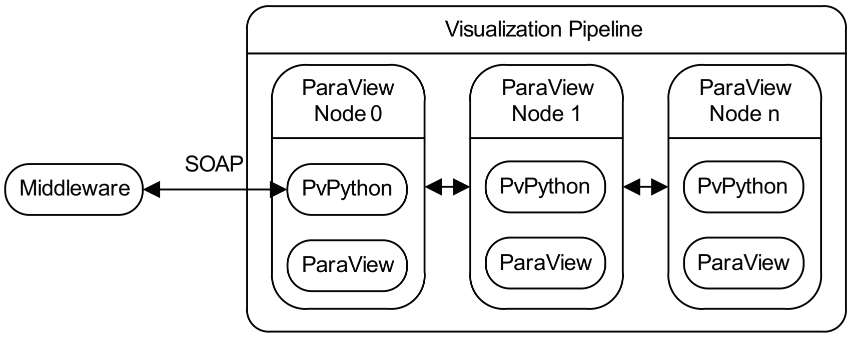


Figure 3.2: For each visualization session the middleware will start one or more ParaView nodes that will be working together on the visualization. The middleware will only communicate with the ParaView nodes through node 0. This means that node 0 will be responsible for compositing the output from all nodes and that all Operating System interaction (such as loading and saving of data) will be relative to the system on which node 0 is running.

3.5 The Middleware Component

The middleware component we developed is the most important contribution of our project. The component is a server process that takes client requests and communicates with the visualization back-end. It is responsible for serving clients with visualizations, appointing visualization server resources and for controlling those resources (the visualization server processes) while hiding ParaView's complexity from the user. In this section we will motivate important decisions made in the process of developing this component. Some additional terminology will be provided as background where needed for a correct understanding of the information given.

3.5.1 Server-Client Communication

One of the project's main objectives was to provide clients with remote rendering capabilities and this requirement forces the framework to be designed in a client-server fashion. A major decision in the general design of client-server systems is the choice for a certain way of communication between client and server. One possibility would be to implement a custom communication protocol but this would needlessly complicate the development of a custom client application and it would also mean rewriting functionality for which better solutions are available.

Over the past few years the concept of Web Services has become very popular (see [20] for a discussion on the Web Service paradigm). Web services can provide all kinds of services to clients, like retrieving or processing information, doing heavy calculations on better-equipped servers, etc. They are implemented as Remote Procedure Calls (RPC's) that can be invoked remotely over the HTTP protocol that is used by web servers. Older web services implemented Remote Procedure Calls by using XML (eXtensible Markup Language), a common format to describe structural data in human-readable text. Large disadvantage of XML is that it is rather inefficient for transferring binary data. Because we expect to be transferring much binary data through the web service required for our framework XML is not a viable option. More recent implementations of web services wrap the remote calls and responses into messages according to the Simple Object Access Protocol (SOAP) which addresses this inefficiency in an extension called 'SOAP-with-Attachments' [2] (SwA). In [16] SOAP and SwA are compared, showing that SOAP-with-Attachments has lower processing overhead and higher throughput than 'regular' SOAP.

Attempts to create a common web service interface have resulted in the construction of the Web Services Definition Language (WSDL). With WSDL one can abstract a web service into endpoints and operations on those endpoints, along with the definition of inputs and outputs to those operations.

Because of the versatility and the (ongoing) standardization of web services we have chosen webservices as the basis for the client-server communication in our framework. Many implementations of web service frameworks are available but our attention has focussed on an implementation by the Apache Software Foundation (ASF) called Apache Axis2[3], or short: Axis. The Axis framework is both a client and a server implementation of a web service engine that is based on SOAP and which is implemented in the programming language Java.

The last step to create a web service using Axis is writing a Java Servlet based on an Axis Servlet and to deploy this servlet in a Servlet Container, along with a definition of the functionality that needs to be exposed through the web service. Servlet Containers, or Web Containers, are responsible for providing an execution environment to servlets by taking care of deploying, starting and stopping servlets, and receiving and sending request and response messages. Many Servlet Containers are available, but we have chosen to use Apache Tomcat, another project of the Apache Software Foundation, to deploy our Axis servlet, mainly because of interoperability.

Note that, while Tomcat will by default run in stand-alone mode, it is also possible to integrate a Tomcat configuration into the popular Apache web server[4], making the web service transparently available over the default network port for web traffic, port 80. This may be important when firewalls (or organizational policies for example) prevent Tomcat to act as a web server on other ports when the default port is already in use.

With this setup we have a solution with standardized components with proven technology and which are already actively maintained. Figure 3.3 shows the basic infrastructure of the web service for our framework when embedded in a regular web server.

3.5.2 Management

The middleware component will not be producing any results or data but it is responsible for managing most aspects of the framework, as explained in the introduction of this section. We have recognized four

[2]See `http://www.w3.org/TR/SOAP-attachments` for information on the extension

[3]The AXIS2 Framework of the Apache Software Foundation (ASF): `http://ws.apache.org/axis2/`

[4]The Apache web server has the highest market share for web servers, running on approximately 47% of all web servers around the world (based on estimates of NetCraft in June 2009 at `http://news.netcraft.com/archives/web_server_survey.html`.

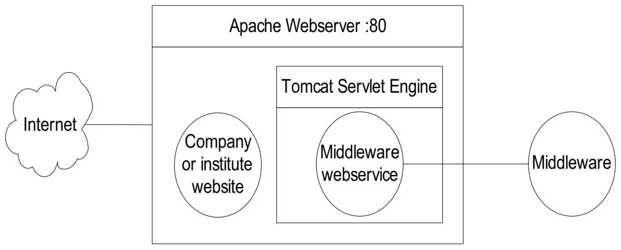


Figure 3.3: An overview of our middleware-and-webservice when embedded inside an Apache webserver (which is also serving some website). In case the Tomcat Servlet Container is used in stand-alone mode Tomcat will serve requests to the webservice by starting a webserver of its own.

types of management activities, and we chose to define a ’manager’ for each of these activities, to be able to cleanly separate the activities in the source code of the middleware. In Figure 3.4 we show how the middleware component can be broken down into smaller components: the web service and the managers. Specific tasks for these managers is mentioned below, along with a short explanation of the functionality provided.

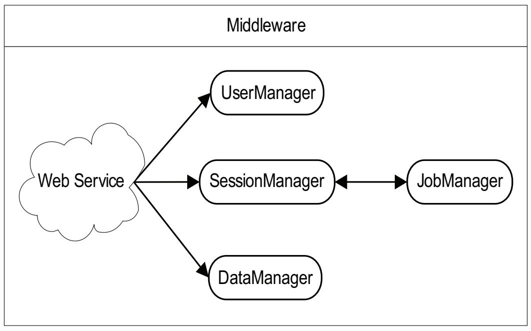


Figure 3.4: The middleware component in more detail, showing the different managers and the (flow of) communication between the web service and the managers.

Data Manager The Data Manager is responsible for keeping track of the datasets and visualizations available on the server and of the relations among them. This manager also deals with reserving space for jobs and storing job results, both on disk and in the database. When a job is finished, the Data Manager will also retrieve the data for these results on client request.

Job Manager The Job Manager will maintain a pool of workers, which are threads in the middleware that can carry out work. This pool has a fixed size, and its size will define the number of sessions that can be served simultaneously. Each session started by a client (through the web service), is submitted as a separate thread in the pool. The most important functionality offered by the Job Manager is running the jobs that are submitted. Each job belongs to a session, and these sessions will be appointed to workers in the pool when there are workers available. When a worker picks up a session the visualization pipeline needed for this session is assembled. A new ParaView instance is started and ordered to start on the first job. The worker will continue to run jobs until the session is either closed or idle for too long.

User Manager The user manager takes care of retrieving users and (hierarchical) groups from the database, and performing user authentication. The User Manager also provides functionality to specify exactly which groups have access to which dataset or visualization, based on the concept of Access Control Lists (ACL’s). A detailed explanation of this functionality can be found in Section 4.4.

Session Manager The other managers are relatively passive, compared to the Session Manager. The largest part of the calls to the web service are handled by the Session Manager. This manager is in charge of setting up and closing instances and sessions on client request. A ’session’ in this sense is the sequence of renders for a specific visualization of a specific dataset for a specific client. A client can start multiple sessions from a client application, which are then considered sessions in a single ’instance’. A user starting two or more client applications will also have two or more instances on the server. Besides managing sessions and instances, the Session Manager will also appoint all jobs submitted by users to the session that they belong to.

Another important piece of functionality offered by the Session Manager is the collection of a status report for a given instance. Such a status report contains the updates for all changed jobs for all changed sessions for a given instance. Because of the nature of a web service, all information that is exchanged will be exchanged in messages, going from the client to the server and with a response from the server back to the client. We discuss this limitation of the web service concept in more detail in Section 4.5.1.

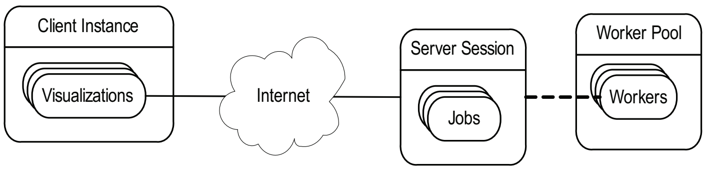


Figure 3.5: Relation between client-side visualizations and server-side sessions, and between sessions and workers. Each separate client visualization has one session on the server side (in the middleware) and each session will be handled by one worker.

3.5.3 Data Management

A substantial amount of time and effort was spent on finding a way to effectively manage the available data (we more strictly define the term ’available data’ below) and in this section we report on the results of this effort. Besides that we consider the chosen solution suitable for our framework, we also consider the solution a valuable contribution to this thesis. An essential argument in the reasoning towards our solution is that datasets and visualizations are highly related and should not be seen as separate entities. We will show that, despite the tight relation between datasets and visualizations, careful division of the visualization pipeline still offers a flexible solution.

3.5.3.1 Data vs. Meta-data

When defining the ’available data’ we should specify two kinds of data. First we have the actual image- or volume data acquired in some digitization process, like scanning or photographing histological sections, doing MRI scans, etc. Besides this type of data we also have meta-data, which contains properties like a description of the object examined, information on the acquisition technique used, location of the actual data, etc.

3.5.3.2 Dataset vs. Modality

In medicine the term ’modality’ is used to refer to the method that is applied to treat or examine a patient[5] and in imaging applications a modality would refer to the way the acquisition has been done. For our purpose we consider a modality to be some appearance or form of a dataset. MRI and CT scans and histological volumes for instance are examples of modalities, acquired by MRI and CT scanning devices or by scanning or photographing histology sections respectively.

The idea to define a modality in this section is to create the possibility of comparative visualization between different modalities and therefore we will consider the fact that multiple modalities of a single object (human orbit, pelvis, etc) can belong to the same dataset. More specifically, in our framework

[5]In `http://en.wikipedia.org/wiki/Modality` for example

we consider modalities to be sub-elements of a dataset in the collection of data that we have to manage (Figure 3.6).

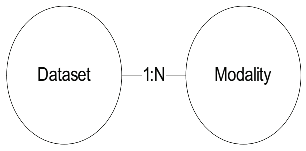


Figure 3.6: To support multiple modalities we added a 'Modality' object with a one-to-many relationship between the dataset and the modality.

3.5.3.3 Dataset vs. Visualization

A conceptual visualization pipeline is usually separated into three steps (see [21]):

1. Data Enrichment/Enhancement: preparing and filtering the raw data through interpolation, smoothing, error-correcting, etc into 'derived data'
2. Visualization Mapping: mapping the 'derived data' to Abstract Visualization Objects (AVO's), representing graphical primitives with attributes such as colours, positions, etc
3. Rendering: drawing the Abstract Visualization Objects to a rendering context using the optical models that are available

In practice (i.e. in ParaView) the conceptual pipeline is extended (prefixed) with a step to supply the raw data to the pipeline. This means that in order to run visualizations in our framework any pipeline must consist of at least these four steps. In the first argument above we stated that datasets and visualizations are highly related and should not be considered separate. This relation exists because the input to the visualization, the visualizations' 'raw data', is completely defined by the modality that we would like to visualize.

3.5.3.4 Solution and Design

In a relational database it is straight-forward to model the dataset- and modality objects, and the one-to-many relation between them. We used a standard third-party database (in our case MySQL [6] to manage this information, but any relational database would suffice. Part of the power and flexibility of our solution is that we were also able to include the (objects of) the pipeline in our relational model. A naive solution could approach the issue by predefining complete pipelines for each modality, like in (Figure 3.7).

