NTNU
Innovation and Creativity

User Interface for 3D Visualization with Emphasis on Combined Voxel and Surface Representation

Design Report

Runar Ylvisåker Lyngset

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Ketil Bø, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The objective of this work is to design an intuitive user interface for representing both voxel and surface models in scientific visualizations.

Assignment given: 20. January 2006
Supervisor: Ketil Bø, IDI

Abstract

The thesis presents a user interface design aimed at the scenario where a dual representation of a volume is desired in order to emphasize certain parts of a volume using surface graphics while the rest of the volume is rendered using direct volume rendering techniques. A typical situation in which this configuration can prove useful is when studying images acquired for medical purposes. Sometimes the user wants to identify and represent an organ using an opaque surface in an otherwise partly opaque visualization of the volume data set. The design is based on the visualization library VTK along with Trolltech Qt, a GUI Toolkit in C++. The choice of using VTK as a visualization library was made after evaluating similar systems. The report includes a state of the art chapter, the requirements for the system, the system design and the results achieved after implementing the design are shown.

Preface

This thesis is the result of the work of one student, Runar Ylvisåker Lyngset. It was written during the spring of 2006 at Institutt for Datateknikk of Informasjonsvitenskap at NTNU in Trondheim, Norway. This constitutes the final work of my Master of Science degree at NTNU. The system designed was requested by Trollhetta AS and teaching supervisor was Ketil Bø.

Trondheim

Runar Ylvisåker Lyngset

Contents

List of Figures

Chapter 1

Introduction

Volume visualization has become an increasingly active research topic the last decades and the result has been a number of groundbreaking volume rendering algorithms and techniques for volume visualization. These algorithms and techniques have since been subject to significant optimalization in order to achieve interactive or, ideally, real-time performance. The main improvement in frame rates has been through using special purpose hardware and migrating the algorithms and techniques to take advantage of the hardware available.

In the article "Volume Rendering in Medical Applications: We've got pretty images, what's left to do" a panel of key experts from well known visualization companies discusses what the challenges are in the field of volume visualization in the near future. One of the panelists, Bill Lorensen from GE Corporate Research states that although the core technology is well developed, volume rendering is still not used routinely in most hospitals. One of the reasons for this is that for the most part, user interfaces are complicated and not directly tied to the traditional techniques of the radiologist, such as 2D displays, filming and archiving.

Karen Zuiderveld states in the same article that radiologists are experts in reading 2D images and the use of volume rendering does not necessarily add to the diagnostic information unless the use of 3D techniques is fast and easy. In order to achieve an efficient use of volume rendering the technology needs to be seamlessly integrated into the diagnostic work flow[1].

1.1 Problem Description

Having a simple and intuitive user interface to volume visualization applications is vital for such systems to gain acceptance with a larger number of

user. This thesis seeks to design such a user interface that is easy to become proficient with and is focused on the core functionality. The potential use of the system is for making volume visualizations that are capable of displaying both volumes rendered using direct volume rendering techniques and the use of opaque surface geometry to emphasize internal structures of the volume with a certain property.

1.2 Challenges

A crucial step in the process of designing such a system is to find the right software libraries on which to base the design upon. Visualization libraries are vast systems with a lot of functionality and development at this level is beyond the scope of this project. The main goal of the design is to create a way of utilizing an existing visualization library in order to interactively create visualizations. A challenge of the design is to create user interface features that are intuitive to the user by ensuring that the system behaves the way the user thinks it will do.

1.3 Thesis Title and Description

The title of the thesis reads: "Design of User Interface for 3D Visualization with Emphasis on Combined Volume and Surface Representation". An attempt will be made to create a system for volume visualization which is capable of rendering volumes with a dual representation which can utilize well-established techniques for direct volume rendering as well as using polygonal surface graphics to render parts of the volume which are to be emphasized.

1.4 Solution Strategies

The work in this thesis has three major components, namely the process of finding the right library toolkits to use, the system design and the implementation of the system. In order to establish a foundation on which the design is created, different systems will be evaluated briefly to find good candidates for creating the system. The system will need two software libraries in order to achieve the desired functionality of a visualization system with a well designed user interface. Firstly, the visualization functionality of the system will require a software library in order for the system to achieve its primary goal. Secondly, the user interface needs to be based on a good GUI toolkit library in order to successfully creating an easy to use and intuitive user interface. The combination of the two libraries is dependent on components for bridging the gap between the two libraries to fully utilize the functionalities of the respective software libraries. Therefore, when different

systems are evaluated as candidates for the design, such a link between the two needs to exist.

1.5 Thesis Outline

The structure of this thesis is based on the three main parts of work associated with the project.

- The Background Theory (chapter 2) and State of the Art (chapter 3) chapters constitute the part of work related to the evaluation of different systems that provide the functionality of the system.
- Chapter 4 specifies the requirements for the system while chapter 5, System Design, presents the conceptual design and detailed system design including class design used in the implementation of the system.
- Chapter 6 presents the resulting implementation of the design by giving examples of uses and configurations for the finished system.

Chapter 2

Background theory

2.1 Volume Visualization

Volume rendering or volume graphics is a field in 3D computer graphics that deals with the representation and visualization of discrete elements arranged in a volume structure. This technique is very versatile and can thus represent any object because it samples the entire volume as opposed to 3D surface graphics which only represents points in space which are connected to form surfaces.

Volume graphics are based on arranging volume elements called voxels in a 3D array/structure. Voxels are popularly compared with pixels (abbreviation for "picture elements") as elements in a spatial representation while voxels are elements that represent a component of a volume. Although the "x" in the word "voxel" has no direct relation to the term "volume element" it is rather the result of the notion of a volumetric pixel that has led to the word "voxel".

A voxel can be represented in different ways and the fact that it is represented as a near cubic element does not imply that they are necessarily stores as such. Voxel structures are often generated from other datasets i.e. a stack of 2D images from a CT scan where the 2D image pixels form a single 3D voxel in order to create a volume structure.

2.1.1 Direct Volume Rendering

While other techniques such as iso-surface extraction pre-process the volume data set in order to produce images, direct volume rendering involves the entire data set as the "direct"-word may suggest. Being a 3D rendering technique implies that it is used to produce 2D images from a 3D data set. One of the advantages of using direct volume rendering is that it has the potential of displaying the complete data set. The technique relies upon color

and opacity transfer functions in order to map voxels with a certain intensity to a given opacity and color value. This means that every voxel in the volume is included in the process which implies that an obscured internal voxel may contribute to the final result. A disadvantage with the technique is that images produced may become hard to interpret because of internal structures obscured by other objects. It is therefore crucial to use carefully designed transfer functions that emphasize the desired features while omitting unnecessary information. Direct volume rendering is a memory consumptive and computationally intensive technique while dedicated hardware support has improved the performance of the algorithms used dramatically [2].

2.1.2 Iso-surface Extraction

As the problem with direct volume rendering is to distinguish internal structures of a volume, this is the mere purpose of iso-surface extraction. The technique involves connecting voxels of the same intensity and forming a polygonal surface representation of the structure. The contours are displayed as 3D opaque surface geometry which can be rendered on any hardware accelerated computer very quickly. Once the surface is generated, it requires no further processing, so it is a very fast and can display a large amount of data quickly once generated. The techniques has several disadvantages compared to direct volume rendering. Since the surface rendered is opaque at least to some extent, internal structure of the volume is often lost when displaying iso-surfaces. A possible way of solving this problem is to cut away parts of the rendered volume in order to display hidden structures. The iso-surface is generated from elements with similar densities inside the volume. It is important to realize that a structure with similar density not necessarily is a structure in the real world. Iso-surface can therefore be misleading and should be used with great care. Because iso-surfaces are usually rendered using a uniform material and color, it can be difficult to give them meaningful colors. However, for the purpose of emphasizing structures inside a volume, iso-surfaces have proved to be very useful [3].

2.1.3 Best of Both Worlds

For a system to fully take advantage of the two previous techniques, it should be capable of combining the two for the best results. Using direct volume rendering, the entire volume may be rendered given the opacity and color transfer functions. To emphasize internal structures of the volume, iso-surfaces can be generated to display the contours of similar densities. This approach is very useful in for instance inspecting medical data sets where an organ may be emphasized using surface geometry.

Chapter 3

State of the Art

This chapter gives a brief summary of the current state of the "art" of volume visualization systems. Below, examples of system that have found a way of approaching the visualization of volumes is presented briefly and a discussion on the different systems is given at the end of the chapter.

3.1 Graphics libraries

There are several systems that provide the relevant functionality for the system in design. The system will be implemented in C++ and among the available software libraries, the following four visualization libraries are evaluated.

3.1.1 VGL

VGL 3.2 is the latest release of the C++ Volume Graphics Library from the German company Volume Graphics. Is provides a 3D graphics solution for voxel data and the claim to have the industry's leading solution to the combination of voxel data representation and conventional 3D graphics representation [4]. This implies that the systems uses the advantages associated with using dedicated graphics hardware for higher rendering speeds.

VGL is a commercially licensed product and is used in the traditional volume visualization field of medicine in addition to being used by high tech branches like automobile and aerospace companies. VGL licenses are also offered to academic and research and development institutions at a discount.

Being a C++ class graphics library, VGL includes an extensive API which the company claims provides the most powerful set of features on the market. VGL offers a unique resource management in order to access large datasets on a consumer PC. It has a highly efficient memory access routines

and the system accesses the underlying data structures efficiently so that they do not need to be duplicated in the application using the library [5].

VGL can render any type of volume dataset and has optimized support for 8- and 16- bit voxel data types. It has no limits for how many datasets that can be rendered simultaneously.

When it comes to rendering techniques, VGL supports both software and hardware rendering algorithms. It claims to have ultra fast software renderers for both iso-surface generation, volumetric ray-tracing and MIP which is a form of texture-based rendering. VGL supports hardware accelerated rendering and both soft- and hardware rendered images can be shown in the same scene. Using hardware rendering makes it possible to use hardware shading features such as advanced material, lighting and shadow properties on voxel data. Software volume rendering is generally slower that hardware rendering and offers fewer possibilities. Using hardware accelerated graphics enables the VGL user to keep up with the rapid changes is hardware and take advantage of new technology for producing photo realistic images. VGL uses native OpenGL and is therefore able to utilize OpenGL hardware fully and incorporate its features with its system without being dependent upon any other 3rd party system. Figure 3.1 shows a mummy data set visualized using VGL demonstrating the high image quality obtained using the system.

VGL's highly efficient memory management and rendering techniques makes it possible to support time dependent visualization, so called 4D imaging techniques. This enables the user to watch a sequence of rendered images interactively to study processed rather that just a state of a system. This feature is becoming increasingly important in medical application where it can be used to study how an organ works by observing live images.

VGL is a native C++ library that can be used to develop applications in Windows, Linux and MacOSX. Support for other operating systems is also available upon request. VGL is designed to work on Windows with Microsoft Visual C++ 6.0 and Visual C 7.0, Intel C++ 6.0 and 7.0. On Linux VGL supports GCC 2.95.x or Intel C++ 6.0 and 7.0. On Mac VGL supports GCC 2.95.x. VGL can be developed using most GUI toolkits that are OpenGL enabled. Examples of supported toolkits are Qt, MFC and X/Motif [5].

3.1.2 Coin3D / SIM Voleon

Coin3D is a C++ 3D graphics library from the Norwegian company Systems In Motion. It is a collection of software libraries built to exploit the capabilities of OpenGL. Coin3D is a high level library to simplify the development

Figure 3.1: Mummy data set rendered using VGL

process for creating advanced graphics and visualization systems. The Library is a class library based on a scene graph originally designed in the Open Inventor API from Silicon Graphics. Coin3D is based on the Open Inventor 2.1 API and is a retained mode system [6]. Retained mode is a programming model for 3D graphics where the representation of objects, their spatial relationships, their attributes and positions are held in memory and managed by a library layer. This creates an abstraction useful for the programmer creating a system not having to individually manage every object's loading, managing, culling or rendering [7]. Coin3D being a scene graph based library means that it is based on an object-oriented data structure called a scene graph. The concept of a scene graph it to arrange the logical structure of a graphical scene and often spatial relations between objects according to a predefined model or hierarchical tree structure. Nodes in a scene graph may have many children but have usually only one parent node. This is a very useful and efficient way of arranging objects in a scene for graphics systems. When a change is made for a given node, the changes will affect every child of the node because they have inherited its parent's attributes. This also works for geometrical transformations because all transformation matrices for a given level in the scene graph may be concatenated for performing less costly matrix multiplications for a given sub-tree in a scene graph [8].

SIM Voleon is an add on package for volume rendering to be used with Coin3D. The package is easily integrated into a Coin3D application and is built on the existing scene graph technology. The volume rendering techniques used in SIM Voleon are texture based and cover both 3D- and 2D textures. The 3D texture technique is done by using viewport aligned texture slices while the 2D texture uses object-aligned slicing to render the volume. The 3D texture approach requires dedicated 3D graphics cards that support 3D texture mapping and produces image with the best rendering quality. By default the system will check whether the user's graphics is capable of handling 3D textures. If not, the system will fall back to the slower and more memory consuming 2D texture approach. To handle large data sets the volume rendered is divided into optimal sub-cubes for higher renderings speeds of large data sets. The system has dynamic color lookup tables for voxel coloring.

Since SIM Voleon is dependent on Coin3D to work, it also shares all its properties and can be combined to form a dual representation of voxel data along with polygonal surface graphics. It would therefore be a good candidate for creating a system for volume visualization using both direct volume rendering techniques along with iso-surface extracted surfaces. Since the purpose of this thesis is to design a user interface for such a system, SIM Voleon is a candidate for use as a base for the design. SIM Voleon supports direct object picking in the 3D view interface through an interaction object

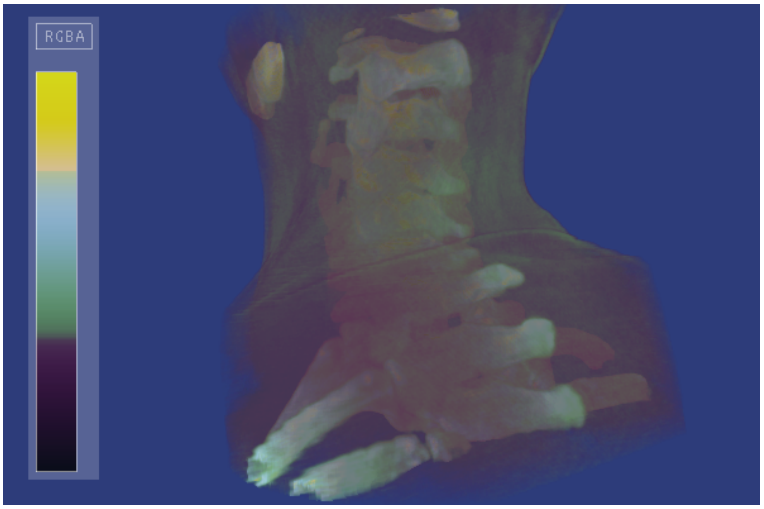


Figure 3.2: A volumetric data set of a human spine visualized using SIM Voleon.

that maps the input from the user's input device. This is very intuitive and useful if the system will be used to model and edit the object's transformations in the scene. Currently SIM Voleon only supports loading of the VOL file format. Support for other formats can be achieved by extending a class in the library called SoVolumeReader [9]. Figure 3.2 shows a spine data set rendered using SIM Voleon.

3.1.3 OpenGL Volumizer

OpenGL Volumizer is a volume rendering API from Silicon Graphics to make available advanced 3D graphics features for application developers such as 3D texture mapping and hardware supported transfer functions. The purpose of this library is to take advantage of features supported on OpenGL-based system in addition to other components that are essential in designing applications for volume visualization [10]. It is a high level interface to volume rendering technology which can be extended with other system to fully take advantage of its capabilities. The system is highly flexible both when it comes to low level services and utilities for high-level operations. One of the most useful features that can be utilized in modern graphics hardware is 3D texture mapping, a technology that enables hardware acceleration for volume rendering in order to achieve real-time rendering speeds and high quality images. OpenGL Volumizer has support for combined representation of direct volume rendering with opaque surface geometry. Volumizer can also render multiple volumes and utilize hardware enabled shading features such as custom shaders through a high-level interface. For large datasets the sys-

tem uses data-paging, memory management and graphics resource control to handle multiple resolutions in order to handle datasets that are too large to fit in main memory. The system is scalable for parallel rendering on multiple graphics pipes and the system is thread-safe which allows the system to run on multiple processors [11].

While system performance it optimized for SGI Onyx (R) class systems, OpenGL Volumizer can be used on both 32- and 64 bit Linux and Windows OpenGL-based systems. Figure 3.3 shows a data set rendered using OpenGL Volumizer and demonstrates the improvements of using hardware enabled shading features in volume rendering.

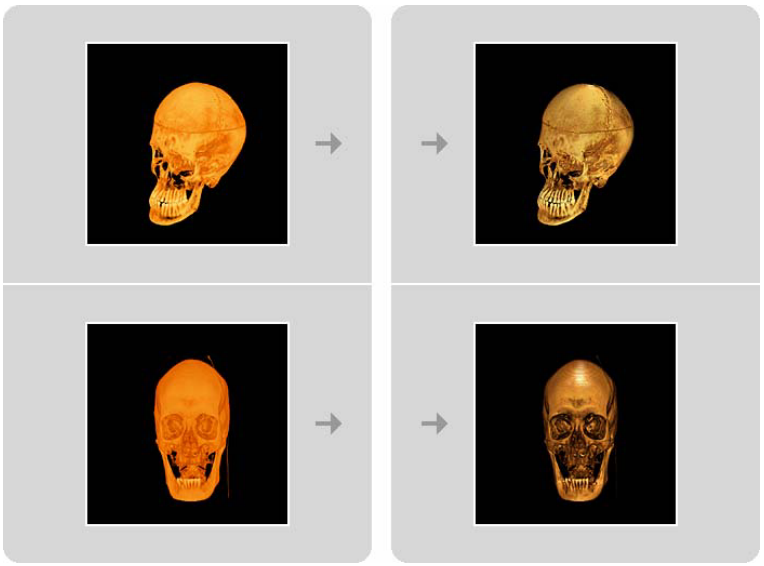


Figure 3.3: Figure shows the improvements when using hardware shading in volume rendering as opposed to traditional volume rendering.

3.1.4 Visualization Toolkit (VTK)

The Visualization Toolkit (VTK) from Kitware Inc. is an open source object-oriented C++ library for computer graphics, visualization and image processing. VTK is a large and complex system which involves a lot of functionality and different techniques. Since it is an open source library, the development of the system has been shared between hundreds of individual contributors and Kitware. Although it is a huge system and hard to get an overview the concept of building a visualization in VTK is easy to learn and is consistent throughout the library. This enables the VTK user to use advanced functionality once he has learned the basics of making a program [12].

VTK can be used for development using the interpreted programming languages Tcl, Python and Java. This is accomplished through "wrappers" that enable the interpreted language to use VTK code compiled in C++. This makes developing applications in VTK available to many developers who master these programming languages. The VTK system can therefore be represented as to basic subsystems: the compiled C++ core and the interpreted wrappers. The user can select which languages to include wrappers for if he wishes to use this functionality. If the user is a proficient C++ developer, there is no need to build support for other languages. The VTK library can be built for several platforms and programming languages using the open source cross-making tool CMake also from Kitware. Using CMake, VTK can make projects for a range of compilers for both Unix and Windows systems. Examples of supported compilers are: Borland, MinGW, MSYS, NMake, Unix makefiles, Visual Studio 6, Visual Studio 7(2003), Visual Studio 8(2005) and Watcom WMake. After making the projects, the VTK library files and assemblies can be built using the selected compiler. There are also built in support for popular C++ GUI Toolkits that can be selected before building the library. Examples of such toolkits are MFC and Trolltech Qt [12].

The VTK system is based on two models, namely the Graphics Model and the Visualization Model. The first of the two, the Graphics model is responsible for different components necessary for creating the images in the scene. Examples of such components are actors, props, lights, cameras, properties, mappers, renderers, render windows and render window interactors. Props are the things which are seen in the scene. An vtkActor is a subclass of vtkProp3D. Similarly, vtkVolume is also a subclass of vtkProp3D if we are doing volume rendering. vtkLight object may be used to control the illumination of the scene, but are not required to create a scene, because default light sources are always defines implicitly if none is specified. Camera objects control how an object in the scene is projected onto the screen and can be positioned in the scene to render a 3D scene. 2D scenes do not need a camera to specify its image projection. A renderer and render window object is also needed to produce an image on the user's screen based on the graphics engine. Interactor object are used in conjunction with a render window to handle user input which corresponds with what is projected onto the screen. While the Graphics Model is responsible for producing images from graphical data, the Visualization Model's responsibility is to transform data sets or information into graphical data [12]. Figure 3.4 shows a scene rendered from a head data set using VTK.

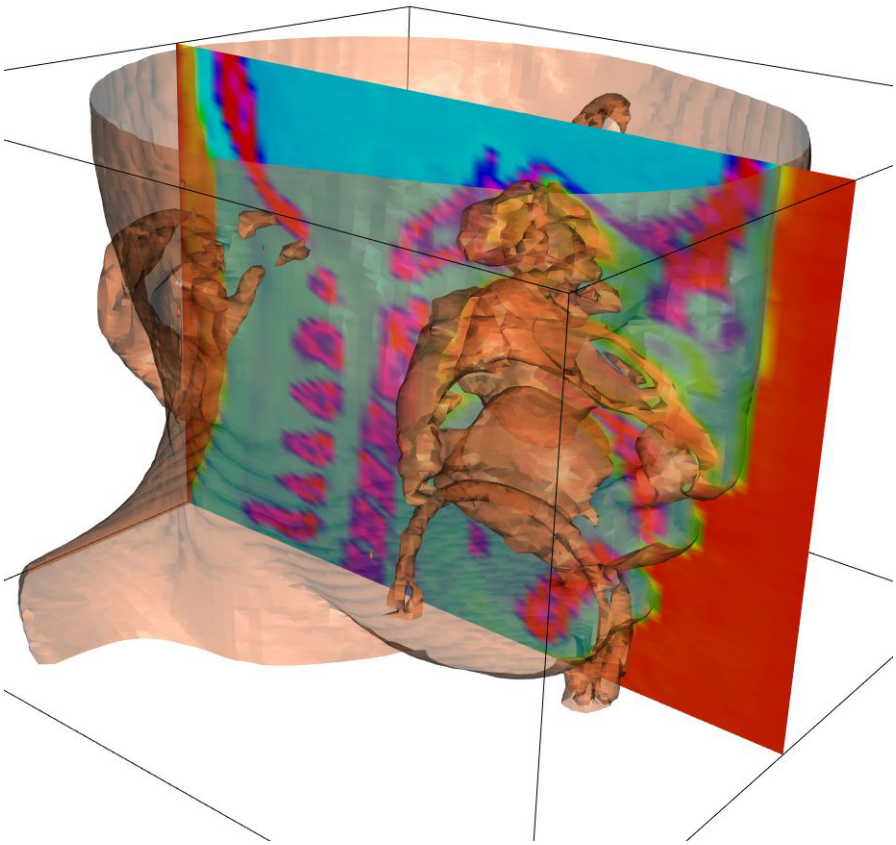


Figure 3.4: Head volume rendered using VTK. Skin rendered as iso-contour with saggital cross slice view rendered with color lookup table.

3.2 Graphics Libraries Discussion

The four visualization libraries above are all well designed and acknowledged systems. All systems support both Unix and Windows platforms and are all implemented in C++. The benchmarking of performance and image quality of the systems are out of the scope of this thesis, because of commercial licenses and special hardware required. But judging from the information provided by the companies that make them a notion of what their main focus is can be found.

The most commercially oriented and system that focuses the most on volume visualization in the industry is VGL from Volume Graphics. Their vision is to provide the highest image quality with the best possible performance for commercial use, which of course, comes at price. VGL utilizes hardware features in order to generate photo realistic images with hardware shading and lighting and this makes it a very competitive candidate if such a system was planned. Volume Graphics' showcase application, VGStudio uses Trolltech Qt which will also be used as GUI Toolkit for the design in this project.

VTK is an open source system which has a lot of functionality which is partly provided by the VTK community and owned by Kitware. The system is extensive and can create most known visualization techniques and is free for commercial use as long as the copyright acknowledgments are provided with the binaries and source code. VTK does not have the hardware specific features of VGL but has support for VolumePro hardware accelerated graphics cards for volume rendering. VTK is easily integrated with Qt and is a very good candidate for use in this design.

SIM Voleon does not provide a lot of functionality apart from basic volume rendering techniques and its support for file formats is very limited. Since this is a young system when it comes to volume visualization it does not compete with either VTK or VGL.

OpenGL Volumizer is a commercially available system which requires a commercial license for use. It is a technologically advanced system which focuses on cutting edge hardware technology on mainly Silicon Graphics systems but is also available on Linux and Windows platforms. The license costs make it less interesting for this purpose.

Chapter 4

System Requirements

The requirements for an application of this nature are based on what is to be expected of the system. A main criteria for a successful design of the system is that the system performs and behaves in a way that is intuitive and at the same time is capable of utilizing the underlying visualization engine.

As the title of the thesis suggests, the purpose of this system is to have a dual representation of a volume data set of both direct volume rendering and opaque surface rendering. By using a visualization library that is capable of rendering both techniques, this can be accomplished by selecting the right system components. In order to utilize the visualization library, however, an approach that is flexible and able to handle the visualization components regardless if they are voxel renderings or represent surfaces is suggested. The challenge is therefore to use the right software libraries and find a way to combine them to form a complete system that can utilize the capabilities of the visualization library and give the user an intuitive way of interacting with the system.

4.1 Functionality

The following specifies the functionality required in the system.

4.1.1 Program Characteristics

The program design is to consist of a visualization library with a visualization pipeline structure in order to manually edit this structure for interaction with the scene. The application's user interface will be created using a modern and efficient GUI Toolkit which can offer seamless integration with the visualization library. The internal state of the system will be represented using XML-files for easy editing, saving and loading of a pipeline network representation which constitutes the visualization. Interactive GUI widgets

must be designed in order to change the state of the system.

4.1.2 Surface Graphics Functionality

The system must be able to read, filter and display a number of different file types and polygonal data sets.

4.1.3 Volume Rendering Functionality

The system must be capable of applying common volume rendering techniques. An example of such a technique is volumetric ray casting which is performed on the volume using a color and opacity transfer function to map a specific color and opacity to a given iso-value. The transfer function must be editable in order to interactively specify the color and opacity values. In addition the system must include functionality for iso-surface extraction and display. The system must support the simultaneous display of both the techniques above.

4.1.4 3D View Navigation

The rotation, panning and dollying of the scene is to be intuitively arranged to perform in accordance with the user's expectations. The following arrangement is more or less standard in graphics and visualization systems.

- Rotation of the scene is accomplished through the use of the left mouse button while moving the mouse cursor.
- Panning of the scene is done by holding the middle mouse button while moving the mouse to move the view plane perpendicular to the view vector.
- Dollying or moving the camera closer or further away from the scene is accomplisher either by using the right mouse button while moving the mouse up and down or using the scroll-wheel on a mouse that has such a feature.

4.1.5 VTK Module Editing

- The system must have a feature that can represent the current state of the visualization pipeline as a directed graph.
- The connections between the modules in the pipeline must be visible to see the data flow in the visualization.
- The modules must be movable in order to arrange the graph in a visually satisfactory way.

- The modules represented as nodes in a graph must be selectable in order to show and edit the attributes for each module.
- The user must be able to add new modules and delete existing modules
- The pipeline graph must house a function for connecting the modules inside the widget as well as removing connections between modules.
- The attributes for a given module is to be represented in a way that is easy to edit and update.

4.2 Performance

The visualization library used for this design will need OpenGL enabled graphics card for rendering the scenes interactively. Modern visualization libraries use hardware accelerated graphics in order to display the scenes because of the speed this offers compared to 2D-rendered images. For the user to interact with the geometry or images produced, a modern computer with sufficient RAM and CPU will be required. The software should run effortlessly on a 1.7 GHz PC with 512MB RAM with en OpenGL enabled graphics card with a 64MB frame buffer.

4.3 Attributes

Satisfactory implementation of the following system attributes will be required in the design.

4.3.1 Portability

I order to cover the largest possible number of users and scenarios, the system will need to be independent of operating system and machine architecture. A system user is not to be forced to use a certain operating system in order to use the application. This can be realized through using a platform independent visualization library and GUI Toolkit. The code written for such platform independent systems is universal to all compilers and the code only needs to be compiled on the platform of interest using libraries built for the platform.

4.3.2 Correctness

The representation of the visualization pipeline must at all times show the real state of the system. In order to ensure this, all modules used in the system will need to be built into the system by creating specific code required for parsing the XML-document and instantiating the modules. This will in some cases limit the number of combinations of modules to connect, but an

advantage is that the system can handle the supported modules efficiently and not get a run-time error if an unsupported module is used.

4.3.3 Maintainability

The coding style used in the implementation must be consistent with the software libraries used. All classes created for the system must have the same prefix in order to identify them as members of the system. All attributes and functions must be documented well and C++ documentation is to be generated using the doxyGen system for creating HTML-documentation including descriptive diagrams. A clear and unambiguous XML-format must be used in order to easily add support for other visualization modules than the ones included in the implementation.

4.4 Design Constraints

Since the purpose of this design is to show a way of building an application around a visualization library, a limited set of features will be supported in the system. The subset of visualization features supported will be sufficient to show the idea behind the system and to create both simple and examples of more advanced visualizations. The system's focus will not be on distributed computing or parallel processing. Such considerations will therefore not be included in the design. The system will be run locally on a single processor and security issues are not taken into account as it is assumed that the system is run on a sane and healthy platform. The system will be a high level which will be based on the capabilities of the software libraries used.

To simplify the design in order to satisfy implementation time constraints there will be no functionality for undo/redo operations in the system. When a change is made and written to the XML-document there is no framework for undoing the operations performed. This might be incorporated at a later stage along with other advanced functionality.

Chapter 5

System Design

The systems used in the design are presented below and explanations why the were selected are given. The conceptual design of the system is described briefly with a more elaborate description of the system following below. To fully describe the design, the class design is presented last to provide the implementation details.

5.1 Choice of System Components

The design will be implemented in C++. Because visualization sometimes includes heavy computations and interactive response times, C++ is a natural choice of language because of its nature being a fully compiled programming language run from machine code. If a language such as Java would be used, it would come at a cost of speed and interactivity. This is the reason why most graphics libraries are written in C++. C++ is a object oriented language which offers a lot of flexibility and robustness if used properly and should therefore be a good choice for the implementation of this design.

The Visualization Toolkit (VTK) from Kitware will be used as the visualization library in the design. Even though there are alternatives that can better handle large data sets and have more advanced functionality, VTK should make a solid base to an application of this type because it is an open source library that is easy to use. It is a vast system and houses a lot of functionality. It's class library is very large and has support for most every known visualization technique known. It is continually maintained and extended with new functionality all the time. By using this library the system should be set for future updates and new functionality. The version of the library used in this implementation is the latest release at the time of the design which is VTK version 5.0.