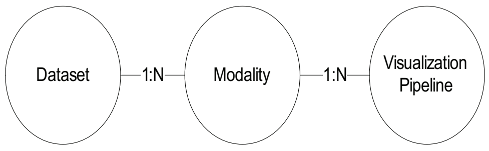


Figure 3.7: A naive solution could predefine a visualization for each available modality, a solution that is rigid and lacks re-use.

Our approach addresses the issue of lack of re-use by separating the pipeline into a 'Source' and a 'Visualization', where the 'Source' is responsible for providing input for the pipeline (based on the chosen modality) and where the 'Visualization' contains all the elements of the conceptual pipeline. The proposed scheme is illustrated in Figure 3.8.

[6] `http://www.mysql.com/`

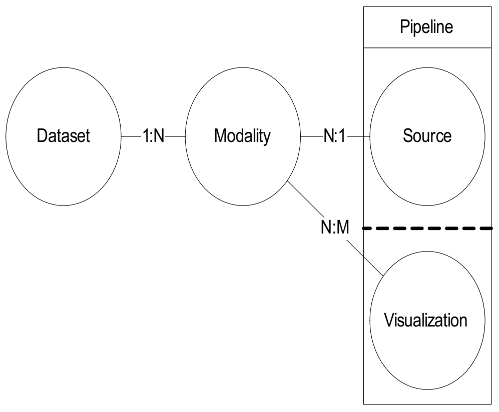


Figure 3.8: Our proposed solution splits the modality-determined part of the pipeline (which we will call the ’Source’) from the visualization logic (the ’Visualization’) and models the pipeline elements as separate objects. Source objects will be used to provide input to the visualization, while the ’Visualization’ provides the output.

In Section 3.4 we discussed our choice to use Python to control visualizations in ParaView and in Figure 3.8 the ’Source’ and ’Visualization’ parts of the pipeline are depicted as objects with relations to the modality object. A contribution of our approach is that each pipeline object is stored in the relational database as a snippet of Python code that is responsible for accessing the required ParaView functionality. Any ’Source’ pipeline object will contain the Python code necessary to load the data for a given modality compatible to the source. All ’Visualization’ objects will contain code necessary to transform the input data into some defined visualization.

3.5.3.5 Conclusion

With the definition of datasets and modalities, and the definition of separate pipeline objects using Python snippets, we have obtained a highly flexible solution for our visualization framework. Several interesting properties of our approach can be mentioned, which we can sum up as follows.

- Multiple modalities can be specified for each dataset.
- For each type of modality data only one ’Source’ object needs to be written and maintained, which can be re-used for other modalities with the same data-type.
- For a modality multiple visualization routines (’Visualization’ objects) can be stored, and these objects can be used by one or more modalities.
- Arbitrarily complex pipeline objects can be created in the high-level programming language Python, which is well-documented and well-supported and many third-party additions to Python are freely available.
- Our approach encapsulates datasets and appropriate visualizations in a single relational database model, where all pipeline objects for the visualizations will be stored in the database along with the data. This means that the database will contain both the data and the visualization routines appropriate for that data on a single location.

In this section we tried to provide a high-level overview of our scheme of data encapsulation and motivated the process of getting to this scheme. Because the actual scheme is slightly more complex we will supply further details of the data encapsulation in Chapter 4.2.

Chapter 4

Framework Implementation

In this chapter we discuss implementational topics and issues that we faced in the development of our framework. We start with a more detailed discussion of the client application. Although we could have included this discussion in the previous chapter on the framework architecture we chose to discuss the component in this chapter. This choice was based on two main considerations: the architecture of the client component does not (substantially) contribute to the architecture of the framework. Also, it is an expandable part of our implementation that can be easily replaced by a custom implementation.

4.1 The Client Component

We mentioned earlier that we developed a (reference) client application for this project. The basic idea of this client application is to interface with the middleware component and submitting user requests, retrieving the results from the server and displaying them on the users' screen when partially or completely transferred.

4.1.1 Requirements

In the introduction of this thesis we already mentioned some of the design goals for the client application. As decisions made in the design of the client application heavily influence the usability and appearance of the application we will discuss them here, starting with the requirements that were recognized beforehand, like we did above while discussing the server design decisions.

In the prior research mentioned earlier [12] we discussed and recommended several requirements appropriate for the client application. All of these requirements have been adopted in the design of the client application. All of the requirements are related to the general idea that as many clients as possible should be able to use the application (without making heavy demands on the user's computer resources and -environment).

Small installation effort While applications are usually available on the client computer after performing some installation routine, this installation will only make the application available on the workstation it is installed on and usage on any other workstation will require a new installation. Also, we should keep installation as short as possible, by keeping the amount of data installed to a minimum.

(Nearly) Operating System independent To conform to the general idea above we tried to avoid creating an application that depends on some specific operating system. While in general the Microsoft Windows operating system dominates the other available operating systems, the actual numbers on operating systems used by our audience may well be different.

No persistent storage of data Many applications install application data (or store data that was previously fetched over a netwerk connection) alongside the application on the target computer. Locally stored data has the advantage of providing easier and faster access to data. However, for most histological datasets that advantage is hardly beneficial because of the lengthy installation process (not even regarding of the large amount of free space required on the client computer). In addition, persistent storage of valuable datasets involves security risks, which should obviously be minimized.

Low internet connection bandwidth Although high-bandwidth internet connections are becoming common, for example for researchers in academic institutes, it is still essential that users are able to access the data collections from their homes, using their own (usually limited) internet connection.

Interactive, simple and intuitive interface The last requirement mentioned here basically contains two parts (an interactive interface and a simple and intuitive interface), were both elements are related to the user experience. Interactivity is essential in providing a good user experience because an unresponsive interface will surely demotivate users. Unresponsiveness in this sense can mean two different things. First, unresponsiveness can occur in an application where a lot of work is done but progress is not reported back often enough, by means of screen updates, progress bars, etc. Another kind of unresponsiveness can be found in the process of (remote) visualization. The actual task of visualization, or the transfer of the result of the visualization, may take long and cause the client to get the feeling of a slow and unresponsive interface.

In Chapter 2.1 we defined what we consider the target audience for this client application, with the amount of computer fluency that can be expected for this group of users. A simple interface will probably be most efficient for this group of users. Note that this requirement as stated here, containing words like ’simple’ and ’intuitive’, is rather subjective. Although it is possible to provide measures of simplicity and intuitivity, by conducting a study using people from the target audience, we consider this to be out of our scope.

4.1.2 Decisions

From a design point of view, the choice for a client application implementation language is important. Several options are available, the most promising of which are AJAX, Adobe Flash, C/C++ and Java. The first two options are web-based programming environments and those languages are designed for applications that run inside a (compatible) browser and using JavaScript in the case of AJAX or a plug-in from Adobe in the case of Flash. An application written using any of the two languages can be considered a thin client. The term ’thin client’ is used for a (remoting) application which has a very small resource footprint, and which relies heavily on a server component doing the more intensive work. Fat clients on the other hand have a larger footprint, can do more work themselves and usually provide more functionality to the user. Applications in C++ are compiled from code to a native code supported by the platform, where Java code is compiled to intermediate code, therefore requiring a so-called JVM (or Java Virtual Machine) to run. In this case C++ has the advantage of running natively (without an interpreter), but the disadvantage that the code has to be compiled per platform, while Java code can be run on any platform for which a JVM is available. Applications in C++ and Java can be anywhere between thin and fat clients.

For several reasons we chose Java as the programming language for our client application. Although thin clients are often used as means to provide web-accessible content they can easily become limiting factors in the functionality of the application, an example being the lack of support for dedicated graphics hardware. Because C++ applications run natively and can be optimized further (because the target-architecture is known at compile time) they will usually be faster than similar Java applications. However, most of the heavy work will be carried out by a visualization server as explained earlier in this chapter, and they are not platform independent. Java applications[1] can be run on any platform for which a Java Virtual Machine is available, meaning that these applications are practically platform independent. On top of this platform independence, an additional advantage is the availability of an automatic deployment mechanism called Java WebStart, available on any workstation that has the Java Runtime components installed. This mechanism will take care of loading, verifying and caching components of an application. After successfully loading the application it will be started as a stand-alone application (as opposed to the well-known Java Applet which is very limited and which only runs inside a browser).

[1]We assume pure-Java applications, which conform to the rules of portability, and which will not include code, or reference to components, that make them platform dependent.

4.1.3 Model-View-Controller (MVC)

In Chapter 2 we showed that we defined multiple views for each type of visualization (the view containing the visualization itself, an orientation view and a parameter view). Because of the tight cooperation of these views with a single source of data we applied the well-known Model-View-Controller architectural pattern (for example in [22], chapter 14). For a correct understanding of the inner workings of the application we briefly explain the pattern and afterwards discuss the result of applying the pattern to our client application.

The pattern defines three components, the model, the view(s) and the controller, and an MVC application is made up of a collection of these triplets. Each of the components in the pattern represents a role in the interface interaction of the application.

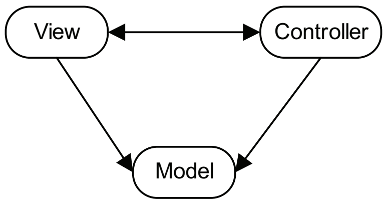


Figure 4.1: The Model-View-Controller pattern components and interaction.

In short the separate components can be described as follows.

Model A model contains the domain-data of the application and the domain-logic needed to manipulate the data. For our application, we have a model containing the visualization's data and its parameter.

View A model can have multiple views, of which the model has no knowledge. A view is a certain representation of the data, which does not necessarily have to be visual. In our application for each visualization views are created containing the visualization result, an orientation view and a parameter view.

Controller The controller is the central element which processes events, for example mouse clicks in the interface, and which is capable of invoking changes on the model. Our application has controllers for each type of visualization (for example for dealing with three dimensional visualizations in which you can rotate the object, plane reconstruction visualization in which you will not rotate the object, but the cutting plane, etc.)

The pattern has proven itself to be useful in dealing with two design issues we faced. First, the pattern cleanly separates data from visualizations-with-interaction which is beneficial for the readability and maintainability of the source code, two properties that are of great importance in open-source projects. Besides improved readability, the clear separation of the pattern elements (specifically the separation between view and controller, and the separation of the model from the other two elements) makes it easier to implement support for other types of visualizations or other data formats. And, when properly followed, the pattern will make sure that views on the model are independent of each other.

In our case we have three views for each visualization: the view containing the (result of) the visualization, the orientation view and the parameter view. Although visualization parameters are mostly changed by user interaction events such as zooming and panning in the visualization view, they may also be altered manually in the parameter view. Changes to any of the views should be correctly propagated to the other views, for which the Model-View-Controller pattern provides all the needs.

4.2 Data Encapsulation Scheme

4.2.1 Early approaches

In our search for a suitable structure to store our meta-data we went through an iterative process. Limitations of a certain solution force the solution into a different direction, which in turn leads to new insights, and possibly another iteration after that.

A promising approach to create a visualization pipeline within Python was based on the fact that visualizations in ParaView (and most visualization systems) are created by setting up a chain of filters, where the output of one filter is connected to the input of the next and the last filter is used as input to ParaView rendering helper objects (such as 'representation' and 'view' objects which map the objects to a scene).

The idea of solely specifying these chains of filters seems logical at first and storing them in a database structure is straight-forward. However, the idea turned out to be rather complicated for several reasons. Filters needing slightly more advanced parameters, lookup tables and programmable filters for example, always required additional, custom code to set up. Another issue is the use of the helper objects: some pipelines need a representation and a renderview (basically all visualizations using a camera) while others will not use representations at all (for example a pipeline that reads some input volume, retrieves a slice from the input and writes that slice to an image file).

4.2.2 The Chosen Approach

In Figure 4.2 a screenshot from the reference client application is shown in which the objects in the encapsulation are clearly visible in a tree structure. The top-level in the tree defines the dataset selected by the user and the modalities of the dataset can be found at the second level. All leaf nodes in the tree are visualizations specified for the available modalities (and they can be re-used for more than one modality, the 'Volume Visualization' appears twice in the tree).

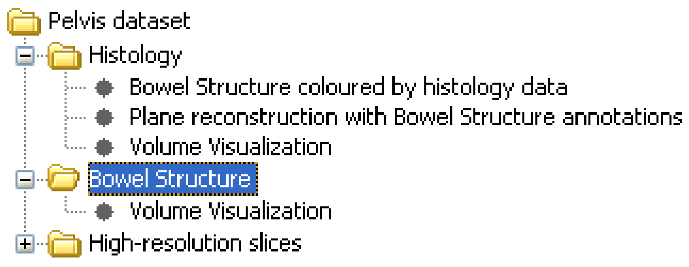


Figure 4.2: The client application screenshot shows the three visible objects in our 'encapsulation' scheme: datasets, modalities and visualizations.

As we explained in the previous chapter, the designed 'data encapsulation' scheme is slightly more complicated, our scheme uses four data objects which we explain in more detail in the following sections. We also explain how the middleware will assemble a pipeline from the different data objects, using the Python snippets we defined for them.

4.2.2.1 Datasets and Modalities

An important idea in the final structure that we obtained is the addition of a layer below the top-level element of a dataset: a modality. Our definition of 'modality' is a little broader than the medical definition. The reason for this is that information derived from the dataset should also be used as a modality. This gives us a convenient and straight-forward way to incorporate annotations in the structure, which we will explain later on in Section 4.5.2. It is possible that two or more modalities within a dataset are aligned to each other and our framework allows this information to be stored in the database as well.

4.2.2.2 Visualizations and Outputs

Datasets and modalities only define what kind of data is available in the system, but not how the data should actually be visualized. For our framework a combination of a 'visualization' and one or more 'outputs' was used. This is a minor extension to our definition of the encapsulation scheme in Section 3.5.3 in which we only mentioned the existence of the 'visualization' object. The visualization is responsible for specifying how the data is to be processed (basically, which filters will be set up) to obtain the visualization. The output(s) will be responsible for doing the actual rendering work from frame to frame and (usually) write the output to the file system in some format.

To cope with the large variety of visualizations and outputs possible within our encapsulation we require both a visualization and an output to define their ’type’, but these types have very different meanings. The supported visualization types were already listed in Section 2.3.3.

Visualization type The type of visualization defines the shape of the visualization before rendering. After rendering it will be an image, like a screenshot, but before rendering it might for example be a 3D view, a plane reconstruction or just an image.

Output type The output type defines how the result of the visualization will be stored on the server. For example we can choose to render and save it to a lossless PNG image or to a heavily compressed JPEG image, a polygonal model, etc. Note that multiple outputs may be defined for a visualization, leaving it up to the user (actually the client application) to make a decision between them. We will show an example of this in Section 4.5.4.

In fact both types are irrelevant for the middleware but they are relevant for client applications: the visualization type implicitly defines the interaction that is possible with the visualization, where the output type defines what the client must do in order to successfully draw the visualization output to the client screen.

4.2.2.3 Building a Visualization Pipeline

We now defined all building blocks for our data structure, but lack the final step needed to setup a complete visualization pipeline in ParaView. As mentioned earlier in Section 3.4 we have introduced an intermediate layer of Python between our middleware application and the ParaView server. In order to construct a complete pipeline we need three pieces of information:

1. The modality to be visualized (usually chosen by the user)
2. The visualization required (usually chosen by the user)
3. The output of the visualization (usually chosen by the client application)

Each of the pieces of information above defines a snippet of Python code to carry out the work, as we explained in Section 3.5.3. In our intermediate layer of Python we assemble pipelines from the pieces by executing the Python snippets in-order. A crucial detail in the implementation is that we make sure that the running context (or the ’scope’) of these pieces is maintained among all the snippets during the lifetime of the pipeline. By running all snippets inside the same scope we can make sure that each of the snippets will have access to objects (parameters, functions, etc) instantiated or altered in any of the previously executed snippets. The process is depicted in Figure 4.3.

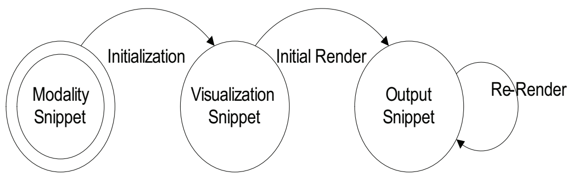


Figure 4.3: The consecutive steps of the pipeline when assembled from the three pipeline objects that we have defined earlier. Loading of data is done in the modality snippet, visualization setup in the visualization snippet, rendering (and frame-to-frame changes) in the output snippet.

4.3 Resource Sharing

4.3.1 ParaView Processes

In order to successfully manage server resources it is important to know, in slightly more detail, how ParaView works under the hood when requested to run in parallel. We mentioned earlier that ParaView is started by running the pvbatch or pvpython executables in an MPI environment (and MPI is responsible for starting the actual ParaView processes in a distributed way across the servers). This distribution of processes can be controlled by the person or process starting ParaView. MPI needs two things in order to start the processes: a parameter specifying the number of nodes needed, and the location of a so-called 'machinefile'. In this file a list of the available machines and the number of nodes allowed to run on each of them can be specified. What MPI does next is iterate through the specified machines in the machinefile and start the requested number of nodes.

All the processes started know their ID in the chain of cooperating nodes (in MPI terms: the 'ring'). In Chapter 3 (Figure 3.2) we already showed the sitation in a diagram and made a distinction between the so-called 'Node0' and the remaining nodes. It is essential for the understanding of this section to know the difference between Node0 (basically the master process) and the other nodes. There are many tasks for which this process is responsible, but the following are most relevant:

- The middleware communicates with ParaView through this node.
- All nodes exchange data (for example for results) with the master node.
- Both reading and writing of data is done by the master node, and filesystem actions in the pipeline are therefore always local to that node.
- In case a ParaView reader is not capable of parallel reading this node might be used to distribute the data.

Some logical consequences follow from this list of properties. First of all, it is important to note that each machine in the cluster might be running the master node[2]. Results of a job are usually stored on disk and the framework should be able to access these results to be able to send them back to the client. This means that each server should be able to access a shared location for storage.

In certain situations it may be beneficial to distribute master nodes evenly across the cluster. Consider a situation where it is common to run visualizations that require a lot of I/O activity from harddisks (large datasets stored in raw file formats, for example). In this situation it makes sense to duplicate the data to each machine and have the master nodes distributed evenly accross the machines. As each Node0 will have access to a locally stored copy of the data the total data throughput can increase.

Each of the machines in the cluster will be inter-connected by some kind of network technology. Bottlenecks in a network are common for data-intensive visualizations, but knowing that all nodes exchange data with the master node could help us prevent these bottlenecks. The idea is that, when you are distributing processes, you could try to keep as many processes as possible local to (i.e. on the same machine as) the master node. Each of these local processes can communicate without a network, effectively minimizing the amount of data transferred over (limited) network connections.

4.3.2 Distribution Strategies

The previous section contains some remarks that can actually be seen as recommendations and for our framework we tried to wrap recommendations like these into distribution strategies. These strategies are implemented as objects containing the logic to specify for each worker how many ParaView processes it is allowed to start and where to start these processes. Although the framework can easily be extended to include more strategies we have currently implemented two strategies: a Simple strategy and a Locality strategy.

[2]It is possible to force the master node to a certain machine, by making sure that the first entry in the machinefile always points to that machine, however, this scales badly because any session in the framework will have at least one ParaView process running on this machine

Simple Strategy The Simple strategy is a default strategy in which the administrator of a framework can control most of the distribution properties. In its configuration you can define how many ParaView nodes are allowed to start in total and also how many are allowed per session. The configuration also specifies the location of the MPI machinefile, and the administrator can predefine which machines are available and the number of slots available on these machines. Because the machinefile is fixed, the Simple strategy cannot prescribe where the processes are started, but it will prescribe how many processes are started. This number always obeys the following set of rules:

- Assign the number of nodes requested by the user (not mandatory) or otherwise half of all ParaView nodes available.
- Never allow more than the maximum number of nodes per session.
- Never allow to leave less nodes available than there are idle workers.

Locality Strategy The Locality strategy was created with the idea in mind that it may be more efficient to keep nodes local to the master node. This strategy is more advanced as the Simple strategy for several reasons. The strategy can be made aware of which machines are capable of running ParaView, how many (physical) cores are available for each of them, and of a value of preference for each machine. With this information, and the fact that the strategy keeps a record of running processes the following behavior is obtained:

- In case the user requested a certain number of threads find all servers that have at least this number available, otherwise pick the server(s) with the most available threads and assign this number. Still obey the second and third rules of the strategy above.
- If multiple machines are left as candidates choose the machine with the lowest 'overbook' ratio (i.e. the lowest ratio between running processes and physically available cores).
- If still there are multiple candidates choose the server with the highest preference value (which is usually the server that has the fastest processor cores).

4.4 User Management and Object Authorization

Datasets can be very expensive and they can be the subject of on-going research and may therefore not be freely available to the public. For those reasons it is important to have some way of specifying who may or may not use the services provided by our framework. Two levels of control were implemented for our project, with the first level being user verification. At logon each user can be asked to supply a username and password, which will be verified against the server. After successful authorization, the server will check if the maximum number of sessions (visualization views) or instances (client applications running) for this user has not been reached yet.

On a second level, a more advanced authorization scheme based on ACLs or Access Control Lists was implemented for the framework. With ACLs the framework administrator will have fine-grained control over the visibility of certain objects in the framework. Objects in this case are relevant elements of the data encapsulation structure explained earlier in Section 4.2. ACLs are always linked to groups, not to users. Users can be in one or more groups, and groups themselves can be in one or more other groups, where the 'Global' group is a predefined group that is the parent of all other groups, either directly or indirectly.

For the framework several so-called privileges can be defined, but currently only one privilege was implemented: the 'select' privilege. Examples of additional privileges could be a 'modify' privilege if functionality would be added to support altering information on database objects, or an 'annotate' privilege if support for client-side image-annotating would be added. Note that adding privileges will always require changes to the code of the framework, while adding ACLs will not.

Each ACL can contain zero or more privileges, and for each combination of ACL-and-privilege multiple groups can be added. To clarify the general idea we provide an example. For a given system, two groups are defined, 'researchgroup1' and 'researchgroup2', that each have a dataset available in the framework. For our example, the first group should be allowed to access both datasets, where the second group will only be allowed to access their own dataset. To enforce this restriction, a system administrator should add an ACL (say ResearchDatasetACL), and add the privilege ’select’ to the ACL. For this combination of ResearchDatasetACL with the ’select’ privilege we add one group: the ’research1’ group. The last step is to connect the newly created ACL to the ’dataset’ object of the ’researchgroup1’ group.

From that moment on, only the ’researchgroup1’ group will have the ’select’ privilege on the dataset object for this group, where the dataset object for the other group has no ACL enforced, and will be accessible by both groups in the system. Note that, without an ACL, all groups in the system would be able to access the dataset of the ’researchgroup2’ group, including other groups besides the two groups mentioned. It would certainly make sense to add another ACL to the second dataset, allowing only the two research groups to access it.

The current framework implementation enforces ACLs on three of the data objects mentioned earlier: datasets, modalities and visualizations. For the output object it does not seem to be useful at this time to restrict access for clients based on ACLs.

4.5 Additional Functionality

In the development of our framework we addressed many different issues, but some of these issues are not related to just the server components or just the client component. In this section we report on topics that do not fit in any of the previous sections of this thesis but which are important implementational aspects of our framework and which we consider essential for a discussion of our project.

4.5.1 A Full-duplex Web Service: Semi-polling

A web service using Remote Procedure Calls (RPC’s) will always exchange messages as question-and-answer. Because of network-issues such as firewalls and NAT (Network Address Translation) protocols requests are always initiated by the client and each request can only be followed by a single response (as opposed to either a full-duplex connection, where both parties can communicate at the same time, or even a half-duplex connection, where only one of the two parties in a conversation can communicate at the same time but still both ways). This pattern of communication, which is characteristic for web services, has some obvious drawbacks and is most apparent in our case when the server needs to signal the client that a job has finished. Some jobs may take a long time to finish and it may also be possible that clients want to submit a batch of jobs. For these reasons the framework needs to be asynchronous: the client submits a request and the server will not respond with the result of the job but rather with a message that the job was successfully received.

Because only the client can initiate communication with the server the concept of ’polling’ is required to provide the client with information on outstanding jobs. The idea of polling is that a request will be sent to the server on which the server may respond with the latest status of the job(s). In our framework we approached the issue by implementing semi-polling. The difference is that the server will block a response to the client for as long as possible until either the request has timed out or the job status has actually changed. This is an efficient way of exchanging the required information with low response times (a blocking request will immediately unblock when new information is available) and low amounts of traffic.

Recall that we defined a client instance as the collection of all sessions belonging to a single client application (in Section 3.5.2). With this definition in mind we tried to further reduce the number of messages exchanged between clients and server. Instead of requiring a (semi-)polling thread for each of the clients’ sessions in an instance a single request to the server will block until either the request times out or until any of the sessions in the instance have changed status. This way each client application will require only one polling thread to collect status updates for each of the sessions within that application.

4.5.2 Visualization Annotations

Most definitions of ’annotation’ describe an annotation as a note or remark added to some document at some location (see [23] for example). In visualization an annotation is usually represented as a label (with some color and size for example) on an object, which is used to provide additional information about the object.