Trolltech's Qt GUI Toolkit has been chosen for creating the user inter-

face for the application. Qt is a C++ cross platform Toolkit which includes a lot of functionality beyond the creation of windows and interaction. It has is own set of data type in order to ensure safe and efficient memory management and operation. Qt also includes a component for interacting with XML-documents, a feature which is very important in the design of this system. Since the system is designed around a visualization pipeline defined in an XML-file easy and efficient XML interaction is vital. The Qt XML sub-library includes two models for interacting with XML-documents, namely SAX and DOM. SAX is an event-based faster way of dealing with XML. DOM is a tree-based API which is more memory consumptive but has advantages when it comes to structuring an ease of use. Since the XML-documents used for the purpose of the design are of limited size, memory is not an issue, and accessing an XML-file using the Qt XML-module has shown to be both efficient and easy to accomplish.

A feature in Qt which simplifies the implementation of the user interface is the Qt Designer. The Qt Designer is an application for interactively designing windows by dragging components onto a canvas and specifying its attributes. Along with the designer a compiler called UIC is used for compiling C++ source code from the patterns created by the Qt Designer. Qt Designer files are XML-based files with .ui extension which are compiled into C++ header files which need to be included in the project in order to use the feature created in the Designer [13].

Another Qt-specific feature is the Meta Object Compiler which compiles the header files in a Qt project that contain Qt specific class definitions into robust working legal C++ code. The reason for this approach is that Qt has its own way of object interaction called signals and slots. This paradigm includes Qt-specific keyword included in the header files which need to be translated into legal C++ code before it can be compiled by the C++ compiler. Signals and slots is a very useful concept which simplifies the communication between objects. Signals can be emitted from any object that is included in the Qt-system. Likewise any object can contain a slot which then can be connected to a signal anywhere in the system. A slot can have any number of signals connected to it and parameters can be passed as a function call like normals function calls [13].

The components used to build a user interfaces in Qt are all derived from the QWidget class which provides the basic interaction features like signals and slots and the events and action handlers. When the term widget is referred to throughout this report the meaning will be a component which derives from QWidget and is used to provide functionality to a user interface in a Qt-based application.

These features makes Qt a very flexible and attractive library for creating applications with interacting windows and widgets. Qt is a cross platform library which can be compiled on Linux as well Mac and Windows machines and can along with VTK therefore be used by a larger number of users and platforms.

5.2 Conceptual Design

The system is designed using two well known and renowned C++ libraries, namely VTK from Kitware and Trolltech's Qt library. At a conceptual level the main features of this system is to utilize the existing VTK framework and features through a graphical user interface made using the Qt library. The purpose of the design is to create a paramount framework to manipulate the VTK visualization pipeline interactively. In order to store and retrieving information about the objects and attributes of different object in a scene, the system will use a simple XML-structure to represent objects in the pipeline and connections and relations between the objects. Through the use of this structure VTK object will be instantiated and connected in order to create the desired scene in the application's main view implemented using Qt. The main system components are shown in figure 5.1.

For the user to be able to edit the pipeline, a set of interactive widgets are needed to register input and to show the contents of the VTK subsystem in the window application.

The VTK visualization pipeline is created by instantiating object from the large class library and connect the objects together to form a procedural network to represent the pipeline.

5.3 User Interface Features

As most of the system will be coded, only the framework for housing the widgets are built using Qt Designer. The main features in the user interface are the following:

- Main Window: Main frame to house all functionality and widgets
- Main Menu: Menu from which files my be loaded and the visibility of the sub-windows of the application may be toggled.
- VTK View widget: This feature is the main view which displays the output of the VTK system and is included in the main window implemented as a widget.

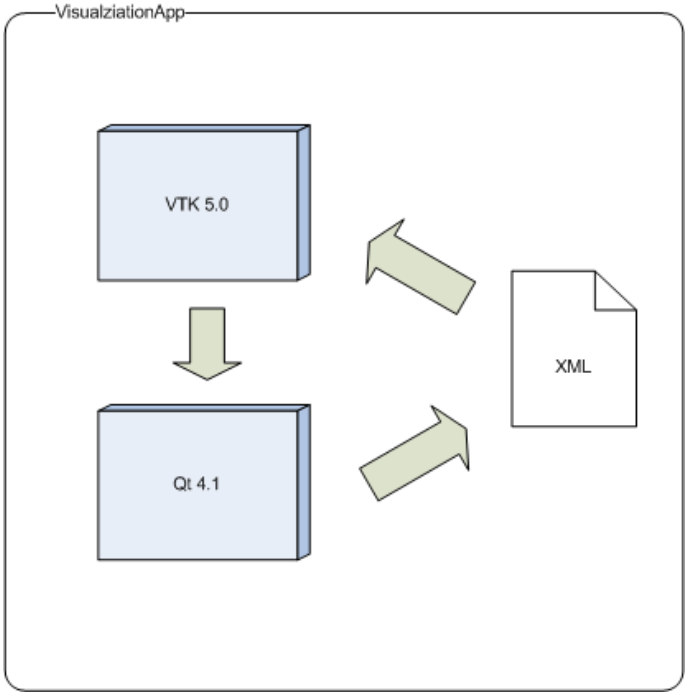


Figure 5.1: System Conceptual Design

- Pipeline widget: This widget displays the visualization pipeline consisting of VTK modules and the connections between them. It also contains functionality for editing the pipeline such as connecting and deleting modules.
- Settings widget: Shows the attributes of a selected module in a tree structure based on the contents of the XML-document describing the scene.
- Help browser: Provides a simple HTML-browser for a local help system.

5.4 Framework Created by Qt Designer

The application requires a main window which houses the widgets that contain the functionality. To ensure an easy to use system, the user interface will be a simple and sleek main window which only includes the necessary functions for the system to operate. The design is focused on making the user interface simple because the purpose of the system is to display the visualizations themselves and the other features merely tools in order to interact with the system. After compiling the .ui-files to C++ code the objects

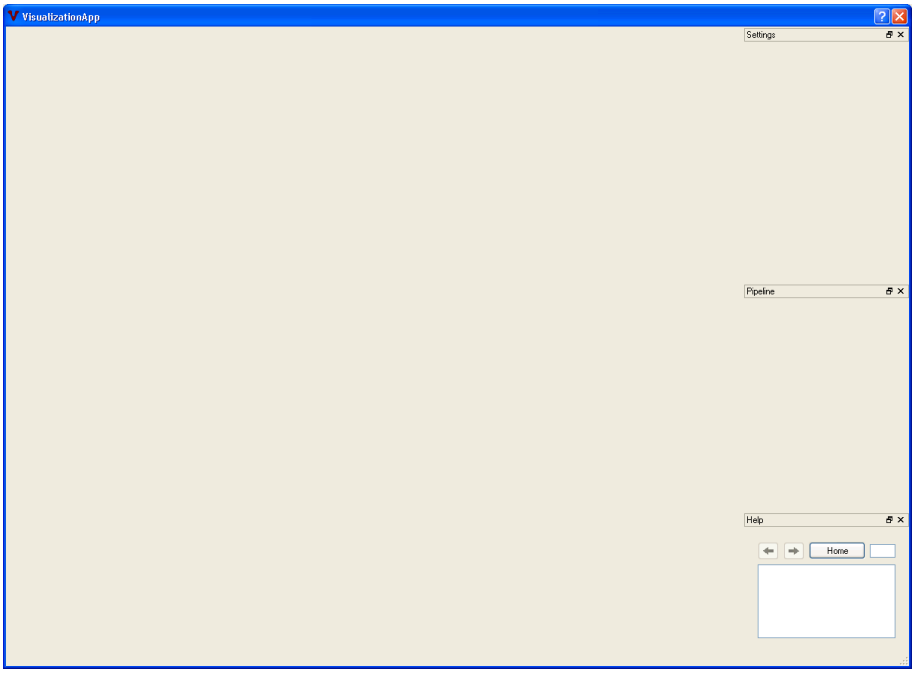


Figure 5.2: The user interface framework created using Qt Designer

created in Qt Designer can be access in code through including the header files generated by the UIC (User Interface Compiler).

5.4.1 Window Framework

Using Qt Designer the framework consists of the following

- Main window
- "Settings": dockable window to contain the functionality for altering a module's attributes.
- "Pipeline": dockable window for pipeline network widget.
- "Help": dockable help browser

The fact that all tool windows in the application are implemented as QDockWidgets means that they can be torn off from their initial placement inside the QMainWindow. When moved away from the sides of the main window they appear as floating windows over the main application as shown in figure 5.3. The dockable windows can be repositioned by the user on any edge of the main window that constitutes a DockWidgetArea. This feature makes the user interface highly flexible and user configurable. The dock widgets may be closed either by clicking the dock window's upper right

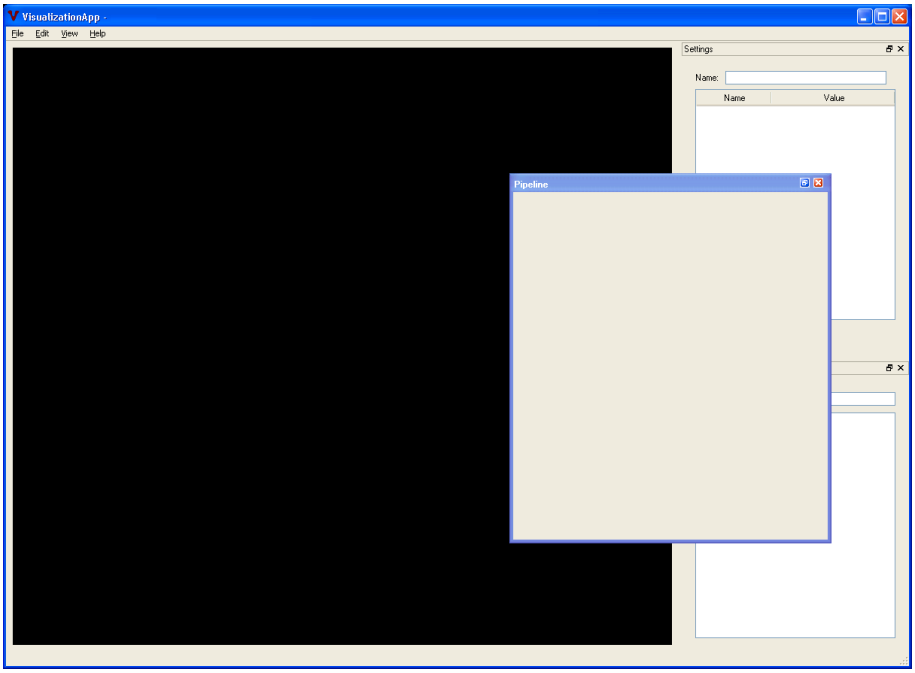


Figure 5.3: Dockable widget torn off the default dock area to the right of the main window.

close-symbol or the widget's visibility may be toggled from the main menu's "View" entry. The help window's visibility can be toggled from the "Help" menu item on the main menu.

When all dockable widgets are closed, the graphics view occupies the entire main window automatically using layout objects to control the size of the dockable widgets along with the graphics view which is also implemented as a QWidget. Figure 5.4 shows the main window while all dock windows have been closed. The size of the dockable windows are also resizable and easily adjusted using the mouse cursor to drag the frames of the windows.

5.4.2 Qt VTK Interaction Widget

The basic interaction interface between the VTK and Qt libraries is provided and distributed with the latest VTK release which is VTK 5.0. In order to use this feature in VTK, the library must be configured using the CMake application in order to include GUI support for the Qt library. When VTK is built, the necessary assemblies and library files are built and the programming interface is provided in the header file qvtkwidget.h located in the VTK file tree. The implementation of this VTK widget for Qt is created and copyrighted by the Sandia Corporation in 2004 for the U.S. Government.

Figure 5.4: All dockable windows are closed and the main graphics view now occupies the entire main window.

Redistribution and use in source and binary forms are permitted provided that the copyright notice in implementation and statement of authorship is included in all copies of the code. The QVTK widget provides access to the VTK "render window" and provides support for the "interactor" object that is used in VTK for receiving input from the user's mouse gestures. The QVTKWidget is included in the user interface by adding it to the main window in Qt Designer because it is implemented as a QWidget and has the basic Qt functionality built into it.

5.5 System Features

The conceptual design section explained the idea behind the system which is further explained in this section. The state of the VTK system is stored in a single XML-file which is formatted in a way that makes is easy to parse the nodes in the XML-document. The VTK library consists of plugins or as frequently referred to as modules in this report. There are four main types of modules, namely sources, filter, mappers and actors. The VTK visualization pipeline consists of such modules arranged in a certain sequence or parallel to each other. Sources are modules that provide some kind of interface to a data set and can only have another module connected to its output and have no input-module. A filter is a processing module that can transform

an input object and provide a transformed output object. A filter can have multiple inputs and outputs. A typical input to a filter is a source module or another filter. A mapper module transforms data objects into graphics data

5.5.1 Main View Functionality

The following items state the tasks of the main view class:

- User Interface setup including VTK view.
- Instantiate pipeline, settings and help widgets.
- VTK renderer and subsystem setup.
- Open scene files.
- Parse file contents and instantiate VTK objects (modules) and set their attributes.
- Connect the modules to form the visualization pipeline.
- Render the scene.

The task of the main view class is to include and instantiate the generated header file from the compiled Qt Designer file and also to handle all interaction with the VTK subsystem. The generated user interface file is included in the main view's class declaration and by calling the function setupUi() which is defined in the Designer-generated header file from within the main view's constructor the components are instantiated. In addition to setting up the generated user interface components, the main menu is created by defining the actions to take whenever an event connected to the menu entry occurs.

5.5.2 VTK as XML

A special XML-format has been designed for use in this system. The different module types have been separated by using their names as identifiers in their nodes in the XML-document. The XML-node's tag name is the type of module, namely "source", "filter", "mapper" or "actor". For further identification, the following other attributes are included in the module's signature:

- Description: A textual brief descriptive name.
- ID: A unique number of the type of module for identification.
- Name: The class name of the VTK module.

- yPos: The Y-position in the Pipeline widget.
- xPos: The X-position in the Pipeline widget.

To specify attributes and connections between modules two other type of nodes are needed inside the module's XML-structure. To specify attributes for each module, the node "property" is used to specify what functions are to be called on the object and the parameter passed in the function call. The pipeline connections are specified using the "connection"-node which is located in the node that is the input-node of the connection. The node specifies what node the connection is made from and to uniquely identify the node, the type of node and its ID is needed to define a connection. The reason why the "connection" node is placed in the receiver module is that the function call creating the object connections is performed on the receiver object in code. The parsing of the structure only needs the information about the type of module and its unique ID. For this operation it is vital that the ID is unique and that the IDs in the scene are increments from 0 for each of the four module types. The reason for this is that the ID is used directly as and index in the main view class in order to retrieve the module-object from its list when making a function call.

I addition to module nodes, a node named "renderer" holds information for the renderer which is scene specific. In the XML-example presented, the background color is specified in the "renderer"-node.

5.5.3 Import Data from XML-Structure

The main view class houses the functions that instantiate the VTK modules. The instantiation is accomplished through two basic functions that read and parse the XML-file in order to instantiate the objects declared in the XML-structure and calls the attribute functions and connect the modules together to form the visualization pipeline.The four type of modules have their dedicated lists that hold the modules' pointers in order to have a structure to manage the objects and structure them for later use. The parsing of the DOM-tree used to represent the XML-document is straight forward by matching tag names in order to separate the different types of modules and call the modules' specific functions that are located in the parsing function.