In a framework capable of 3D visualizations we think an annotation should be more than just a label and we came up with a solution which nicely integrates annotations into our encapsulation model, by considering annotations to be modalities of a dataset. This solution has several advantages: annotations can now be arbitrary objects (text, images, 3D polygonal models, etc) and they can be connected to visualizations, just like 'regular' modalities.

In the discussion of our encapsulation scheme above we mentioned that two modalities can be aligned to each other in our relational database model. To support annotations the middleware is capable of adding the (data of) aligned modalities to the Python scope of a visualization pipeline. This means that any visualization of a modality that has modalities aligned to it can use the data of the aligned modalities to add annotations to the visualization.

Consider the following example. In a certain situation there are two data objects available, a histological volume of a human orbit, and a polygonal model of the optic nerve within the same volume. We would like to create a visualization which extracts arbitrary planes from the volume on which the outlines are drawn of the intersection of the plane with the optic nerve. In order to create this visualization we would add the two data objects as modalities to a dataset and in the database model we specify that the polygonal model is aligned to the volume modality. Finally we add a visualization for the volume modality, which extracts a plane from the volume and cuts the polygonal model with the same plane, after which the output of both the extraction and the cut is rendered into a single scene.

4.5.3 Linked Views and Synchronization

In Chapter 2 we described client functionality to support the concept of linked views for comparative purposes, but until now we did not define any of the prerequisites of linking two views. In this sense the term 'view' will refer to a complete visualization pipeline, so basically it will refer to the window a user will see in the client application when starting a visualization on a modality.

We will not restrict linking of two or more views on the basis of having equal modality, visualization or output. We chose to define the compatibility of two views for linking differently. Each object in the encapsulation scheme has an associated set of parameters which are used to control the visualization. For any of these parameters it can be specified whether or not they are essential for synchronization. To decide whether or not two visualization pipelines can be linked we make two subsets of parameters that are required for synchronization, one for each pipeline. Finally we check whether the lists contain the same parameters and if they do the views are considered synchronizable.

An example could be that for two views (say A and B), both showing volume rendering visualizations, the camera's need to be synchronized to make sure that the two are always viewed from the same eye-point. For this example it will be sufficient to mark all camera parameters for both visualizations as being required for synchronization, after which it is possible to link either view A to view B (thus B becomes the primary view) or view B to view A (A becomes the primary view). After linking any movement to the camera in the primary view will be propagated to the secondary view to ensure that both visualizations are always viewed from the same eye-point.

4.5.4 Zoomify Support

A technique called 'Zoomify', work of Zoomify Inc.[3], was originally developed for interactive browsing of large images on websites. In our section on the supported data types in the client application (Section 2.3.2) we explained that Zoomify images are basically pyramids of tiles of the original image, where each layer (or 'tier') in the pyramid is in a different resolution. The bottom layer of the pyramid contains tiles (all tiles have a predefined size, by default 256x256 pixels) of the image in the original resolution, while the top layer is made up of just a single tile with a low-resolution version of the original image. Advantages of this technique over full-resolution images are that we do not need to transfer the entire image before we see our result and that we can prioritize fetching certain tiles of the image to allow the user to quickly zoom to a certain area of the image if needed. More technical information on the Zoomify technique can be found at the Zoomify website, but also in [1].

[3] `http://www.zoomify.com`

The support for Zoomify requires implementation on both the client and the server side. The server side needs to be capable of ’zoomifying’ the result of the visualization, where the client needs to be able to fetch and reconstruct the tiles into a single image. Implementation of Zoomify on the server-side was easy because of the availability of an open-source Python library[4] to convert regular images into Zoomify pyramids. Because the pipeline objects are also in Python code, they can easily use the functionality of this third-party library to convert the result of a ParaView visualization into a Zoomify image.

The client-side implementation required more work. In Object Oriented Design (OOD) terms we might say that we implemented Zoomify image objects as an extension to regular image objects. This gives us the advantage that every type of visualization (as explained, currently Image views, Plane views and 3D Views are available) that is using image data will also be able to use Zoomify image data. Currently all visualization types use image data as input, which means that all visualizations currently supported are also available with the Zoomify functionality automatically, without further implementational work.

At this point the advantage of being able to use multiple outputs per visualization object becomes apparent. A system administrator could define two outputs for a given visualization, one output that results in a (regular) image and one that results in a Zoomify image. Thin client applications that are not capable of processing the Zoomify data can now select the regular image output, where more advanced client applications can use the advantages of the Zoomify output.

4.5.5 Image Watermarking

Earlier in this thesis we mentioned that medical datasets can be expensive, either because of the exclusiveness of the data or because of the large amount of work invested to acquire and process the data. In [12] we suggested techniques like fingerprinting and watermarking to address the issue of protection. As a result we added a basic visible watermarking algorithm[5] capable of blending a watermark in tiles into the result of a job. This algorithm was added to the intermediate Python layer in which the pipelines will run which makes the functionality available to all pipeline objects. The current implementation could certainly be improved but an example is shown in Figure 4.4.

4.5.6 Lookmarking and Caching

For each job that is submitted to the system several pieces of information (including references to job results) are stored in a database and this is useful to allow users to retrieve job results on demand. However, we have also used this information for two other purposes which we will describe in this section.

As explained, the term ’lookmarking’ is the visualization equivalent of the term ’bookmarking’. Our framework supports saving the state of a visualization so a user can restore or share that visualization at a later time. From an implementational point of view we need to store some information to support lookmarking: the modality and the visualization that were chosen and the complete set of parameter values for all pipeline objects at the time the lookmark was created. These three pieces of information provide all that is necessary to start a new session and restore the old session state. Because we already stored the required information in a database the implementation for support of lookmarking was straight-forward.

With the job information stored it is also possible to support caching. The idea behind caching of jobs is that it can be beneficial to keep results available for some time until a similar job is submitted. When a similar job is submitted the result is already available without doing any work for the job. At first sight lookmarking and caching do not seem related but in our implementation they are very much alike. The similarity is that both lookmarking and caching require a job state to work. Lookmarking requires the state to restore the session, caching needs the state to verify whether a similar job was submitted previously. The difference between the two is that caching also needs to have the job result available.

4.5.6.1 Absolute Parameters

Note that in certain cases it is not possible to restore the state of the session exactly, for example with visualizations using temporal or temporary (internal) variables, visualizations of a running clock and visu-

[4] `http://sourceforge.net/projects/zoomifyimage`
[5] The algorithm is heavily based on code found at `http://code.activestate.com/recipes/362879/`

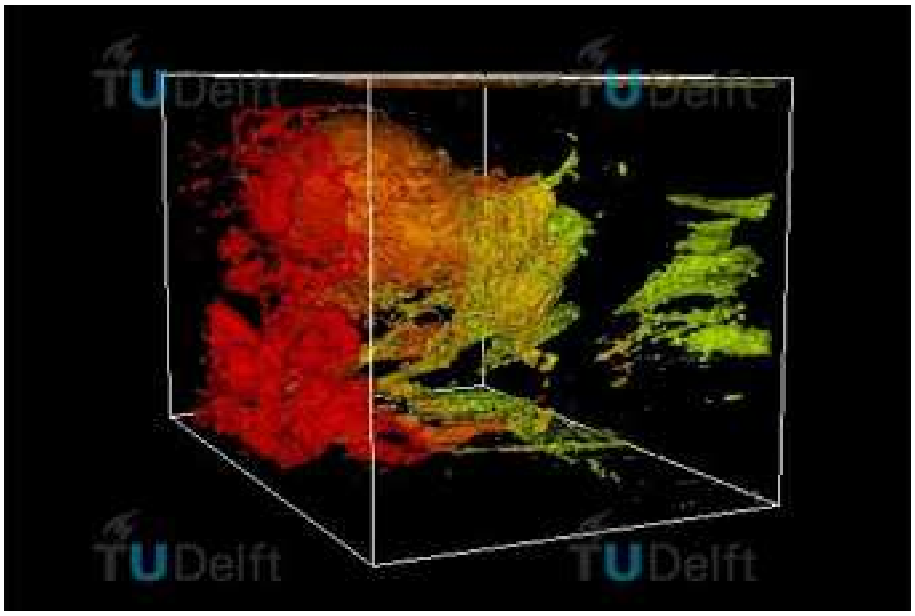


Figure 4.4: An example of a 3D rendered polygon dataset using ten parallel ParaView threads (contribution of each thread is colored differently). A tiled color-watermark containing the logo of Delft University of Technology (TUDelft), and completely independent of the visualizations' parameters, is blended into the result.

alizations where the coloring of the visualization is based on the contribution of processors to the rendering process, to name a few. In these cases the job state does not capture enough information to be either reproducible, in the case of lookmarking, or comparable, in the case of caching. To address this issue all visualizations should use absolute parameters if possible. For example a situation of a visualization using a camera. When a render from another point of view is required, new camera properties (azimuth, elevation, etc) will be specified. However, when relative parameters would be allowed (i.e. ”Rotate the scene by -1 degree” instead of ”Rotate the scene to exactly 90 degrees”) it will never be possible to restore the session correctly. Because currently it is not guaranteed that caching will return a correct result on each and every occasion we chose to disable the caching functionality. More information can be found in the Future work section of Chapter 6.

Chapter 5

Results

In this section we report on the results of quantitative measurements that we performed on a running instance of our framework in an experimental setup. The main goal of our tests was to assess the performance and scaling capabilities of our framework. Due to the practically infinite number of tests possible, we aimed to run tests that are appropriate for two specific scenarios which we already mentioned several times throughout this thesis: a research- and classroom scenario.

5.1 Experimental Setup

5.1.1 Hardware

Because we did not have unlimited hardware resources we tried to set up a fairly representative environment for our tests and therefore we prepared three machines for our tests. For the sake of comparison we describe the machines and their (important) hardware specifics below.

ParaViewServer1 The first node in our ParaView 'cluster' was dedicated for ParaView visualizations and contained an Intel Core 2 Quad Q8300 processor running at 2.50GHz (4987.41 bogomips) with 2GB of DDR2-800 memory, using a Western Digital Velociraptor 80GB SATA2 harddrive (a WD800HLFS) as system disk, which was benchmarked at 117.82 MB/second with 'hdparm', a tool included in most Linux distributions.

ParaViewServer2 The second ParaView node was also dedicated for ParaView jobs and this machine contained an Intel Core 2 Duo E4400 processor running at 2.66Ghz (5311.26 bogomips), also with 2GB of DDR2-800 memory, and using a Western Digital 500GB SATA2 harddrive (a WDC WD2500KS-00M) as its system disk, benchmarked at 63.54 MB/second.

FrameworkServer The machine processing client requests was running inside VMWare [1] as a virtual machine[2]. The host runs on a dual-core Intel E6850 processor at 3.0Ghz with 2GB of memory. The virtual environment was set up with a system disk of 12GB, was given 1GB of memory and was allowed access to both of the host's processors.

All three machines were connected by a 1Gbit network for all tests, except for those tests that were run to measure the influence of the network connection on using multiple cooperating threads spreaded across multiple machines.

Although a separate FrameworkServer is not strictly necessary in order to run our tests, the additional load of running the framework software on either of the other machines might influence the test results and therefore a separate machine was installed. Although it is less efficient to run an operating system in a virtual environment opposed to running one in a native, physical environment, the solution seemed more than adequate for our tests as the FrameworkServer hardly ever reached 50% load on either of its two cores.

[1] : http://www.vmware.com

[2] Virtual machines are machines emulated within other machines, to allow running a different operating systems inside some host operating system

Dataset	Acquisition	Voxels	Volume Size
’medium’	Section-images reduced to 10%; Only 10% of sections kept; Converted to single volume file	301×196×205	42.9 MB
’large’	Section-images reduced to 10%; All sections available; Converted to single volume file	301×196×2052	413.5 MB

Table 5.1: An overview of the datasets used for our tests.

5.1.2 Software

All three systems were installed with a fresh copy of the Linux distribution Ubuntu, version 9.04 (the latest version publicly available), including all software packages needed for compiling software from sourcecode. In addition to the default Ubuntu installation the following (system) packages were installed:

- Java (1.6.0_0), installed from the Ubuntu repositories and needed for Tomcat.
- MySQL (5.0.75), installed to support the database schemes needed for the framework.
- Python 2.6, including development components and the Python SOAP and Python Imaging library, are needed for running and compiling ParaView with Python support.
- NFS (Network File System), used to provide a central storage of shared framework components and for storage of render results.

In addition to the packages mentioned above we needed several framework-specific packages:

- Tomcat (6.0.16), does not require installation and was copied to the local filesystem.
- MPICH2 (1.1.1p1) [3], an MPI library needed for ParaView inter-process communication.
- Mesa (7.0.2) [4], an open-source implementation of the OpenGL specification. For our framework we have specifically used the OffScreen Mesa (OSMesa) implementation, to remove the need for a desktop window system (the X-window system in case of Linux) to be active. ParaView is natively capable of switching to OSMesa when offscreen-rendering is requested.
- ParaView (3.4.0), built from source (including some of our source code changes) and configured to build using the libraries specified above.

5.1.3 Various

For our tests, we used two histological volumes which we refer to as the ’medium’ and ’large’ dataset. These datasets were both downscaled versions of the LUMC pelvic dataset, see Table 5.1 for more information.

For all tests performed the results were timed with millisecond precision. Because the times measured in our experiments were usually over 100ms and certainly not sub-millisecond it seemed reasonable to use this precision.

We created a small application which is able to mimic the behavior of a regular client (i.e. the reference client application). The functionality of the application is limited to submitting jobs and receiving results from the server. The test application does not actually fetch result-data from the server or display data on the client screen. In the tool we included parameters for several things:

[3]MPICH project page can be found at http://www.mcs.anl.gov/research/projects/mpich2/
[4]Mesa project page can be found at http://www.mesa3d.org/

- The number of sessions
- The number of ParaView threads per session
- The number of jobs per session
- The number of times each test should be repeated (we have used the value five as default for all tests)
- The modality
- The visualization

In the experiments described below, where necessary, we will mention which values were used for which tests. For our tests we predefined two modalities, one for each dataset mentioned above. We also predefined a visualization, which is a default volume visualization without transfer function, with each iteration rotating the scene by one degree. In a usual framework visualization pipeline some functionality will be defined to save the output to the filesystem. However, to compare results when increasing the output resolution, we left this part out of the pipeline because saving image data involves significant overhead (relative to the output resolution). Although this overhead exists in practice it will needlessly interfere with our results and was therefore omitted.

For each job completed on the server we have access to three timestamps: the time of the submit, the start time (the time of pick-up by a worker) and the end time (the time at which a job is either finished or failed). Two important metrics from the results of the test application will often be shown in the figures below: the pipeline initialization times, which is the difference between submit- and start-time of the first job, and the job work time, which is calculated by substracting the job start-time from its end-time. See Figure 5.1 for an example of the sequence of events in a session for which two jobs are submitted and it shows the appropriate time differences measured in this chapter.

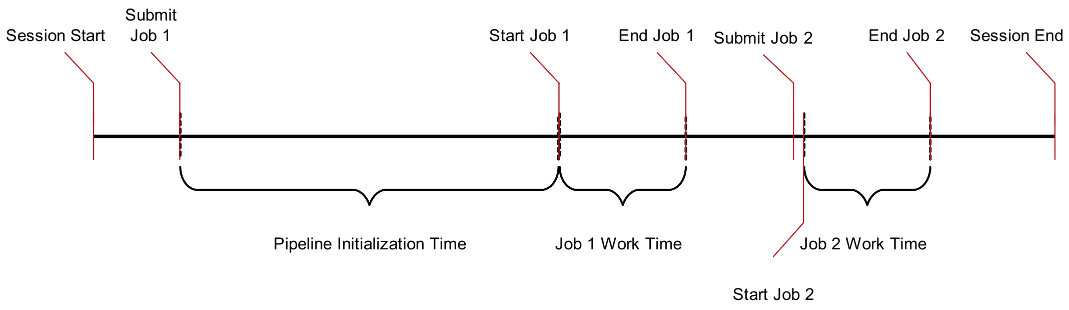


Figure 5.1: For each job we calculate three timestamps: the time of submit, start and end of the job. Pipeline initialization is performed directly after the first job is received, therefore the 'pipeline initialization time' is the difference between submit- and start-time of the first job. The 'job work time' is the difference between the job start-time and job end-time, for our tests job work times will be averaged over all jobs in the session.

5.2 Testing Terminology

Before we start describing our tests and results it is essential to explain some of the terminology needed for a correct interpretation of the tests. We refer back to Figure 3.2 and Figure 3.5 where we illustrated the relations between visualizations and sessions, and between sessions and workers. For each visualization started by a user one session is created which will be served by one worker in the middleware. This worker will in turn start one or more ParaView nodes to do the visualization. The following terms will be used frequently in the remaining part of this chapter:

core When we refer to 'cores' we mean physical processor cores. The number of physical cores is the measure for the number of tasks a processor can perform in parallel, without resorting to switching from process to process to mimic multi-tasking.

server, machine The term 'server' or 'machine' is used whenever we refer to an instance of an operating system that will be running processes, and as explained above, these instances may either be physical (the two ParaViewServers) or virtual (the FrameworkMachine).

client thread, session A 'client thread' represents one session from one user. A session uniquely defines a modality and a visualization on that modality. For example, when the reference client application is used, each visualization window for each user will result in a separate session on the server.

job Each job is used for one frame, or render, of a visualization and jobs are always started within a given session.

node, ParaView thread The middleware decides how many 'ParaView threads' are started for a given session. ParaView itself calls the ParaView thread a 'node', and when doing parallel processing it basically separates cooperating nodes in two categories: node 0 (a master node) and node 1-to-n (satellite nodes). Each of those nodes is a single process running on any of the ParaView machines available to the framework.

Strategies: '1-1', '2-1', '4-2', etc. In several cases we were interested in the effect of distributing processing nodes around the servers in different ways. We will occasionally refer to these distributions with short notations in 'x-y' form and the idea behind this is as follows. The 'x' stands for the number of consecutive processes on ParaViewServer1, the 'y' stands for the same on ParaViewServer2. If a certain number of nodes (> 1) is selected for a single job, the distribution will follow the strategy specified. For example, if we would like to run a job on six parallel paraview nodes and we choose a '2-1' strategy, the nodes will be distributed as follows: ParaViewServer1 will get (node0, node1, node3, node4) and ParaViewServer2 will get (node2, node 5). Accordingly, a '1-1' strategy will equally distribute the nodes over the machines and a '0-1' strategy will only run nodes on the second server.

As explained earlier, we performed tests based on a research- and a classroom scenario. As there are some major differences between these situations we need different kinds of tests and therefore it seemed useful to go for this two-scenario approach. For a research environment we expect that users will not request many visualizations of a certain modality, but they might have more concurrent sessions running for comparative goals. Besides using more datasets, they will probably also more often switch between them.

In a classroom scenario we do not expect that many dataset switches, but we certainly expect many concurrent client sessions doing many visualizations on a dataset that is to be the subject of a particular course for example.

The former scenario will likely benefit from parallel loading and processing of the different datasets, as users in this scenario come equally spread in time. For the latter scenario we are more interested in scaling of the job processing times when more clients access the server simultaneously. Also, the number of parallel processes will have to be restrained because these processes can easily flood the servers in several ways (at least for memory-accesses and I/O operations on harddrives), when there are many users doing visualizations at the same time.

5.3 Preliminary Tests

We first carried out some preliminary tests to quantify general properties of the framework and its underlying visualization system. We also use the results of these tests to determine some of the boundaries and parameters of later tests.

5.3.1 Caching Test

Most of the tests we did involved submitting a series of jobs for a session and calculating the average time needed to complete a job. We needed to find a safe boundary for the number of jobs to start per session. Too few will result in a large variance in the results because of possible caching effects when starting ParaView processes, too many jobs is not useful because that will not contribute to the results.

To test the behavior of running series of jobs we ran a test with a single client which will increase the number of jobs submitted per session. We did this test on the medium dataset, and used only one ParaView thread for the sessions. The goal was to find out at which number of jobs per session the average job time would not significantly change anymore.

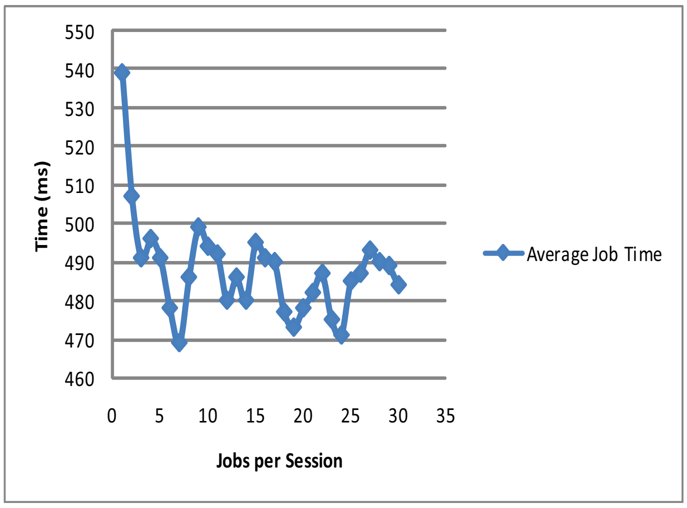


Figure 5.2: Job times when increasing the number of successive jobs per session.

The results showed that there are no clearly visible effects of caching. After two or three jobs per session for the same visualization the times were rather constant. With a total of 30 jobs per session the average was 488 ms with a standard deviation of less than 13 ms. With these results in mind we considered it to be safe to use 20 jobs per session in the following tests to get an appropriate average value for our measurements.

5.3.2 Network Interconnection Test

As mentioned earlier ParaView uses a message-passing library (MPI) to exchange data between cooperating nodes in a visualization. In earlier test setups we used a Fast Ethernet (100Mbit) network to connect the ParaView machines. However, we were interested in testing the influence of the underlying network on the performance of cooperating threads, mainly to prevent a network bottleneck to act as a confounding factor in our tests. A small test showed us that this connection is actually extremely important. For the test we (software-)limited the speed of the network card of one of the ParaView machines to 10Mbit, 100Mbit and 1000Mbit respectively (the latter being the actual limit of the network cards in our cluster). We used the medium dataset, set a default output resolution of 1024x1024 pixels and used a ’2-1’ distribution strategy. Finally, we had the number of cooperating ParaView threads increase from 1 to 12. The actual distribution is not really important for the outcome of the test but with the ’2-1’ strategy we should notice network bottlenecks after every third thread that will get added. Measurements of the pipeline initialization time for each of the network speeds and for each number of cooperating threads is shown in Figure 5.3.

When a pipeline is initialized all data for the visualization is loaded into memory and distributed among all cooperating nodes. This distribution is managed by the master node and any node that is on a distant machine will require transfer of data over the network to that machine. When two distant nodes are cooperating in the same visualization we expect that data is transferred to both nodes over the same network connection at the same time.

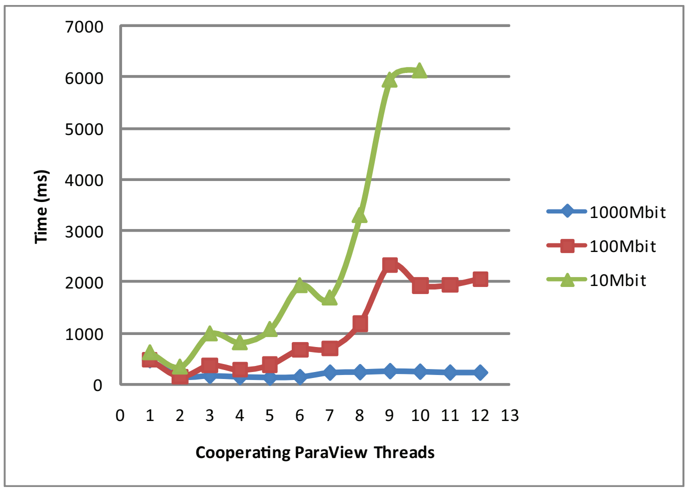


Figure 5.3: Test results showing pipeline initialization time with increasing ParaView threads at different network speeds (the last two runs of the test at 10mbit did not finish in less than 10 minutes and were therefore omitted).

The figure clearly shows that with 10mbit connections even a single node on a distant machine (i.e. distant from node0) will heavily degrade the performance of the system which means that the network connection is slower than the data distribution process. The 100mbit connection performs much better, until a second distant node is requested, requiring data to two nodes to be transferred over the network. At the highest speed the influence is not really visible, but note that our test setup only used two machines.

These test results have forced us to make sure that the machines in our test setup were connected at 1000Mbit. In fact, when a larger cluster is used in practice, it is advisable to connect the machines at even higher speeds, using technologies like Fibre Channel [5] (available at speeds up to 20Gbits), to minimize inter-machine latency and bottlenecks in network throughput.