5.5.4 Instantiating VTK Modules

This means that every function defined in the XML-document need to have a corresponding function call somewhere in the parsing function in order for the system to work. The implications of this way of organizing the system are that when support for a new VTK module is added to the system, it needs to be included a number of different places in the code. The following

```
<pipeline>

  <renderer>
  <property name="background" argtype="double" arg1="0" arg2="0" arg3="0.4" />
  </renderer>

  <source description="file" id="0" name="vtkDataSetReader" yPos="0.1" xPos="0.1">
  <property name="fileName" url="Data/tensors.vtk" />
  </source>

  <mapper description="mapper" id="0" name="vtkDataSetMapper" yPos="0.3" xPos="0.1">
  <connection type="vtkDataSetReader" id="0" />
  </mapper>

  <actor description="actor" id="0" name="vtkActor" yPos="0.6" xPos="0.1">
  <property name="diffuseColor" argtype="double" arg1="1" arg2="1" arg3="1" />
  <property name="specular" argtype="double" arg1=".5" />
  <property name="specularPower" argtype="int" arg1="20" />
  <property name="opacity" argtype="double" arg1=".8" />
  <property name="visibility" state="true" />
  <connection type="vtkDataSetMapper" id="0" />
  </actor>

</pipeline>
```

Figure 5.5: Example of XML format used to describe a simple VTK scene. It consists of a file reader, a mapper to transform the input from the reader into the graphical data used by the actor. The scene created by this configuration can be seen in figure 6.4

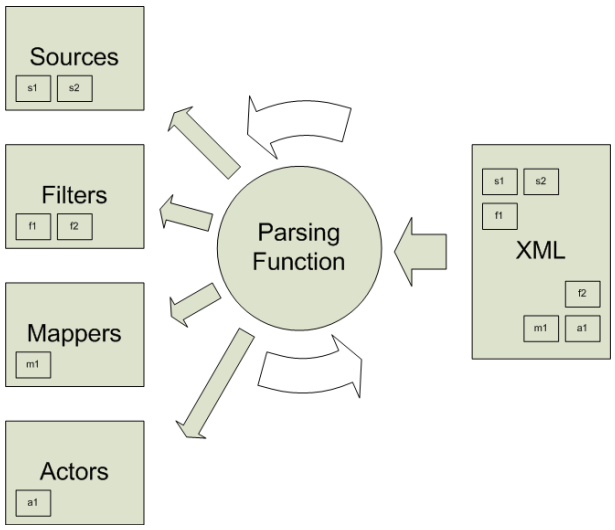


Figure 5.6: The parsing of the different types of VTK modules sorted and added to container-lists.

explains where each module needs to be included in order to support it in the system.

- The VTK module's header file located in the VTK library file structure needs to be #included in the main view's class definition.
- It needs to exist in the parsing function's main cycle in order to instantiate it and add it to the list of modules that represents its type. To be able to receive a connection from another module, an entry is needed in the section of the module from which the connection is received in the parsing function. This entry adds a new connection to the list of connections in the main view class. A connection is an object defined in the class VConnection, which is described below.
- An XML-template for each module used in the system is located in the file "template/ModuleTemplates.xml". This file is used when a new module is added to the scene and to generate the menu which is used when adding a new module. A new template entry for the new module is needed for the system to support it.
- In the function which connects the different modules the module name is needed in order to parse the contents of the list of connections. From this list of connections, connections are created after all objects have been instantiated to form the final visualization pipeline.

When the parsing of the modules is performed, the object that corresponds to the current element of the DOM-tree is instantiated and added to

the list of pointer that represents the module type. The module's attributes are also defined in XML and their corresponding functions are called with the arguments specified in XML.

5.5.5 Pipeline Widget

To represent the VTK visualization pipeline, a widget for accessing and designing custom pipelines is needed. The pipeline widget is contained within a dockable widget and is therefore very flexible when it comes to placement and size inside the main window. The purpose of the widget is to show the pipeline as a connected graph with the opportunity to select and move the nodes within the widget and to connect the different nodes by dragging an edge between two nodes. Each VTK module is represented as nodes in the graph and the four types of modules, sources, filters, mappers and actors are given a unique color in order to separate the modules' functions. The modules are represented as rectangular boxes with two areas at the top and bottom that act as output and input areas for interconnecting the modules. To indicate the direct of data flow in the network shown, arrows are drawn in these input/output areas. The VTK system uses a lazy evaluation scheme for updating the pipeline [12] and the updating of each module is cascaded like shown in figure 5.8. This figure also demonstrates the direction of data flow in the pipeline.

Tool tips are shown when the mouse cursor is left idle over a module to show what type of module it is. This feature can be expanded to include more information about the module like help information or detailed information about the modules functionality.

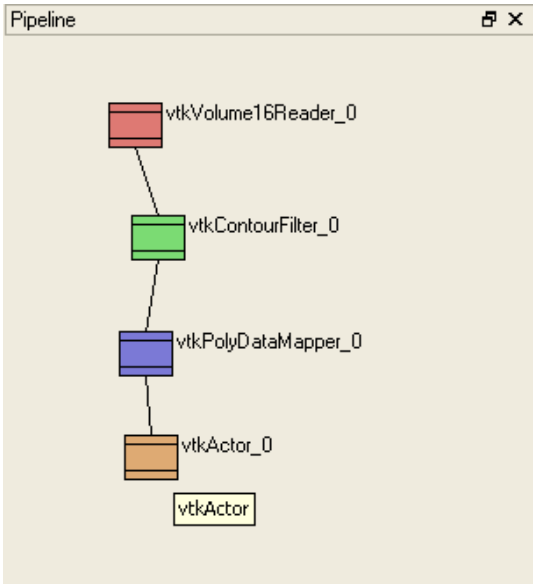


Figure 5.7: The pipeline widget showing the VTK visualization pipeline.

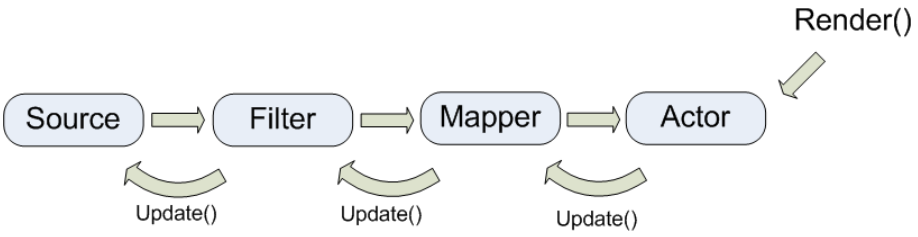


Figure 5.8: The direction of data flow in VTK starting with the source, then filter, mapper and actor last.

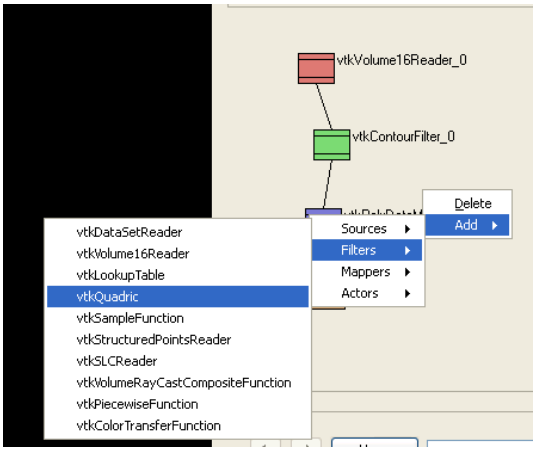


Figure 5.9: Adding a new module to the pipeline using context menu activated by clicking right mouse button.

To add a new module to the network the user needs to activate the add-menu which can be found either in the menu or more intuitively as a context menu which is activated by right clicking the mouse inside the pipeline widget. This menu holds all available modules that are built into the system and is generated from the XML-file "ModuleTemplates.xml" which is included in the "template" folder in the project directory. The modules in the menu are divided into sources, filter, mappers and actors to clearly indicate what their function is and to avoid a large list of names that is difficult for the user to grasp. The operation of adding a new module is shown in figure 5.9. To delete a module, the context menu is activated by right clicking the module the user wishes to remove and select "Delete" from the menu.

Connecting the modules is performed by dragging a link from a modules output area to another modules input area. While the operation is in process, the link between the modules is painted in red and a textual instruction is given saying: "select input" at the position of the mouse cursor. Figure 5.10 shows the operation of connecting two modules in the pipeline widget in process. If the user wishes to remove a connection to a module can click the input module's input area to remove the connection.

5.5.6 Settings Widget

The Settings widget is a display and input widget that shows the attributes of each module activated in the Pipeline widget. It is an interface to the XML-document which defines each widget and its attributes. It displays each attribute or property in the XML-scheme as node in a tree. The user can select the desired attribute and change its value by activating the text

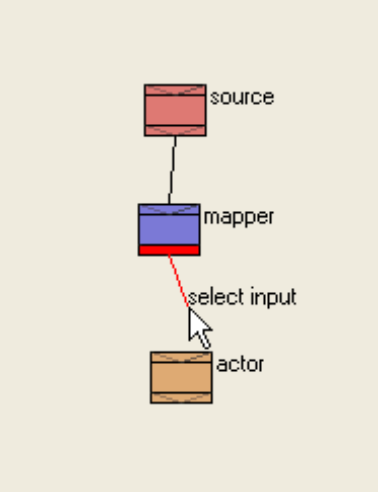


Figure 5.10: A connection is made between a mapper and an actor module.

fields in the tree view and write the attribute's new value. The new value is then written to the XML-document which is then sent to the main view to be parsed and the function call associated with the attribute is called. The widget is implemented using Qt's model/view programming paradigm which is a simplification of the MVC (model/view/controller) paradigm used to implement user interface components. It consists of two major parts. The first part is an underlying model object which interact directly with the data structure which in this case is the XML-file defining each module which makes up the visualization pipeline. This model object makes use of helper objects called "items" to define the model structure which is then made available to the view object which is the second part of the paradigm and the part of the widget the user interacts with. By using this programming technique, the data structures holding the information can be presented in an organized and intuitive way without having to create custom input boxes and display labels for altering the information. It takes some effort to create a good model for the view to use, but it limits the amount of code to create which in turn makes the system easier to maintain and update. The view used in the settings widget is a tree view which presents the attributes of a module, along with the property and connection nodes as tree nodes in the first column which are expandable and their different value items show up in the second column where the user can select and alter them. When a value is change the view passes the new value to the underlying model which keeps track of the value and what item it belongs to. The model is responsible for writing the new value to the data structure and perform the changes necessary to update the visualization pipeline which produces the new scene in the main view. In addition to the tree view displaying module

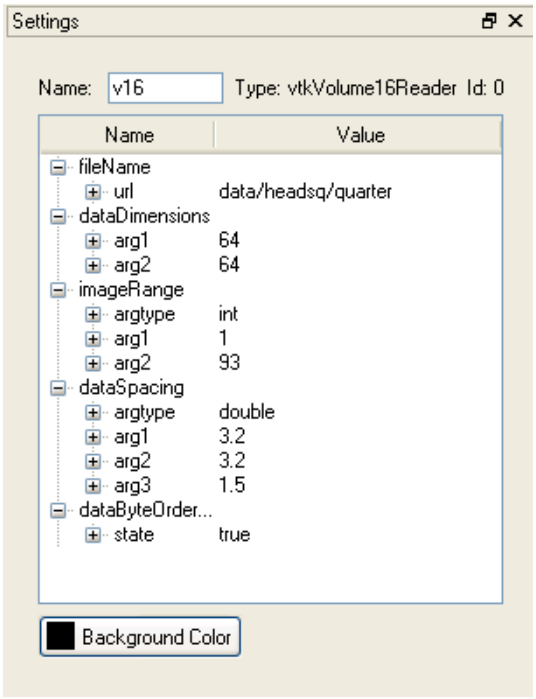


Figure 5.11: The Settings view showing the expanded view of a module's attributes.

information a button for setting the scene's background color has been added to the widget. In order to better represent a certain module property the view can be extended to include custom widgets inside the tree view in order to have a more intuitive and easier to use interface for the user not familiar with the VTK system. An example of such a scenario is to represent a color property as a button inside the tree view that activates a color picking widget instead of manually entering the RGB-components of the color. Another scenario closely related to volume visualization is the definition of a color and opacity transfer function for use in direct volume rendering. It is much more intuitive to define such a function using interactive widgets that display the colors and function points rather than entering the function arguments manually.

5.5.7 Help Browser Widget

The help browser is an HTML browser for displaying help documents inside the application. The help system is made using HTML and consists of a main page shown in figure 5.12 and hyper linked documents that define a help scenario. In addition to the help system, the widget displays information about every VTK module supported by the system along the generated system documentation for the implementation of the design discussed in this

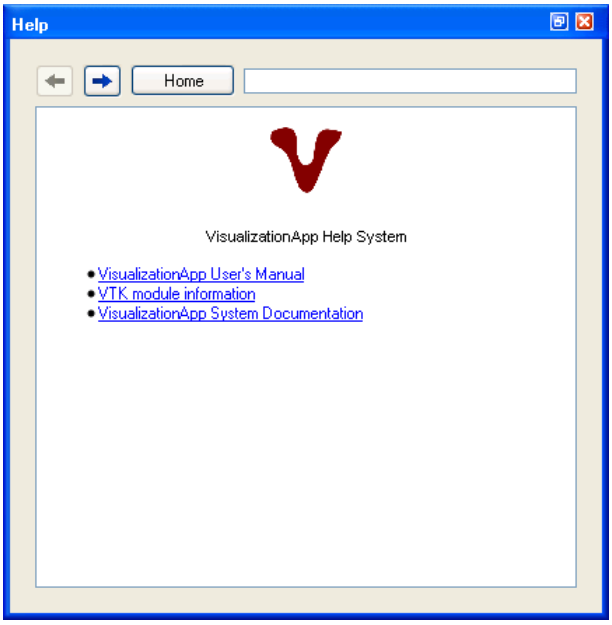


Figure 5.12: Help system menu.

report. The VTK information is retrieved from VTK's web page's doxyGen-generated class documentation pages and saved locally for use in the help browser. The doxyGen-generated class documentation for this project is also saved locally for use in the help browser. The help browser has no networking features, so it can only display files stored locally. All files displayed in the help browser are store inside the "doc" folder in the project folder. The help browser is fully implemented using the Qt Designer by adding the necessary components to the form used for the widget and laying them out using layout objects and component interactions defined by connecting signals and slots. This was done directly in the Qt Designer and demonstrates the capabilities of the Qt Designer. The designer project was compiled using the "uic" to generate C++ code which is included in the system project and included in the implementation along with the rest of the Qt Designer-created user interface features. Figure 5.13 shows the browser displaying the system's user manual. Figure 5.14 shows the VTK information overview while figure 5.15 shows the documentation page of a VTK module named vtkVolume16Reader. The system documentation for the implementation of the system design is shown in the help browser on figure 5.16.

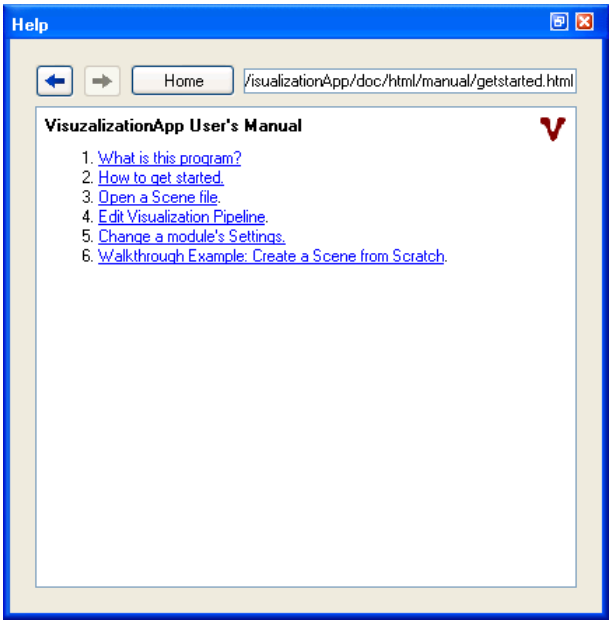


Figure 5.13: Help browser showing HTML User Manual.