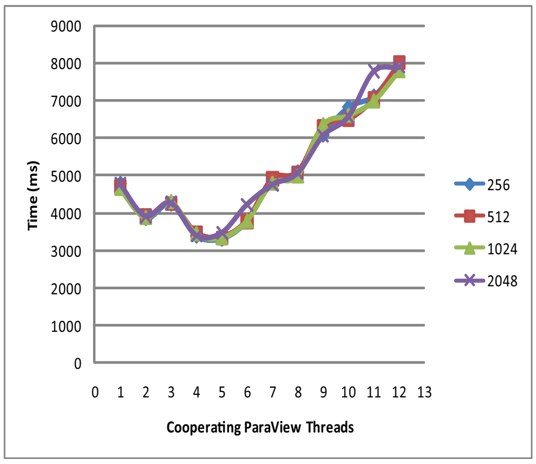
5.3.3 Output Resolution Parameter Test

During early tests we obtained some odd results in tests where the number of cooperating ParaView threads was gradually increased, starting at a single thread. Although the time required for pipeline initialization dropped when adding more nodes, the render time would often be increased when using two or more nodes (compared to using only a single node). After verifying possible causes of this behavior we found that the resolution of the output is an important factor in the process. To see whether the output resolution is of influence to the results we performed the following test. For both the medium and the large volume we ran the default visualization, while increasing both the number of cooperating threads and the output resolution. For the resolution we started at 256x256 pixels and increased the resolution in four steps, up to 2048x2048 pixels (note that this means that the output of each next iteration contains four times the number of pixels). A ’2-1’ distribution pattern was used in the tests. The results are shown below.

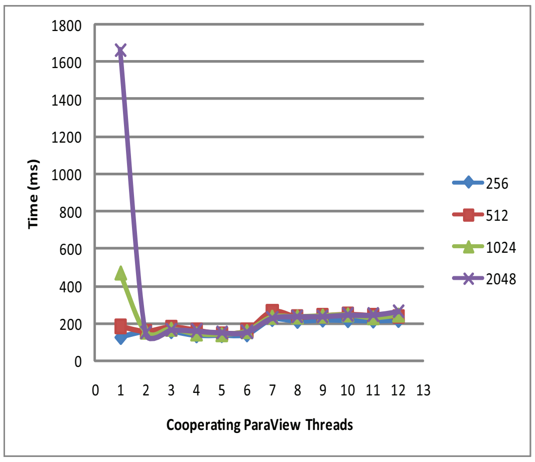
We see from Figure 5.4(a) and Figure 5.4(c) that the initialization times for both datasets decrease when adding (a limited number of) nodes, although the decrease is more prominently visible with the larger dataset. We examine this behavior in more detail later, but for this test we can say that the pipeline initialization is not related to resolution output (which means that an equal amount of data is loaded, regardless of the visualization and its output resolution).

Another conclusion can be drawn from the results which is way more important and relevant for the current test. The only case for which the output resolution was of significant importance was the single-node case. For both datasets for the lowest two resolutions a single node had a lower ’working time’ than any other combination of nodes. However, when the resolution was increased a very different behavior

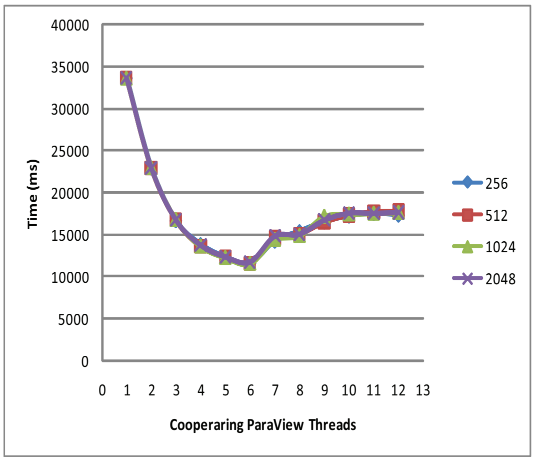
[5]http://www.fibrechannel.org/



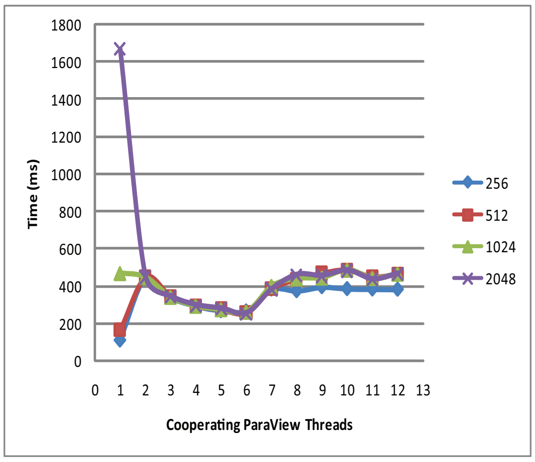
(a) Pipeline initialization times for the medium dataset.



(b) Job work times for the medium dataset.



(c) Pipeline initialization times for the large dataset.



(d) Job work times the large dataset.

Figure 5.4: Pipeline initialization (left) and job work times (right) for all resolutions.

appeared. The working time for the single-node pipelines scaled almost exponentially with resolution. When two or more nodes were involved in the visualization the resolution did not really seem to affect the times. Very likely, although not completely certain, this behavior was the result of a choice of ParaView for sort-first or sort-last volume rendering. Sort-first and sort-last rendering are choices for a system to distribute work among multiple processes. With sort-first rendering, the screen space is divided and each process contributes only to a certain screen area. With sort-last rendering the rendered objects are divided and the result is composited to a single output (see [12] and [18] for a discussion on sort-first and sort-last schemes for distributed rendering).

In short the conclusions of the results were as follows. For a combination of low resolutions and small-to-moderately-sized datasets a single node may in fact be the best solution. However, for most datasets combined with higher resolutions, both the lower initialization- and lower rendering times obtained by using more nodes will be an improvement. For the remaining tests we used a resolution of 1024x1024 pixels as default because compared to the other three resolutions the 1024x1024 resolution is the most appropriate choice considering the currently common screen resolutions for client workstations.

5.4 Classroom Scenario: Many Concurrent Clients

With the first scenario test in our test-set we tried to illustrate the behavior that is to be expected when the system is stress-tested using many concurrent client requests, like in a classroom with many students working at the same time.

The initialization part of the pipeline can be rather time-consuming as this part is dominated by data-loading and preprocessing. As the first part of this test, we were interested in the pipeline initialization times when increasing the number of concurrent clients.

Besides the initialization time we are also interested in the average job time once the pipeline initialization has finished, which we will examine in the second part of this test.

Note that this test was actually close to a worst-case scenario as each of the client threads submitted jobs without delay, while a real user would probably inspect intermediary results before requesting a new render. However, we did not ’cheat’ by submitting the jobs in a batch, but rather submitted a new job only after a previous job had finished, like in a real-life scenario.

5.4.1 Test Parameters

The parameter values for the test are mentioned below. We selected the medium dataset as subject for this test, as the large dataset would exhaust memory quite quickly, thereby reducing the number of tests possible. In this test we used the Locality strategy because the simplistic strategies we used earlier (’1-1’, ’4-2’, etc) have no memory and are therefore only applicable for distributing ParaView threads (which was perfectly fine for the tests above).

We explained the Locality strategy earlier in Section 4.3.2. In short the idea of the strategy is to assign server resources in such a way that multiple ParaView nodes will always be placed on a single machine (hence the term ’locality’) and that processes are divided equally among the servers (where ’equal’ means that the ratio between the number of physical cores and processes is similar for all servers).

- Dataset(s): medium
- Client thread(s): 1-40
- ParaView thread(s) per session: 1
- Jobs per session: 20
- Strategy: Locality strategy

5.4.2 Results

We started with a comparison of pipeline initialization times, using the three possible server combinations: the quad core, the dual core, and both machines combined. The results are shown in Figure 5.5. To be able to see in more detail what happens when the number of physical cores is exceeded we also provided two additional figures which can be found in Figure 5.6(a) and Figure 5.6(b).

From the test results we can draw several conclusions. From Figure 5.6(a) we can see that the initialization process was rather limited by the processor. If the initialization process was limited by disk I/O it is likely that we would have seen an increase in initialization times even when a second concurrent client would start using the system. However we should not directly generalize the fact that pipeline initialization times are constant as long as the number of parallel clients is below the number of physical cores. Because many different kinds of readers or preprocessing steps can be configured, results may be different.

What can be seen is that the strategy implemented in the middleware successfully distributed the clients over the available processor cores when possible, maintaining the ratio between the number of processes and the number of physical cores of the machines. In the test run where both servers were available to the cluster the initialization times remained (almost) constant until the number of clients equals the (total) number of physical processor cores. Another interesting property can be deduced from Figure 5.6(b) where the number of concurrent clients was greater-or-equal than the number of cores. Three linear trends were drawn over the results, along with their line equations and an additional value (R^2) showing the relative

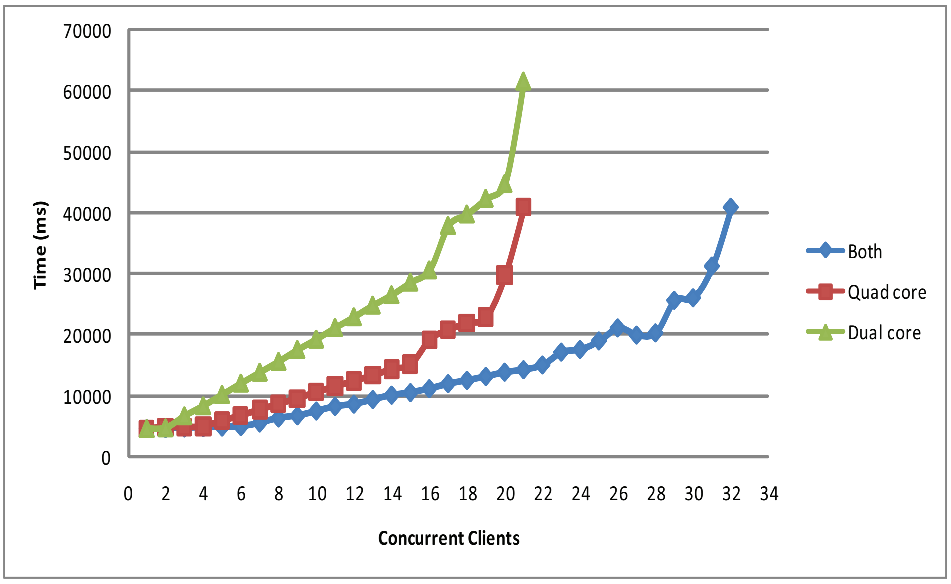


Figure 5.5: The initialization times for all three server combinations for the concurrencies for which we were able to get test results without errors due to timeouts.

error of the trend-line. The figure clearly shows that initialization times scaled 'only' linearly with the number of clients when all processors were in use.

In the case of our medium dataset the amount of memory required by each visualization pipeline was close to 130MB. Simple mathematics show that a machine equipped with 2GB of main memory will be capable of running approximately 16 of these pipelines simultaneously. After main memory has been used up, the operating system will start moving data and programs from memory to disk (a process called 'swapping') to free up memory for running processes. The overhead caused by swapping adversely affects the performance of the system and the result of this can be clearly seen in Figure 5.5 at the point where approximately 18 client threads are running concurrently. Again, the result of adding a server was very noticable: adding a second server with the same amount of memory will effectively double the total system memory, theoretically allowing twice as many client processes to fit in physical memory.

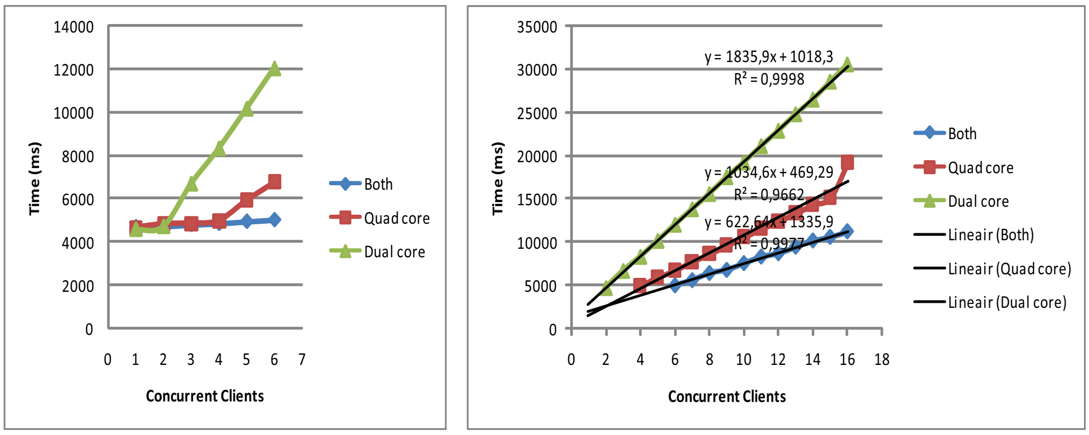
Finally, in Figure 5.7 one can see the job work times for the different configurations. At the beginning the trend of the job work times were very similar to that of the initialization times. Even when the initialization times were getting heavily affected by swapping issues the job work times kept scaling linearly.