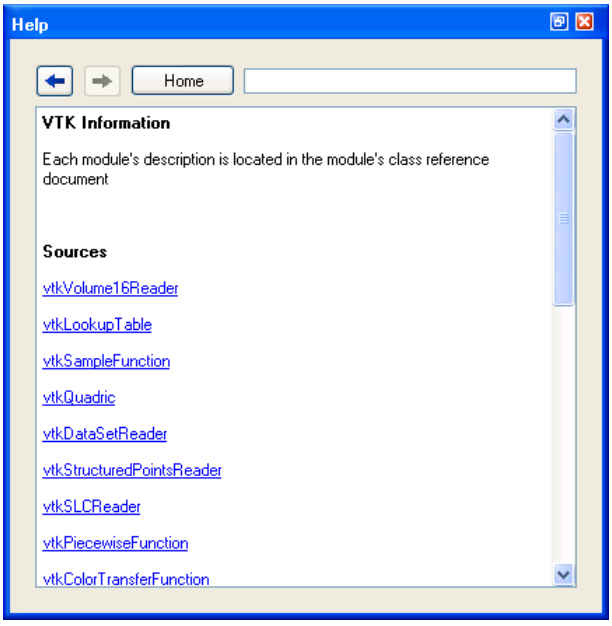


Figure 5.14: Help Browser showing VTK Modules supported by the system written in HTML.

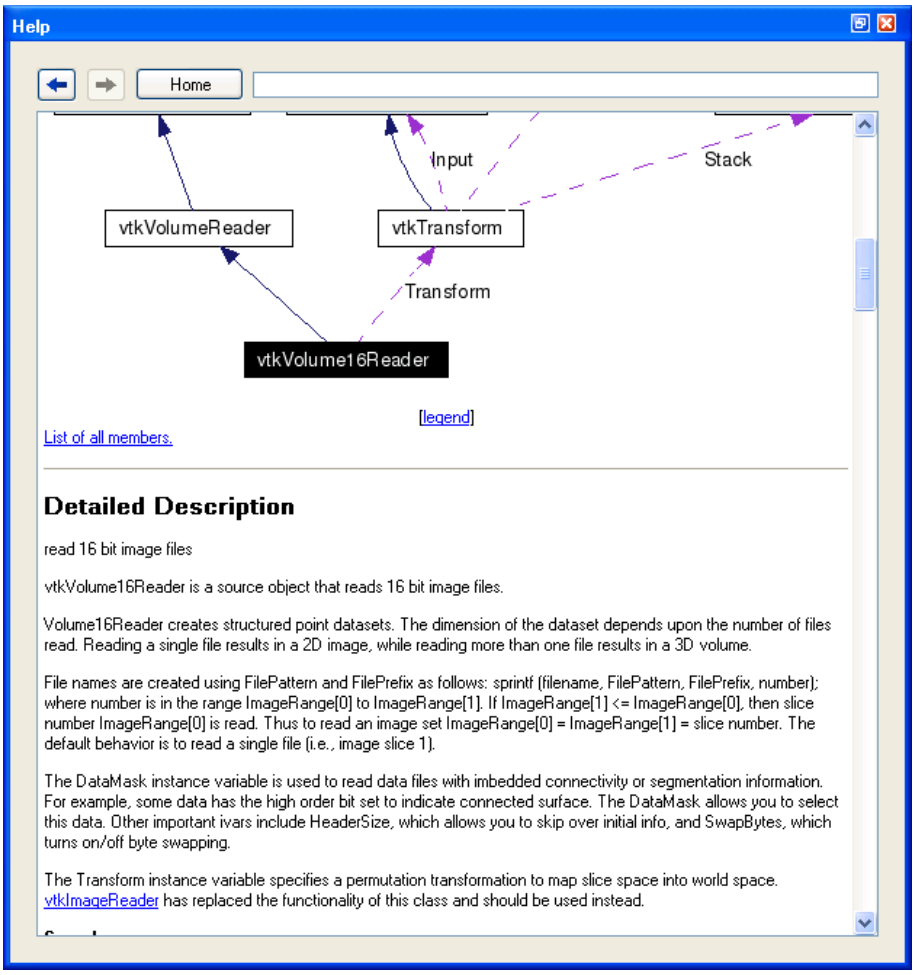


Figure 5.15: VTK's doxyGen HTML class documentation for vtkVolume16Reader shown in Help Browser.

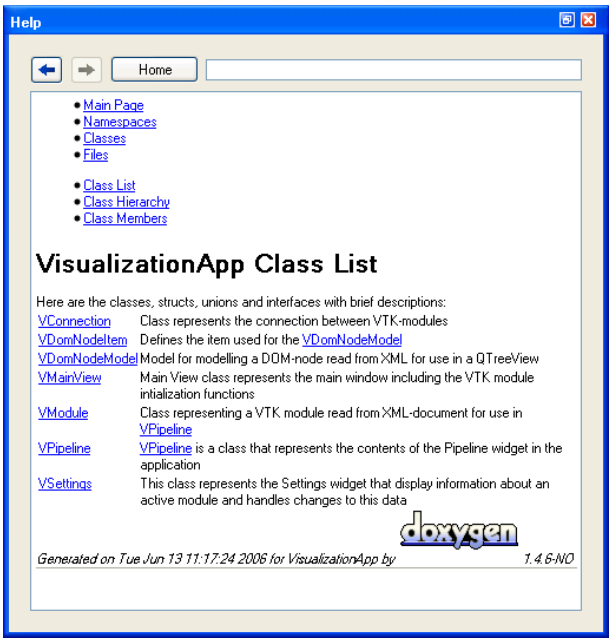


Figure 5.16: The generated HTML project system design class documentation shown in Help Browser.

5.6 System Class Design

The classes used in the implementation of the system design are named using the same prefix v- to clearly indicate that they are parts of the project and not to be confused with other libraries or implementations. The classes are declared in a header files and the function definitions are placed in cpp-files. The classes are organized in the classic C++ convention where the header and implementation files have the same name except for its file extension. The system is developed in Microsoft Visual Studio 8 (2005) downloaded from the Microsoft Academic Alliance website. For Qt development the educational license for Qt 4.1 and Qt Visual Studio Integration obtained by the project supervisor Ketil Bø was used. VTK 5.0 was downloaded from the VTK web page and configured using CMake 2.3 and built using Microsoft Visual Studio 8. The project is organized as a Visual Studio-solution project with the necessary Qt-features for generating C++ code from the Qt-specific class declarations in order to make use of Qt features such as signals and slots and actions and events.

A schematic overview of the different classes used in the system is shown in figure 5.17. The placement of the classes is relative to where the appear in the initial arrangement of the user interface.

The interaction between the different objects which define the widgets in the system is what defines the functionality of the application. In figure 5.18 a module is selected in the Pipeline widget which activates its property node in the tree view in the Settings widget. When an attribute's value is altered, the new XML element is parsed in the main view object which produces a new scene.

5.6.1 Main View Class With VTK-controls

The main view is implemented in the class VMainView which is a QMainWindow subclass which inherits all necessary Qt functionality. In addition, using multiple inheritance, it subclasses the class generated by the Qt Designer named Ui_MainWindow which defined the basic windows including the dock widgets used for the pipeline, settings widgets and help browser.

The constructor initialized the main window by instantiating the ui-features generated in Qt Designer and creates the main menu structure by creating actions that are added to the menu and connecting them to the slots to be activated when the menu item is selected.

The Pipeline and Settings widgets are instantiated and added to the different dockable widgets included in the Qt Designer generated code. The

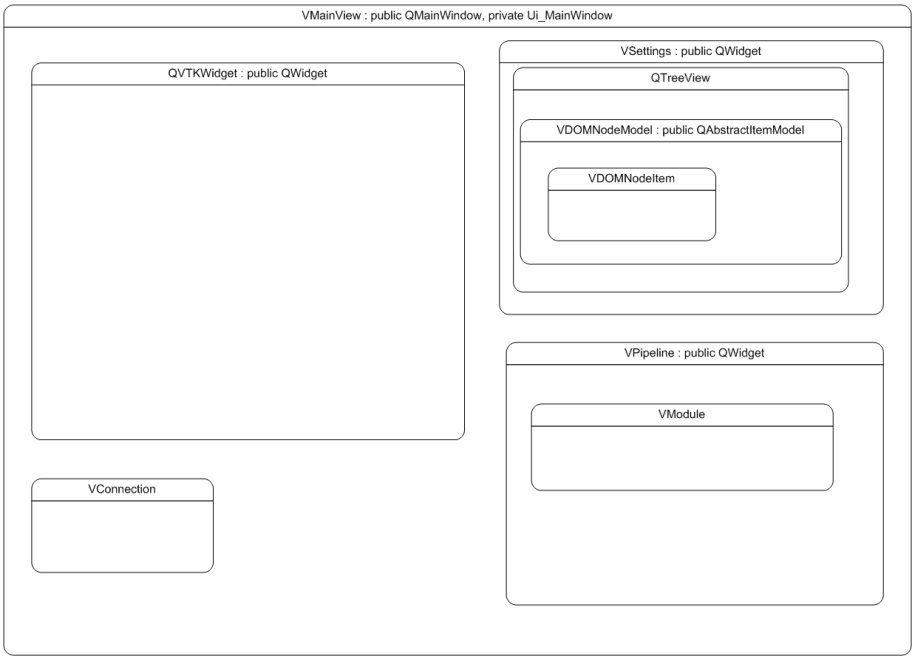


Figure 5.17: A schematic view of the classes used in the system. The placement of the classes is relative to where they initially appear int the user interface.

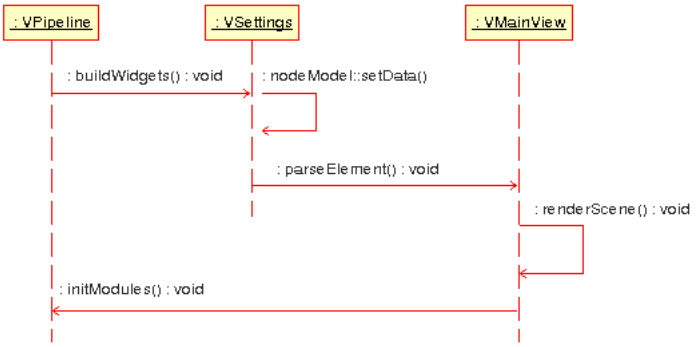


Figure 5.18: The sequence of selecting a module in the Pipeline widget, entering a new attribute value in the Settings widget and parsing the XML-node and rendering the new scene.

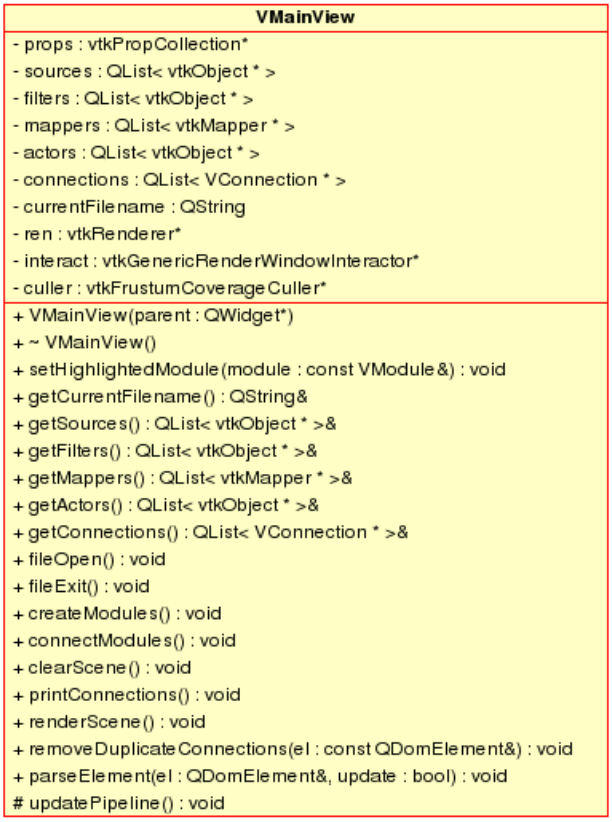


Figure 5.19: VMainView class diagram

dockable widgets are organized using layout objects that put the stuff in the right places.

The help browser's start page is directed to point at the start file at doc/html/help.html which is the start page for the help system.

The vtkRenderer object is instantiated and added to the vtkWidget used for screen output in the Qt application. A vtkPropCollection is instantiated for holding the VTK-actors that are to be rendered in the scene.

The class diagram for VMainView is shown in figure 5.19 and shows all member attributes and functions.

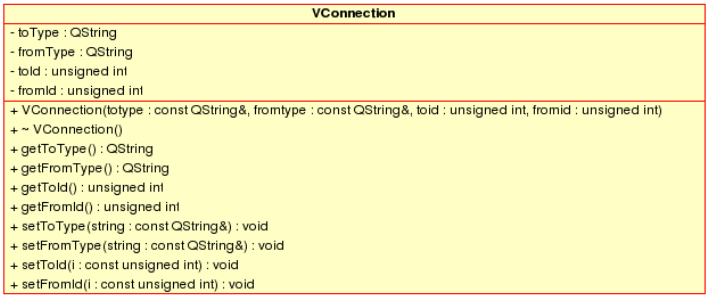


Figure 5.20: VConnection class diagram

5.6.2 Connecting Modules

A connection between two modules is implemented as a class called VConnection. It holds the type of the two modules being connected as strings along with their IDs stored as an integers to uniquely identify them in the pipeline. The connection object is created based in the connection-nodes in the modules stored in the XML-structure. They are instantiated in VMainView's function parseElement() and stored in the QList "connections" for later use in the function connectModules(). As a result the function parseElement() must be run first in order for the connections to first be stored in "connections". The class diagram for class VConnection is shown in figure 5.20.

5.6.3 VTK-Module Representation

To represent VTK modules in the pipeline network graph in the Pipeline widget a class for the representation of the modules as nodes in the graph is needed. The class is named VModule and the attributes for this class consists of information associated with a module which is available in the widget. The attributes for the VModule class are shown in figure 5.21 and in addition to information such as color and position in the widget the class also contains information about the module's type, name, ID, tool tip text and textual description. It also contains some boolean flag attributes for use in the widget's paint-function.