5.5 Research Scenario: Visualization Parallelization

As we wanted to provide insight into the behavior of parallelization in our framework (i.e. parallelization within a visualization, in contrast to serving multiple clients in parallel as in the previous test) we performed several tests to see what would happen when more cores and/or more servers were added as cooperating nodes to a single visualization. This is a likely situation for example when highly-detailed information is to be made available to researchers. The datasets-to-work-with are expected to be larger but less clients are using the system concurrently.

5.5.1 Test Parameters

To show the effects of parallelization we limited the test to a single client running a single visualization job to give each job the full potential of the hardware. In addition to performing tests with increasing numbers of cooperating nodes, we were also interested in observing what happened when we used different node distribution strategies (to find out if there is a noticable difference between the '4-2' and '2-1' strategy).



(a) The part of the results where the number of concurrent threads is below the number of physical processors.

(b) The same results, but from the point where the number of concurrent clients exceeds the number of physical processors (up to 16 concurrent clients).

Figure 5.6: Pipeline initialization times for all the server combinations, with increasing numbers of concurrent clients.

Finally, we did our tests on both the medium and the large dataset to see whether any conclusions are generally applicable for different data sizes.

- Dataset(s): medium, large
- Client thread(s): 1
- ParaView thread(s): 1-12
- Jobs per session: 20
- Strategy: Simple strategy ('4-2', '2-1')

5.5.2 Results

After running all tests and comparing the results we noticed there is hardly any difference between the '4-2' and the '2-1' strategies. The only noticable differences occured at pipeline initialization times where the '4-2' strategy shows a steeper descent than the '2-1' strategy, when increasing up to 5 ParaView nodes (at six cooperating nodes the two strategies only differ approximately 100ms which is not visible from the diagrams). Because of this indifference we only show results for the '2-1' strategy.

In Figure 5.8(a) and Figure 5.8(b) the results are shown for the medium dataset. Clearly visible is that pipeline initialization became significantly lower when adding ParaView nodes. At approximately five cooperating nodes a minimum was reached which is close to the number of physical cores available in our setup of the framework. With the job work times we observed the same behavior. Note that the large decrease in time from one to two nodes is a result of the same effect as discussed earlier in the resolution test. Although the work times also kept decreasing up to five parallel threads it can be argued whether the larger number of threads can be justified for a dataset of this size.

For the large dataset the results (Figure 5.9(a) and Figure 5.9(b)) were better, from our point of view. To illustrate this we will consider the initialization times. For the medium dataset the initialization times dropped from 4708 ms to 3186 ms when increasing from one to five nodes: a decrease of approximately 32%. However, the times of the large dataset dropped from 33487 ms to 11748 ms when increasing to six nodes, a decrease of almost 65%. For the large dataset the job work times decreased until six threads were cooperating on the visualization.

For both datasets we can conclude that both the initialization and the work times increase rapidly when more cooperating processes are used for a single visualization than there are physical cores available.

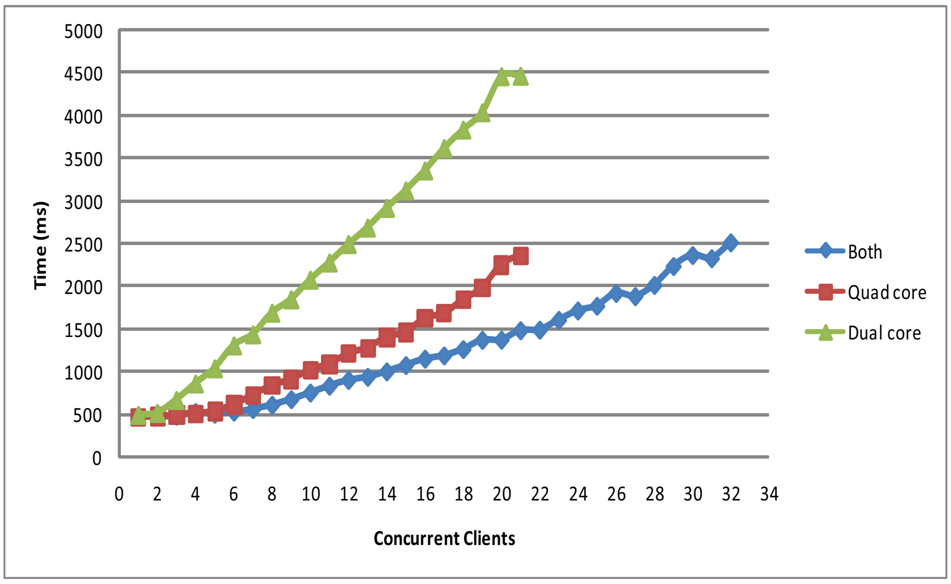


Figure 5.7: Average job work times with increasing numbers of concurrent clients working on the medium dataset, where each session uses a single ParaView thread.

5.6 Additional Tests

5.6.1 Memory Footprint

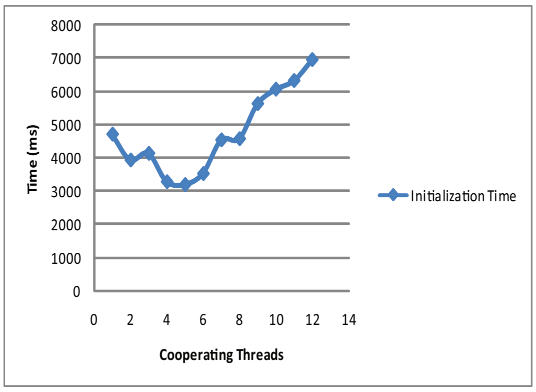
When rendering in parallel or distributed mode (i.e. using multiple nodes for each visualization pipeline) we expect ParaView to divide data among the available nodes. When running multiple pipelines simultaneously (i.e. multiple clients using one node per client) we may observe a slightly different behavior. To gain some insight into memory usage in these situations we performed some tests on both the medium and the large dataset. First we tested the memory load when increasing the number of clients requesting visualizations where each client got a single ParaView thread. In the second test we used a single user while increasing the number of nodes cooperating on a single visualization. Note that these two tests are almost equal to the classroom scenario and the research scenario we discussed above. The only difference is that at the end of the visualization we took a snapshot of the running processes on both ParaView machines (we took the snapshot using a well-known Unix-based process viewer tool called 'Top').

Several memory-related metrics of the running processes can be obtained from the snapshots[6]. The first idea was to use the total size of the memory image of all the processes (this value is called 'VIRT', short for 'Virtual Memory'). However, this value is rather system-dependent. For example, this includes a portion of memory for the process, called 'SHR', which is the amount of memory that is sharable (which is not the same as 'shared') among other processes running on the same machine. As it is difficult to measure which part of this memory is actually resident in memory we decided to use another metric.

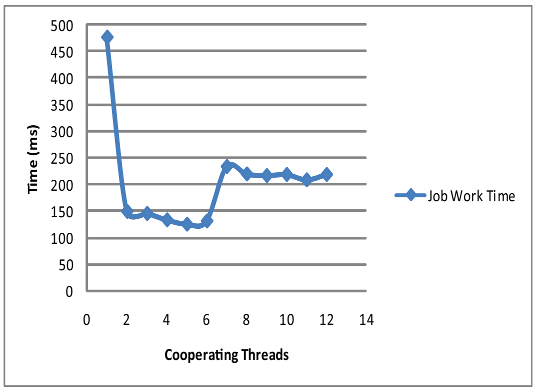
A more precise metric for comparison is the 'DATA' metric which solely includes the amount of memory devoted to program data and -stack (the amount of memory needed for executable code is excluded from this value).

In Figure 5.10 we combined the results for each of the datasets and for both tests in a single diagram. It is clearly visible that in parallel rendering mode (i.e. an increasing number of nodes) the data was distributed over the nodes, with an overhead of approximately 28 MB per additional node. If we compare the results of the medium and large datasets for this same mode we see that the lines are exactly parallel which leads us to the conclusion that the overhead is caused by ParaView (probably due to some additional bookkeeping), and not related to the size of the dataset we were working on.

[6]For a list of fields in the output of Top see `http://man-wiki.net/index.php/1:top`

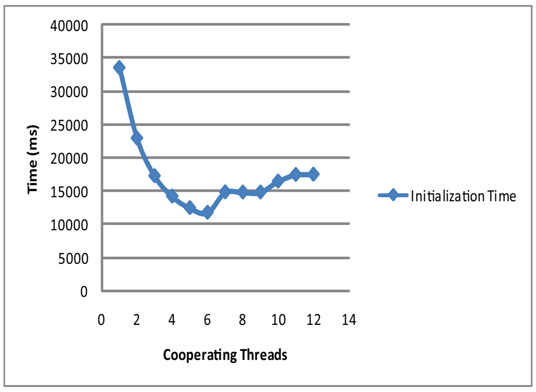


(a) Pipeline initialization times when increasing the number of ParaView threads.

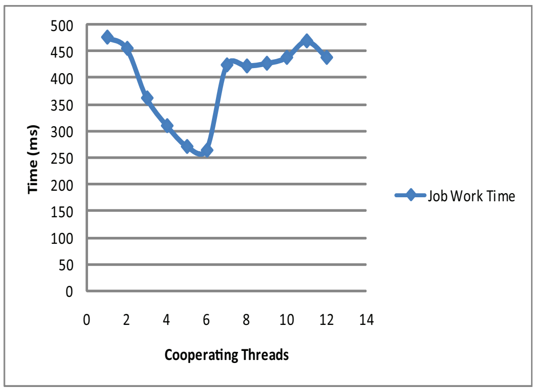


(b) Average job work times when increasing the number of ParaView threads.

Figure 5.8: Pipeline initialization and job work times with increasing numbers of ParaView threads working for a single client on the medium dataset.



(a) Pipeline initialization times.



(b) Average job work times.

Figure 5.9: Increasing numbers of ParaView threads working on the large dataset.

When considering the test with increasing numbers of clients we can conclude that the memory footprint required scaled linearly with the size of the footprint of a single (master) process. In fact the results of the test show that not only the 'DATA' metric of the snapshot increased linearly but all other memory-related values increased linearly.

5.6.1.1 Discussion

We can see from the test above that memory can be exhausted quite quickly when serving several clients concurrently on large datasets. Also, in our results for the classroom scenario test, specifically Figure 5.5, we see that we can reasonably serve approximately 30 concurrent clients using the Locality strategy with two servers. But from the same figure we see that both machines are independently capable of serving up to 18 clients at the same time. In an ideal situation we would expect the linear scaling to continue up to the total number of clients the two servers are capable of serving independently, which would be 36. The answer to the question why this values differ lies in the effect that swapping has on the performance, combined with the fact that the Locality strategy tries to keep the best ratio between running processes and available physical cores. Compared to the dual core the quad core machine will have twice as much processes running and will exhaust its memory twice as fast (as both machines had the same amount of memory).

Although the Locality strategy shows promising results in our tests, it seems that we may improve

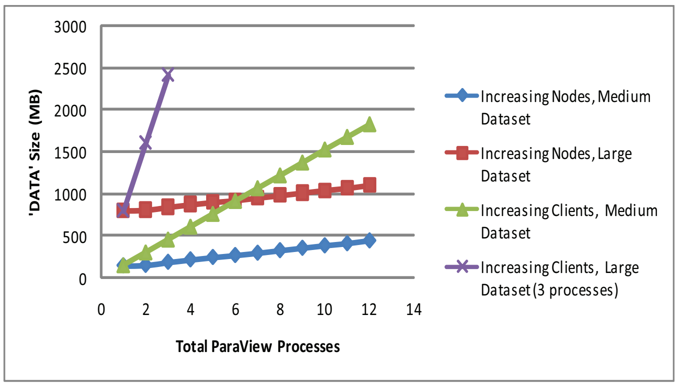


Figure 5.10: Total size of the 'DATA' components for increasing numbers of ParaView processes.

on the results of the classroom scenario (i.e. improve the pipeline initialization times, probably not the job work times, but that is difficult to predict). We could do this by creating a strategy which will not mainly base its decisions on ratio between running processes and physical processor cores, but rather on the amount of memory available on each of the machines. The best solution for this strategy would be to pick the server that has most (non-virtual) memory available at the time of starting the processes, but this requires the middleware to query the memory status of the available ParaView machines. An initial solution could use the ratio between total memory available on the machines: two servers with the same amount of memory should be running the same number of processes.