5.6.4 Pipeline Widget

The Pipeline widget's class is named VPipeline and inherits the QWidget class which provides the basic Qt widget functionality. The object is instantiated inside one of the main view's dockable widgets to make available the functionality for moving and resizing the widget. The Pipeline widget is a graphical interface to the visualization pipeline and represents the modules as node in the directed graph. The Pipeline graph contains four different types

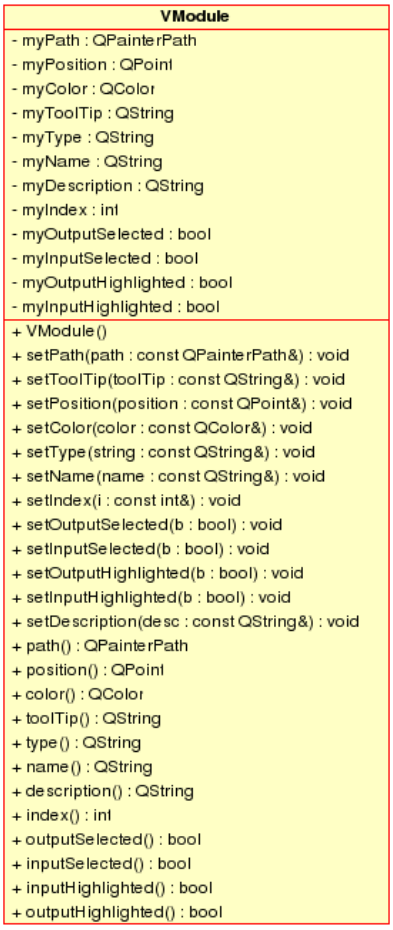


Figure 5.21: VModule class diagram

of nodes: sources, filters, mappers and actors. This corresponds to the VTK organization of objects that have different functions in the the visualization pipeline. The nodes are drawn as rectangular boxes filled with a given color according to its type. In addition, each module has both an output and input field marked with an arrow. This arrow indicates the direction of the data flow in the pipeline and with the a source module at the top of the graph, the modules should be placed below the so that the data flow is shown to have a direction pointing downwards. The nodes in the graph are objects of the VModule class and are drawn based on the attributes given to such an object. To clearly show that a source module can have no input, no input area is created for the source-nodes. Similarly actor modules have no output and actor-nodes are therefore not drawn with an output area in the graph. The drawing of the nodes and edges is performed in the paintEvent()-function of VPipeline and uses the VModule-objects as a base for drawing. It loops through a QList of VModule objects to paint each object in the graph. By checking the type of module in the loop the painting of the modules can be customized to fit its properties. The descriptive name of the module is drawn to the right of the module-node. To create the connections in the graph, the list of connections located in the VMainView object is used to check against each VModule-object's position and the edge between the nodes are drawn as simple black lines. The function initModules() parses the XML-document and creates the VModule objects that are used for drawing the graph. The class diagram for the VPipeline class is shown in figure 5.22. A much used function is the moduleAt()-function which finds the VModule object that is placed under the mouse cursor when a button is clicked. This is a vital function for manipulating the nodes, showing tool tips and in operations such as connecting modules and deleting modules.

5.6.5 Settings Widget Class

The Settings widget class inherits the QWidget functionality and the main functionality of the class is placed in the tree view which provides an interface to the XML-document loaded by implementing the Model/View programming paradigm. In this case the Model/View scheme involves a model for representing an XML DOM-node as the data to be presented in the view and handle user input from the view. The class that implements the model is VDomNodeModel and will be discussed below. The view used to present the data the model prepares to the user is a QTreeView with two columns and simple text-cells to provide interaction with the model items. To set the background color of the scene a button i displayed below the tree view. The button has an icon showing the current background color. This button activates a color picking dialog box when clicked and to change the background color, the user needs to pick a color from a standard color widget and confirm to set the new color.

Figure 5.22: VPipeline Class diagram

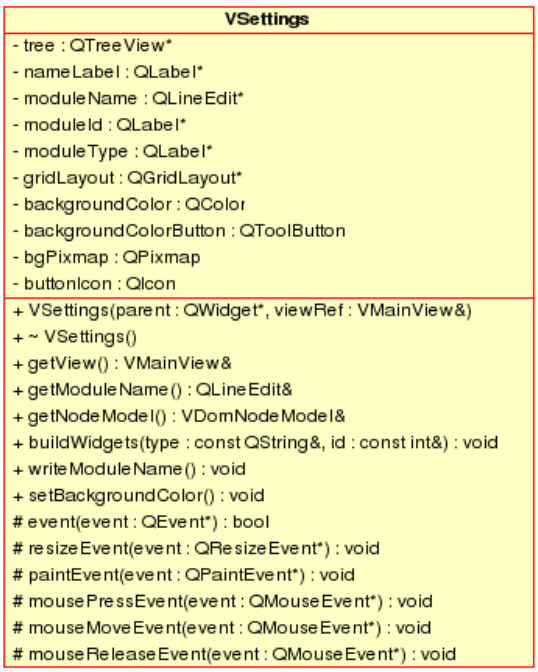


Figure 5.23: VSettings class diagram

5.6.6 DOM-node Model

The DOM-node model which is used to provide the tree's view and functionality inherits the QAbstractItemModel class which has functions that define how the XML DOM-node will be presented and changed according to the user input. In order to present the nodes as items in a tree view an item class is needed. In this case a class named VDomNodeItem is created to define an item for the model to use with the view. The model creates QModelIndex objects which are used by the view and holds all necessary information about a tree item in order to update the underlying data structure. The model contains functions for both presenting the data and setting new values when it received input from the tree view. The function responsible for arranging the data for the tree view is data() and the function that stores the new values entered in the tree view is setData(). VDomNodeModel's class diagram is shown in figure 5.24.

5.6.7 Item for use in DOM-Node Model

The item used in VDomNodeModel is named VDomNodeItem provides functionality related to the parent/child relationship in a hierarchical structure for use in the tree view in the Settings widget. It resolves the structure of

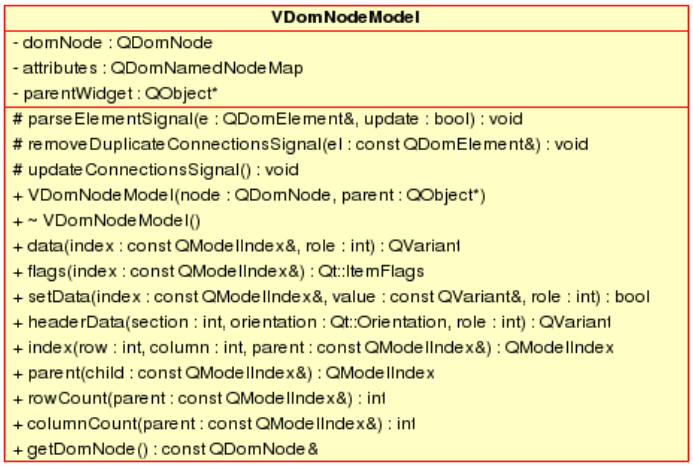


Figure 5.24: VDomNodeModel class diagram

the hierarchy based on a node's attributes and arranges them in way that is easier and more intuitive to the user. A DOM node can have multiple attributes in the main tag, so the VDomNodeItems are arranged to present an attribute as a single item in the tree. This structure is defined in the child()-function and makes the data easier to work with while preserving the compact format used for defining the nodes in the XML-scheme.

For detailed and complete class reference of the system design, see Appendix B.

5.7 Discussion

The most common use of the VTK library is to create custom scenes in code, be it Python, Java, Tcl or C++. This approach leaves no room to change the visualization pipeline during execution like the design suggested in this thesis. By constructing a layer for instantiating VTK classes and connecting the objects in run-time, the user suddenly has the choice of changing or adjusting an object's attributes interactively and the result is shown immediately on the screen. Because of the focus of the design involving volume rendering and the dual representation of opaque surfaces and direct volume rendering, such a design feature is very useful for creating and adjusting VTK modules in order to find the right values for iso-surfaces and in order to create useful color and opacity transfer functions. Such a transfer function must be individually adjusted for each data set and the user's chance of seeing the results of his adjustments interactively is crucial for efficient modeling of a certain part of the volume of interest.

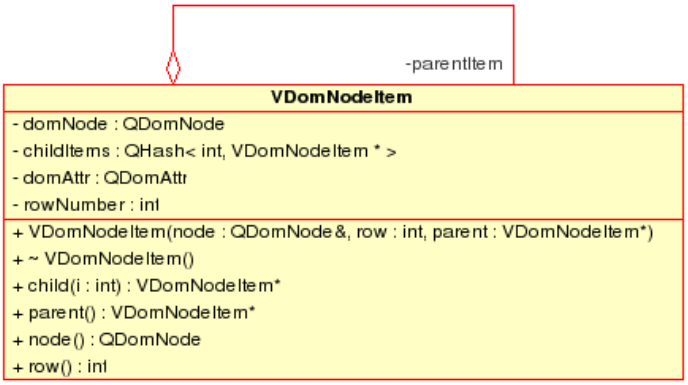


Figure 5.25: VDomNodeItem class diagram

Even though there is a certain overhead when it comes to code creation in order to support the VTK modules in the system, the intuitive user interface design will utilize the existing VTK subsystem in order to interactively create great visualizations. By having a sound structure to handle the components of the system it is easier to create a working scene from components already built into the system and easier to use and bring the capabilities of VTK out to a larger number of users. The way this system is designed makes it more generic and easier to add functionality and include support for other VTK classes. An alternative way of using VTK for such a system would be to build custom controls and widgets to control the properties of each module. This severely increases the programming overhead and makes maintenance and expanding the system harder than if it is organized in the generic way suggested in this design report.

Chapter 6

Results

6.1 User Interface Implementation

The user interface implemented proves to be a dynamic and flexible base for this system. The system design has led to the application presented in the following figures starting with figure 6.1 which shows the default layout of the user interface and a ray casted visualization of a human head. The user interface consists of the main windows along with the three dockable windows presented in the System Design chapter. The Settings widget is placed at the top of the window while the Pipeline widget is in middle with the help browser at the bottom. This is the default view, but the user interface is very flexible and the dockable windows may be moved around in the main window and docked as shown in figure 6.2 or left floating over the application as shown in figure 6.3. The dockable widgets may also be closed and reopened by toggling their visibility from the View menu as shown in figure 6.4. Trolltech Qt offers a great platform for creating applications and together with VTK the system works very well and should be a good platform for further development on the system to achieve even greater performance and error-proof design. Simplicity and minimalistic design has been an major goal when designing the user interface. The expectations and plans to make a simple and functional application has been achieved with this design. The choice of using Qt for creating the GUI and the capabilities of VTK makes this implementation a platform with great potential of being a successful application. However the fact that the user interface utilizes VTK specific components makes the system somewhat dependent of further knowledge about VTK and its architecture. By creating an integrated help system and by including VTK specific information in it, this issue has been taken into account. The design is not intended to be a system to be run by someone unfamiliar with volume visualization, but anyone who has some kind of previous knowledge about the techniques and concepts of volume visualization should find this system useful.

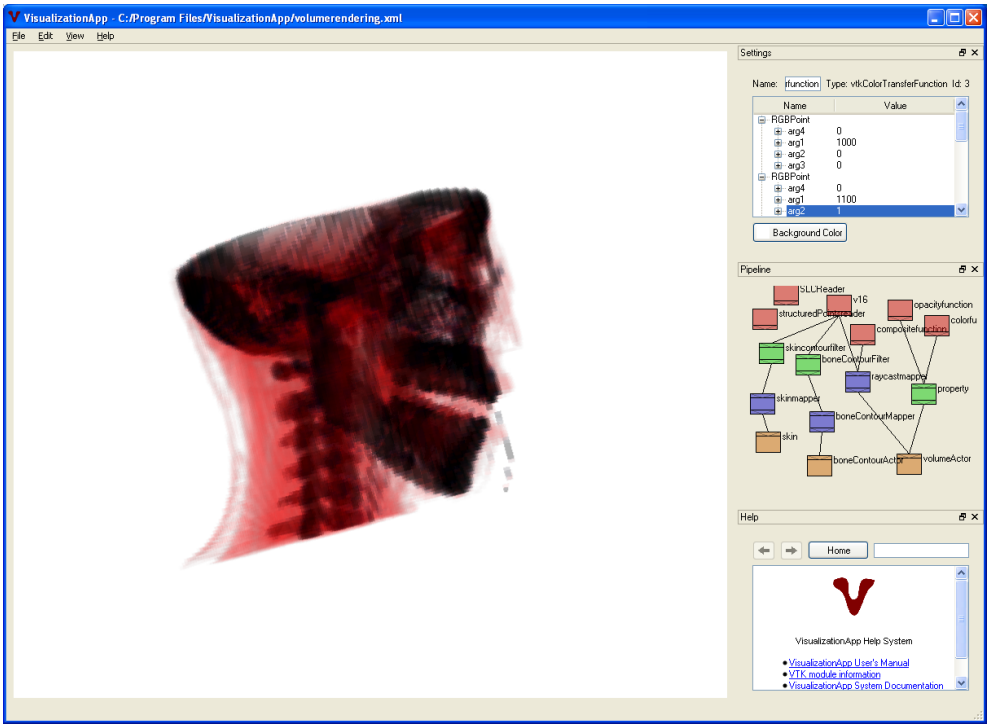


Figure 6.1: Resulting user interface with CT head data set rendered using Ray Casting.

6.2 Example Visualizations

Using the module vtkDataSetReader, simple polygonal files can be read and displayed in the scene using a simple network as shown in figure 6.4.

6.3 Volume Visualizations

Volumes may be rendered using either contours as iso-surfaces as in figure 6.5, 6.6, 6.7 and 6.8 or direct volume rendering using ray casting with opacity and color transfer functions as shown in figure 6.9 and 6.11.

The two approaches can also be used together to obtain the dual representation of voxel visualization and surface graphics aimed for in this project. Figure 6.10 shows a head data set from 93 images with dimension 64 x 64 pixels stacked to form a volume. The skin contour is partly transparent and also shows the internal cavities of the head that have the same density as the outer skin after extracting the iso-surface from the volume. The volume is also rendered using ray casting with opacity and color functions to emphasize the head volume's skull structure in white color. The transfer functions are

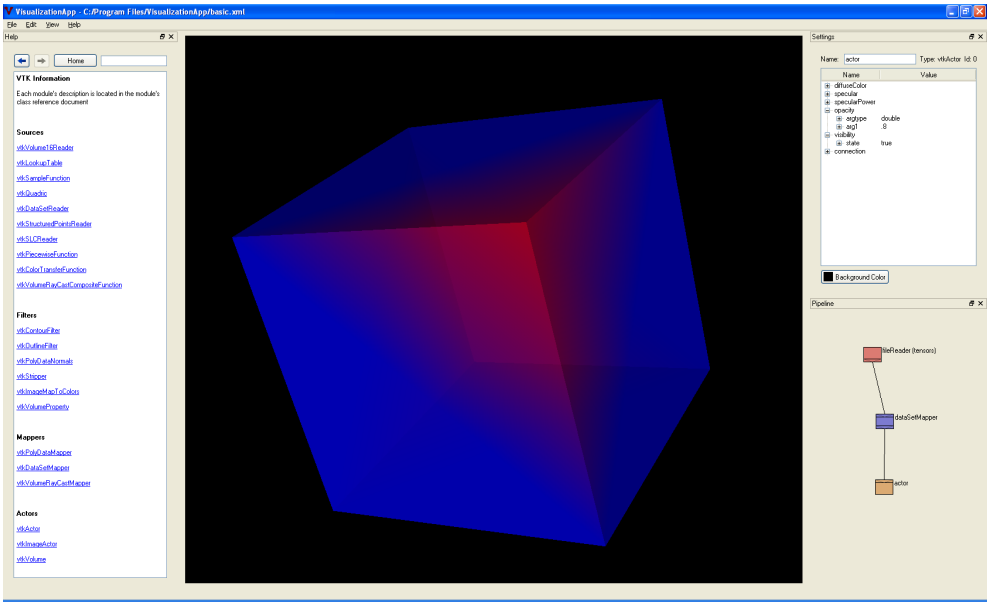


Figure 6.2: User interface with help browser moved to left docking area to better display more contents in the HTML documents. Similarly the other two dockable windows may be relocated to either the left, top or bottom of the main window.

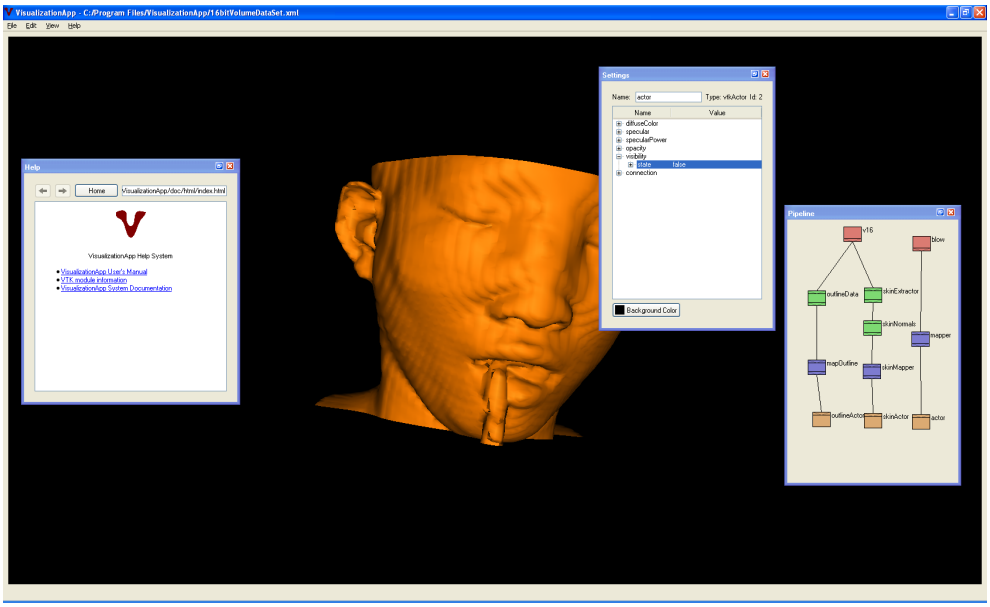


Figure 6.3: The dockable windows are left floating over the main window. This demonstrates the flexibility of a user interface implemented with Qt.

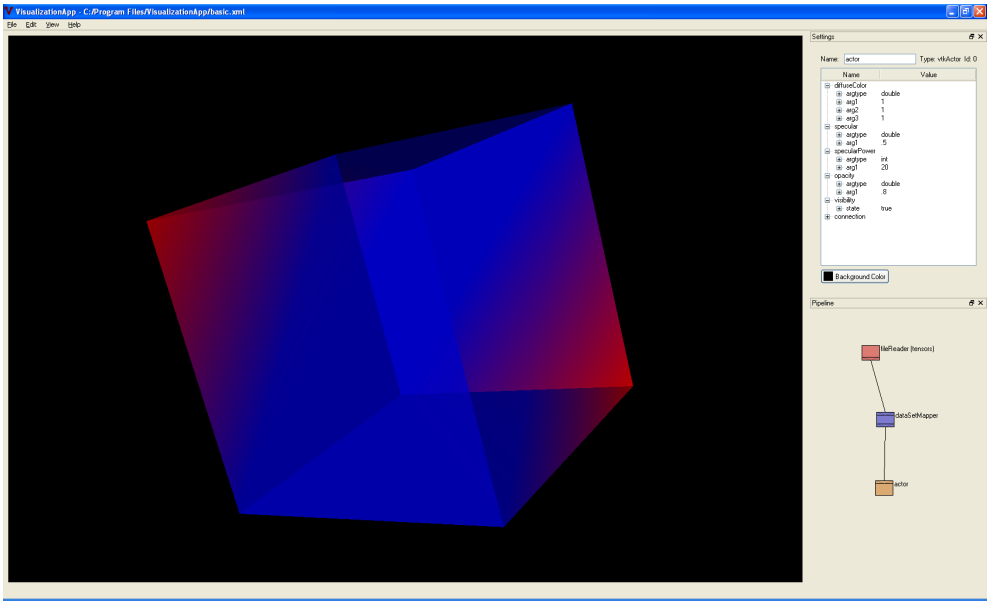


Figure 6.4: A basic polygonal model. Basic user interface configuration without help browser.

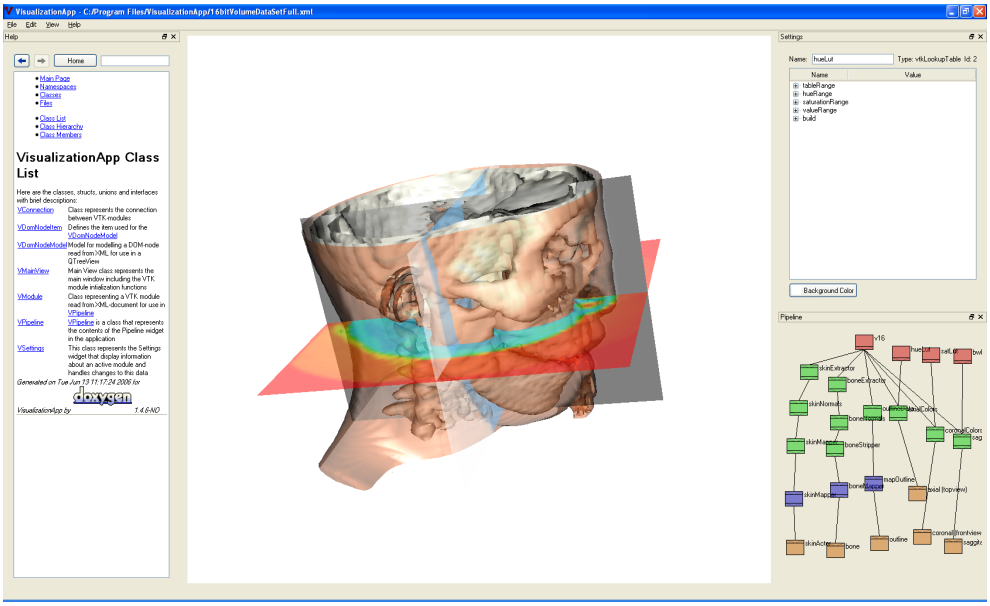


Figure 6.5: CT data set visualization using iso-surfaces to show bone and skin and cross slices showing saggital, coronal and axial views of the volume. The applications's help browser is shown at the left of the screen.

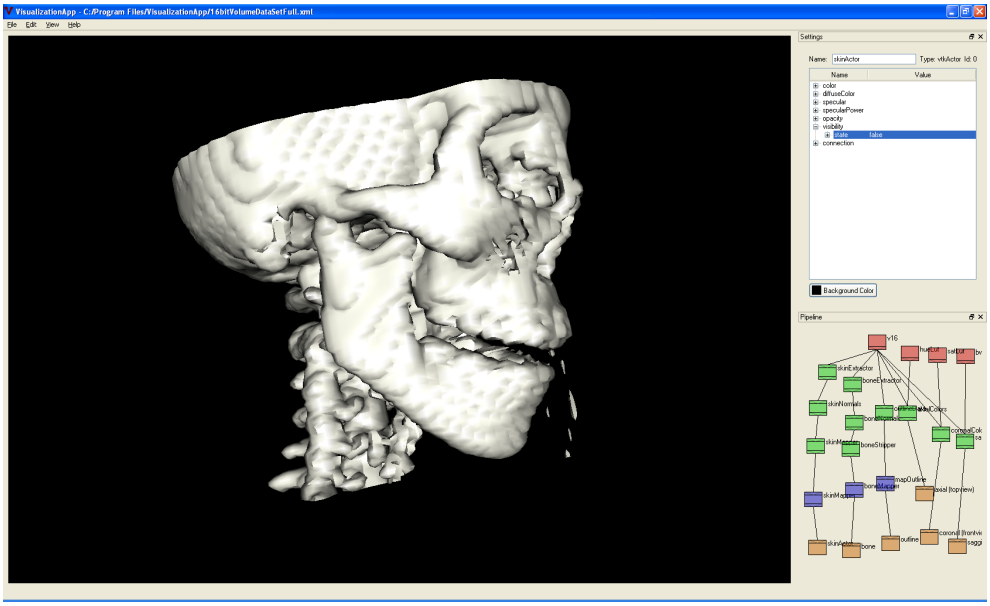


Figure 6.6: Skull contour extracted from volume using iso-surfaces.

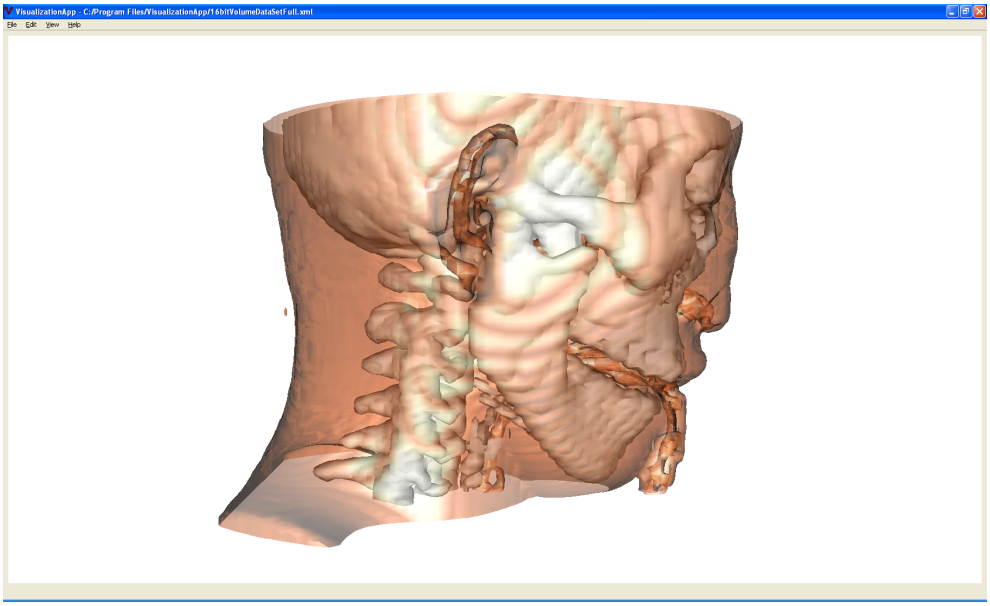


Figure 6.7: Both skin and bones are rendered using iso-surface representation. All tool windows are closed for main view to occupy the entire screen.

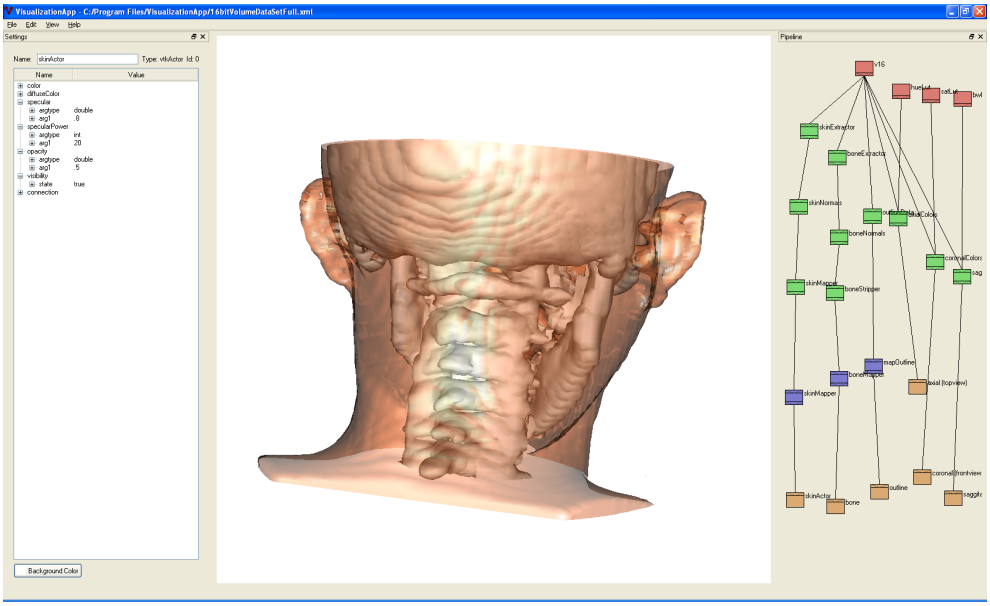


Figure 6.8: Skull with skin and bone rendered using iso-surfaces contours.

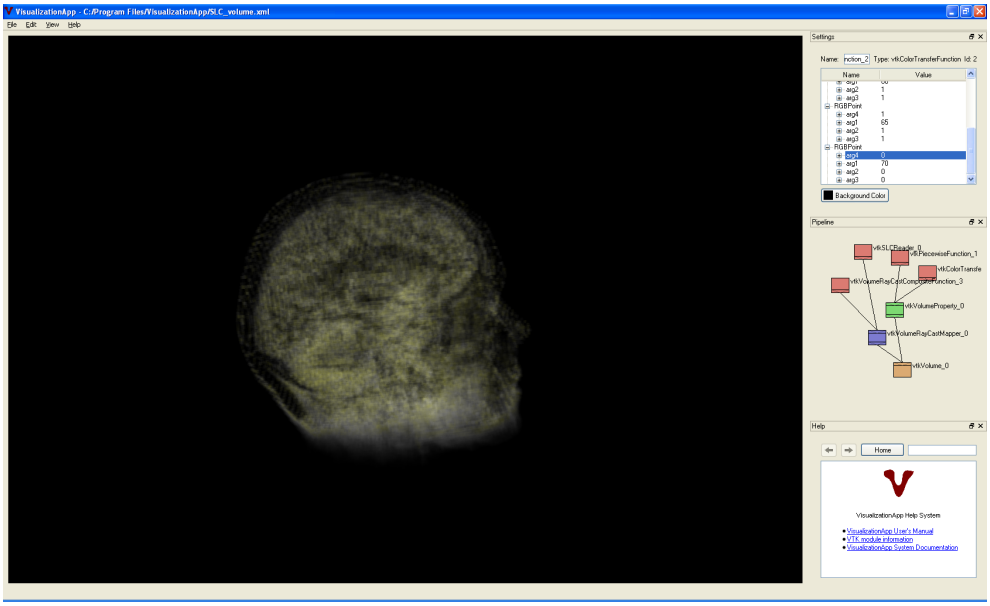


Figure 6.9: Head volume rendered with ray casting showing brain.

Figure 6.10: The skull is rendered using ray casting while internal cavities and skin are emphasized with an semi-transparent iso-surface.

manually defined in the pipeline and settings widgets and are implemented as vtkPiecewiseFunction for the opacity function and vtkColorTransferFunction. These modules should have a custom widget in the settings view to interactively define the functions to better fit the current volume in later versions of the system. A dual visualization of an iron protein is shown in figure 6.12 with both an iso-surface contour and direct volume rendering while figure 6.11 shows the same data set rendered using ray casting.

6.4 Project Details

The project is generically named "‘VisualizationApp" and is a Microsoft Visual Studio 2005 project and will not open in earlier versions of Visual Studio. In order to build it, VTK libraries need to be installed locally in C:/VTK and a commercial installation of Qt 4.1 must be present. The application will work on any computer when installed using the installer described below.