5.7 Conclusions

From the results of the tests performed above we can conclude the following statements:

- Network interconnection speed is a major factor in the effectiveness of distributed rendering.
- With a given number (>1) of ParaView threads cooperating in a visualization the performance will depend only on the amount of data and the combination of filters and rendering applied to the data, it will not depend on the resolution of the output.
- In a classroom scenario (with many concurrent clients) it turns out to be useful to add hardware to the system in order to serve more clients, as long as the clients are correctly distributed over the machines. Also, it is important to identify which distribution is most effective for a given combination of hardware and visualizations. Our results show that when no bottlenecks are present visualization pipelines hardly influence each other (no increase in initialization- or job times). Bottlenecks due to limited processor resources turn out to make the system scale linearly, while memory limitation bottlenecks make the system scale exponentially.
- In a research scenario (with many cooperating ParaView nodes per client session), it is also beneficial to add hardware to a system and both initialization- and job times will usually decrease when increasing the number of cooperating nodes. Due to the large amount of data that needs to be distributed among cooperating nodes it is useful to correctly motivate decisions for node distribution strategies.

Chapter 6

Conclusions and Future Work

In this thesis we reported on our project to create a framework for the remote 3D visualization of large datasets. In the following section we draw conclusions from our work and results and compare the final outcome with our objectives. We also discuss topics that are candidates for future work to this framework.

6.1 Conclusions

The framework we developed is designed around the concept of acting as a web service, to support remote visualization. We used SOAP (Simple Object Access Protocol) as message protocol for this service which is well-supported in many programming languages. Because web services use regular network transport mechanisms and are usually transparent to firewalls these service can be accessed by a user through any network- or internet connection. Experiments showed that our web service can be accessed from a local network as well as over the internet without notable effort.

Instead of implementing the visualization algorithms from scratch we chose to build our framework on an existing and proven visualization framework called ParaView. ParaView itself is built on VTK (the Visualization Toolkit) which includes numerous efficient and highly-optimized visualization algorithms. ParaView in turn adds parallel and distributed rendering capabilities to the VTK functionality and both VTK and ParaView are under active development.

To allow users to request visualizations, and to assign ParaView resources for those requests efficiently, we developed a middleware component that is responsible for interfacing client applications with ParaView while hiding ParaView's complexity from clients. In our project we investigated several options to establish communication between our middleware and ParaView and we chose to use a ParaView-enabled Python interpreter to do this. This Python interpreter is compiled against the ParaView framework and practically all ParaView functionality is available through it. Because each instance of the interpreter runs in its own process this solution creates a reliable solution for concurrently running pipelines: a problem to any specific ParaView visualization (process) will be restrained to that process only and will not interfere with other running visualizations. This solution was mentioned as the second contribution in our list of contributions in the introduction chapter of this thesis.

We aimed to create a solution that would not be limited to one specific dataset only. In the list of our contributions we mentioned our data management scheme which we dubbed a 'data encapsulation' scheme and which is modeled in a relational database. Our scheme allows comparative visualizations by allowing different modalities of a dataset to be defined. We also argued that data and visualization should not be seen as separate entities because the input to each visualization pipeline is determined by the data (of the modality). As illustrated in this thesis we tried to maximize the amount of re-use in our scheme by splitting the visualization pipeline into three steps, and, because our interface to ParaView is based on Python, it was practical to require a Python snippet in our scheme for each of those pipeline steps. The elegance of our solution is that the visualization routines appropriate for the available data are stored along with the data in a relational database.

A crucial task of the middleware component is to assign server resources to clients. As there are several ways to assign these resources we proposed and implemented the concept of ’strategies’. A strategy in our framework defines how many ParaView nodes will be started to cooperate in a visualization and how they will distributed over the available hardware. Two specific strategies were implemented and evaluated in Chapter 5: a Simple Strategy and a Locality Strategy (see the Future Work in Section 6.2 below for a proposal for an additional strategy).

For this project we also developed a reference client application, to enable use of our framework and to showcase it. Based on the currently recognized target audience we chose to design the application with a simple and intuitive interface. We purposely created the application in a flexible and extensible way to allow modifications and additions to the client application when needed, for example by applying the Model-View-Controller design pattern to ease the implementation of additional views. Note that, as a result of using SOAP as protocol to access the web service, the reference application can be easily replaced with a custom client implementation by anyone who wishes to do so.

In our description of our objectives for this project we mentioned that three visualization algorithms were considered essential for our remote visualization project: Direct Volume Rendering (DVR), surface extraction and rendering, and Multi-Planar Reconstruction (MPR). Because we used ParaView as basis for our framework we are capable of providing an extensive collection of visualization algorithms to the user, including those mentioned in our objectives.

In our objectives we also mentioned that we needed a scalable visualization solution. From our results in Chapter 5 (particularly the classroom- and research scenario tests) we can see that a suitable combination of distribution strategy and multiple ParaView nodes (either concurrently or simultaneously) is effective in allowing more clients to the system and reducing data loading times and rendering times.

We should also add some critical remarks to the scalability part of our results. Like we mentioned in Section 3.3.2 ParaView currently does not support streaming (piece-wise processing of chunks of data) in its visualizations and this is visible in some of the results we saw. In effect this means that we cannot handle arbitrarily large datasets. However, the results show that the memory footprint of a visualization scales constantly when adding more nodes to a single visualization. Therefore we could handle larger datasets by distributing the visualization over multiple servers with no signifcant (memory) overhead. Because of this property we can at least conclude that the total data requirement of a visualization is limited by the total size of the available physical and virtual memory until streaming is supported (more on this in the next section).

An open-source release of the source code of both the framework and the client application is expected after this thesis so that the framework- and client software can be used freely and that future work may contribute to the current foundation.

6.2 Future Work

The scope of this project is broad and inherently there are many things we can suggest that may be improved or added in the future. In this section we share our thoughts on future work on existing and new topics for our framework.

Support for streaming is high on the list of things requiring attention. When the visualization component gets the ability to process its data in chunks the size of the data for visualizations may become practically unlimited without running into the swapping issues we have seen in Chapter 5. However, as discussed earlier, recent progress in the development of ParaView shows that support for streaming is work in progress. Therefore it is likely that the solution is as simple as sitting-and-waiting for an upgrade to ParaView, which adds support for streaming to its extensive list of features.

In Chapter 5 we discussed advantages and disadvantages of the Locality strategy we implemented. We also stated that there is room for additional strategies that should base their distribution on other heuristics, for example a strategy that distributes processes in such a way that the negative effects of swapping are postponed for as long as possible.

Job results are stored in the database along with the visualization parameters used for the given job. It is possible to cache these results and to try to match incoming jobs against the already cached jobs to see whether we could skip the job and return the cached result instead. However, it is difficult to estimate the number of cache ’hits’ that will occur because visualizations often use floating-point parameters, camera angles for example. Another issue is that not all visualizations are reproducible (for example a visualization of a running clock) like we discussed in Section 4.5.6. The advantages of caching are obvious but certain aspects of the solution require further investigation to prove that caching is both beneficial and flawless in real-life situations.

The current support of annotations was primarily intended to provide the basic functionality (consider it a proof-of-concept) of overlaying information on a visualization. More work could certainly be done to improve the current implementation and things like sizing, placing and coloring of labels can be valuable future additions for the end-user.

In our implementation visualization pipeline objects are basically pieces of Python code which are assembled by the middleware in order to construct a complete pipeline. Although this solution is versatile and flexible it requires manual user programming and -editing which can be prone to errors. Future work could be done to develop an application to design and edit these pipelines graphically or to adapt existing visualization editors like DeVIDE [1].

Zoomify support in our client application could also be further improved. The current solution keeps the entire image in memory and all operations on the image (drawing to screen, improving resolution, etc) are done on the full-resolution image. When the client would support working with tiles it may be possible, useful for very large images for example, to reduce the memory footprint by keeping only those tiles in memory that are currently visible on the screen. Also, the current implementation fetches the tiles from left-to-right, top-to-bottom and from low-to-high-resolution. Improvements could start loading tiles in the center of the visible area and start working outwards.

ParaView supports movie-recording by defining animations and saving these using a given frame-rate and output format. Therefore, we could also support this, but future work is required to add the functionality to our framework. This work would include thorough investigation of the animation facilities in ParaView and finding a suitable way to transport results back to the client. Also it would be useful to provide a view to play the movie in the client application.

Current implementation of middleware and client will only exchange image data, either images or Zoomify images, as mentioned. For several reasons it could be useful to support polygonal data in our framework. This would allow polygonal modalities (examples of which could be iso-surfaces and annotations) to be drawn completely on the client-side. The reference client application views already support OpenGL (see Section 2.3.3) and creating a view for client-side hardware-accelerated rendering of polygonal data is straight-forward using OpenGL.

[1] `http://visualisation.tudelft.nl/Projects/DeVIDE`

Bibliography

[1] Mikula, S. et al., *Internet-enabled High-resolution Brain Mapping and Virtual Microscopy*, 2007

[2] Schneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 5th edition, 2009

[3] Stanton, E.T., Kegelmeyer, W.P., *Creating and Managing "Lookmarks" in ParaView*, IEEE InfoVis, pp.p19, 2004 IEEE Symposium on Information Visualization (InfoVis 2004), 2004

[4] Segal, M., Akeley, K., *The OpenGL Graphics System: A Specification*, Silicon Graphics, 2006

[5] Wills, G., *Linked Data Views*, Handbook of Data Visualization, chapter II.9, Springer Handbooks of Computational Statistics, Springer, 2008

[6] Baldonado, M.Q.W., Kuchinsky, A., *Guidelines for Using Multiple Views in Information Visualization*, 2000

[7] Galigher, A.E., Korloff, E.N., *Essentials of practical microtechnique*, Lea and Febiger, 1964

[8] van Zwieten, J.E., *Three-dimensional reconstruction from digitised microscopic sections*, M.Sc. Thesis, Delft University of Technology,

[9] de Haan, A.B., Willekens, B., Klooster, J., Los, A.A., van Zwieten, J., Botha, C.P., Spekreijse, H., IJskes, S.G., Simonsz, H.J., *The Prenatal Development of the Human Orbit*, 2005

[10] http://www.visible-orbit.org, *The Visible Orbit Project website*

[11] van Zwieten, J.E., Botha, C. P., Willekens, B., Schutte, S., Post, F. H., and Simonsz, H.J., *Digitisation and 3d reconstruction of 30 year old microscopic sections of human embryo, foetus and orbit*, in Image Analysis and Recognition, Proc. 3rd Intl. Conf. on Image Analysis and Recognition (ICIAR 2006) (A. Campilho and M. Kamel, eds.), vol. LNCS 4142 of Lecture Notes on Computer Science, pp. 636–647, Springer, 2006 Faculty Electrical Engineering, Mathematics and Computer Science, section Computer Graphics, 2005

[12] Verschuur, J.E., *Remote 3D Visualization of Digitized and Reconstructed Histological Sections*, Research Assignment, Delft University of Technology, Faculty Electrical Engineering, Mathematics and Computer Science, section Computer Graphics, 2009

[13] ParaView Homepage, *http://www.paraview.org*

[14] Law, C.C., Henderson, A., Ahrens, J., *An Application Architecture For Large Data Visualization: A Case Study*, 2001

[15] Schroeder, W.J., Martin, K.M., Lorensen, W.E., *The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization*, 1996

[16] Ying, Y., Huang, Y., Walker, D.W., *A Performance Evaluation of Using SOAP with Attachments for e-Science*, In Proceedings of the UK e-Science All Hands Meeting, 2005

[17] Moreland, K., Rogers, D., Greenfield, J., Geveci, B., *Large Scale Visualization on the Cray XT3 Using ParaView*

[18] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J., Favre, J. *Remote Large Data Visualization in the ParaView Framework*, 2006

[19] Liang, S., *The Java Native Interface, Programmer's Guide and Specification*, http://java.sun.com/docs/books/jni/download/jni.pdf, 1999

[20] Sahai, A., Graupner, S., Kim, W., *The Unfolding of the Web Service Paradigm*, 2002

[21] Haber, R.B., McNabb, D.A., *Visualization Idioms: A Conceptual Model for Scientific Visualization Systems*, Visualization in Scientific Computing, IEEE Computer Society Press, 1990

[22] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison Wesley Signature Series, 2002

[23] Brush, A.J.B., Bargeron, D., Gupta, A., Cadiz, J.J., *Robust Annotation Positioning in Digital Documents*, Proceedings of the SIGCHI conference on Human factors in computing systems p285-292, Seattle, Washington, United States, 2001