6.5 Deployment Project

The system installer project is built and made up from the system executable along with the necessary resource files. The installer program is a Visual

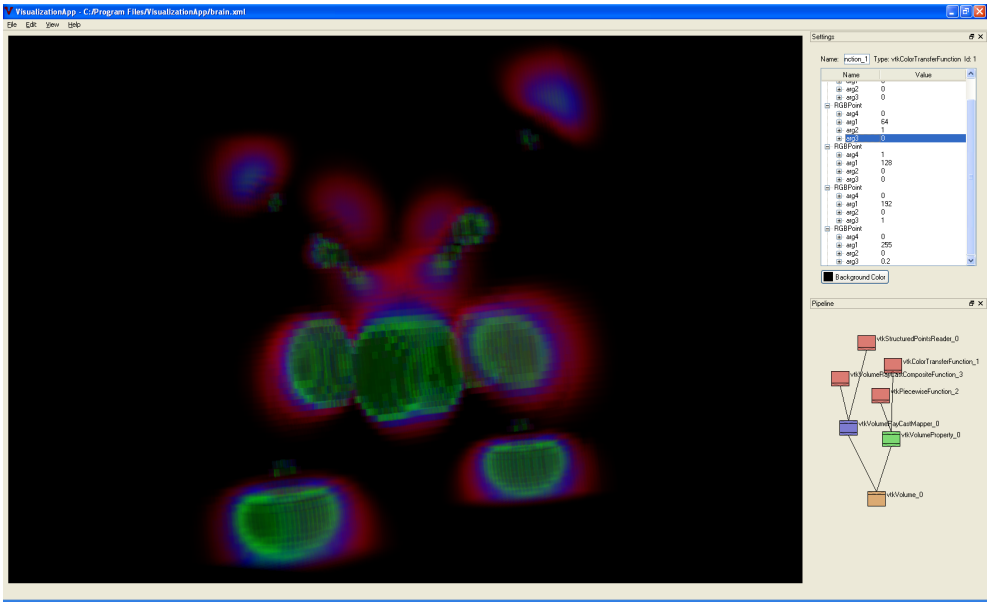


Figure 6.11: Iron protein rendered with ray casting using custom opacity and color transfer functions.

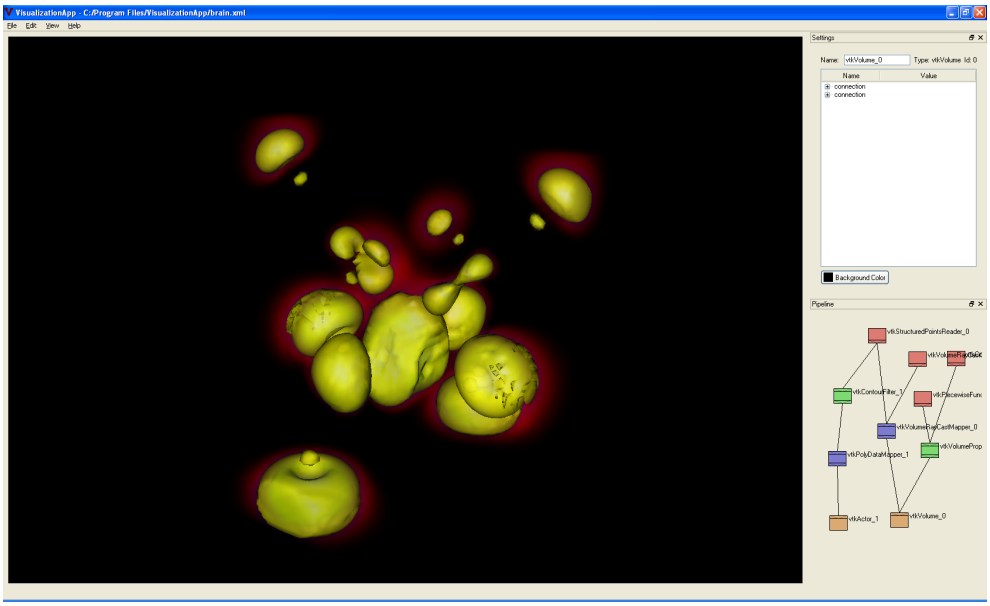


Figure 6.12: Volume visualization of iron protein using ray casting along with a contour generated by iso-surface extraction.

Studio 2005 deployment project which copies the files to the right folder on a target computer so that the visualization application will work on any PC running under Windows XP with an OpenGL enabled graphics card. The project is named "VisualizationAppInstaller" and creates the executable Microsoft Installer file names "VisualizationAppInstaller.msi". The package includes the VTK DLL-files necessary for executing the system. The DLLs are places in the application root folder. System documentation is located in the "doc" - folder along with the HTML user manual and HTML VTKInfo-files. The package also includes example scene files and example data sets.

Chapter 7

Further Work

If further work is to be done on this project the following topics are suggested

- The system could be extended to support a range of new VTK modules in order to create even more advanced visualizations. One way of creating this support could be to create a code generator that scan the VTK source code and generate C++ parsing code and the necessary XML-documents.
- Functionality for cutting a volume should be added by creating support for modules in VTK in order to display internal structures of a volume. Cutting geometry is very useful for inspecting a region of the volume by removing all obscuring information.
- Customized widgets could be added to give the user more intuitive control over the attributes in a given VTK module. An example of such an attribute is a color or opacity transfer function which could be associated with a graph widget that defined the transfer function. To add such support, the system will need new code to define the widget along with a more elaborate XML-scheme to define what actions are to be taken when a certain attribute is accessed. A custom implementation of the QTreeView used for this project using custom widgets activated from the tree cells is a possible solution.
- In order to improve the efficiency of the rendering of a scene when a module is added, deleted or has had an attribute updated, the system functionality can be further developed to work on the objects stored in the system and not delete all objects and rebuild the scene as it is done at present.
- Measures could also be taken to improve the performance of the VTK system when it comes to handling larger data sets. There is a description on the VTK web page how to improve its performance in order

to better handle larger data sets. This is especially relevant when it comes to volume visualization and the ability to use the system to inspect high resolution data sets.

- More advanced functionality when it comes to loading and saving files should be considered. There should be a backup version of the XML-file loaded in case a run-time error occurs when executing the pipeline. If there is no backup file, the user will not be able to load the file which has an erroneous pipeline configuration. A way of implementing this feature, could be to create a temporary version of the file which the system to create the pipeline. The original file, however, should not be written to unless the pipeline configuration is saved by choosing "Save File" from the main menu.

- An undo/redo system can be added in order to keep track of changes to the open document and undo and redo changes to the open XML-file.

Chapter 8

Conclusion

The system designed and implemented during the course of this project meets the expectations prior to taking on the task. The requirements specified have been met and the system has proved to be both flexible and efficient. By using VTK as visualization library, the desired volume visualization functionality is also provided. The fusion of VTK with Qt as GUI toolkit also proves to be a good match and the two prove to be good alternatives for software based on an open source architecture. The system design presented is a relatively novel design and uses modern approaches to solving the task using XML and the latest versions of two cutting edge libraries, VTK and Qt. These two libraries are the best cross platform open source libraries on their own areas and the combination of the two is a powerful one. The signals and slots paradigm found in Qt makes creating advanced applications easy and the fact that they can be compiled on Linux, Mac and Windows platforms makes this a versatile system which could be further improved and developed. The system created with the pipeline network graph and XML structure edited in a settings widget is a simple and intuitive solution once the user understands the structure behind the visualization system. It is not expected that any untrained user can grasp the concept of a visualization pipeline and the functions defined by the VTK modules. Therefore an extensive help system is created to cater for new users getting to know the system. Through exploring example scene files distributed with the system and support through the help system, the visualization application is an intuitive, flexible and well performing solution that could potentially be a successful application.

Bibliography

[1] B. Lorensen, K. Suiderveld, V. Simha, R. Wegenkittl, and M. Meissner. Volume rendering in medical applications: We've got pretty images, whats' left to do? 2002.

[2] Kevin Pulo. Direct volume rendering (dvr), 1999. `http://www.kev.pulo.com.au/sv3/sv3_1999_assignment1/node5.html`.

[3] Kevin Pulo. Isosurfaces, 1999. `http://www.kev.pulo.com.au/sv3/sv3_1999_assignment1/node4.html`.

[4] Vgl 3.2 website. `http://www.volumegraphics.com/products/vgl/index.html`.

[5] Vgl 3.2 product flyer. `http://www.volumegraphics.com/products/vgl/vgl_32_flyer_nopr.pdf`.

[6] Systems in motion home page. `http://www.sim.no`.

[7] Wikipedia: Retained mode. `http://en.wikipedia.org/wiki/Retained_mode`.

[8] Wikipedia: Scene graph. `http://en.wikipedia.org/wiki/Scene_graph`.

[9] *SIMVoleon Documentation.* `http://doc.coin3d.org/SIMVoleon/`.

[10] *OpenGL Volumizer 2.9 Release Notes.* `http://www.sgi.com/products/software/volumizer/relnotes_2.9.pdf`.

[11] *OpenGL Volumizer Tech Summary.* `http://www.sgi.com/products/software/volumizer/techsum.html`.

[12] Lisa S. Avila, Sebastien Barre, Berk Geveci, Amy Henderson, William A. Hoffman, Brad King, C. Charles Law, Kenneth M. Martin, and William J. Schroeder. *The VTK User's Guider VTK 4.2.* Kitware Inc., 2003.

[13] Matthias Kalle Dalheimer. *Programming with Qt, 2nd Edition.* O'Reilly Verlag GmbH & Co, 2002.

Appendix A

User Manual

A.1 What is this Program?

This program provides a way of utilizing the Visualization Toolkit (VTK) in order to dynamically create visualizations of a given data set. The way the user creates the visualization is by manually editing the VTK visualization pipeline. The user interface is shown in figure A.1

The VTK visualization pipeline is the system's execution mechanism and is the way a visualization is produced in VTK. The pipeline consists of plugin-modules that each has a distinct function in the system. These plug-ins are arranged into sources, filters, mappers and actors to form a pipeline that is defined by these three types of plugins.

VisualizationApp provides an intuitive way of connecting these plugins in a graphical pipeline widget, thereby altering the visualization pipeline in order to dynamically create the visualizations the user wishes to create.

Another function of the system is to access each plugin-object and changing the settings for each component of the system through a widget which is activated by selecting the desired plugin in the pipeline network widget.

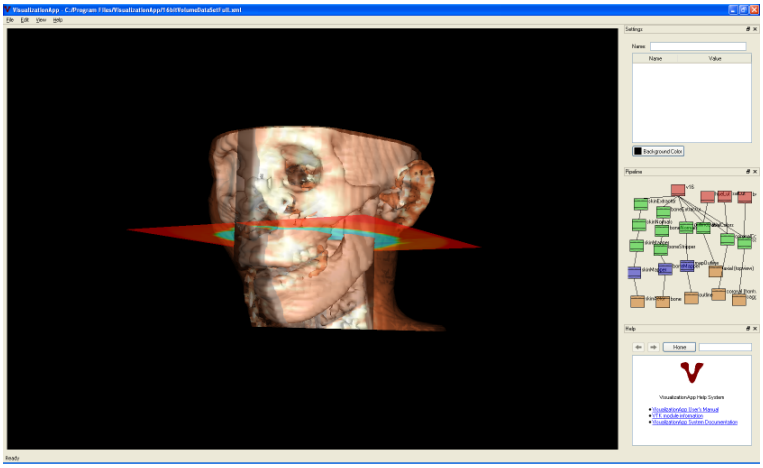


Figure A.1: Figure shows the user interface and it's default configuration.

A.2 How to Get Started

To make yourself familiar with the system, follow he next few steps to lean how the user interface works and how you can use it to create your own visualizations.

Study the *.xml files located in the application folder for example scenes and tips on how to create your own custom scenes.

A.3 Open a Scene File

The Visualization scene is stored in an XML-file located in the application's root folder. The file contains information about each plugin-module and defines the state of each scene.

To open a scene file, select from the main menu: "File" => "Open Scene File" or use hotkey "Ctrl + O". Select one of the XML-files located in the application's root folder in order to load a scene.

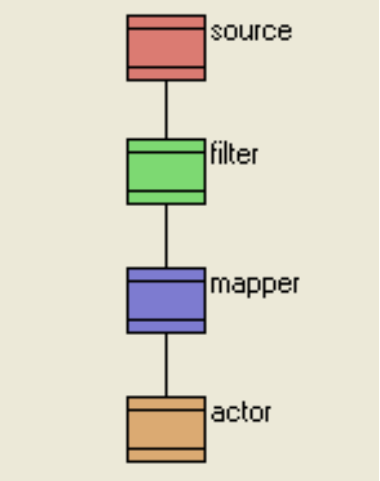


Figure A.2: Pipeline execution order.

A.4 Pipeline Widget

The pipeline network widget represents the visualization pipeline's execution. The chart should be read top-down with the sources placed at the top and the order order of execution read downwards from the top source.

In order to successfully create a visualization in the main view the pipeline need to contain at least one source, one mapper and one actor. In order to filter the data and create advanced visualizations a filter needs to be applied the the data before the mapper is executed. An example pipeline configuration is shown in figure A.2.

A.4.1 Connect Modules

To connect the modules, click the output field of the module you wish to connect from. A red line is drawn between the originating module and the mouse cursor as shown in figure A.3. Then select the input module by clicking its input field in order to connect the two modules.

A.4.2 Disconnect Modules

To disconnect two modules, click the input modules input field. The connection to the module which the connection originates from will now be deleted. Figure A.4 shows the network before the input area of the input modules is clicked. Figure A.5 shows the network after disconnecting.

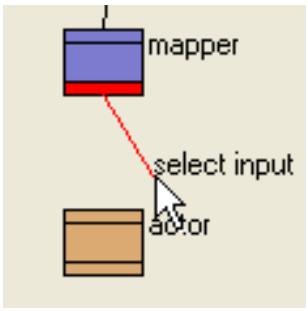


Figure A.3: Connecting modules.

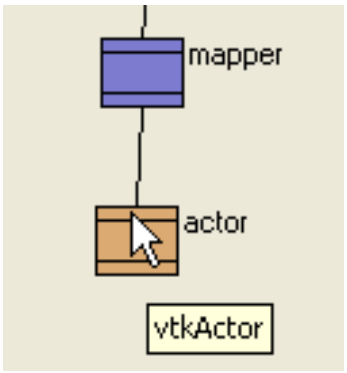


Figure A.4: Before disconnecting. Input area is clicked.

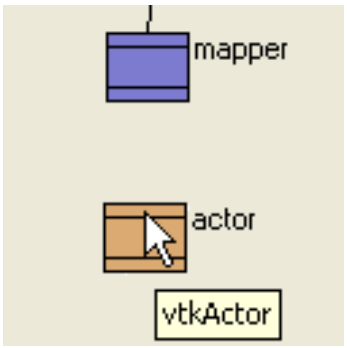


Figure A.5: After disconnecting modules.

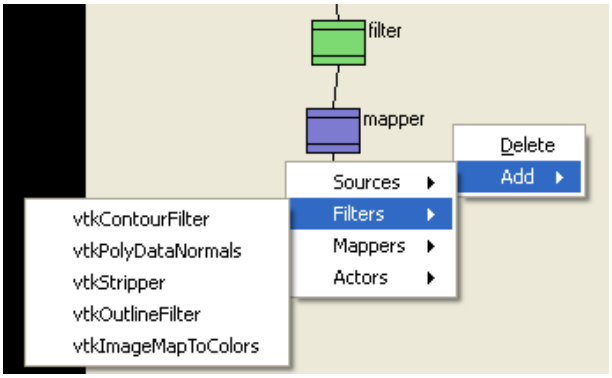


Figure A.6: Add a module using the context menu.

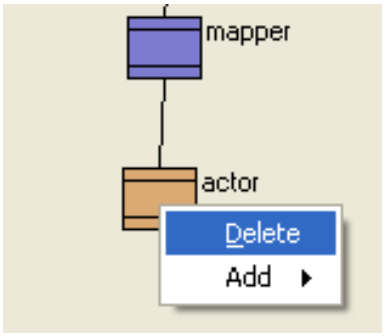


Figure A.7: Delete a module from the context menu activated by right clicking the module in Pipeline widget.

A.4.3 Add a New Module

To add a new module to the pipeline, activate the pipeline network widget's context menu by right clicking inside the widget or use the identical menu in the main menu bar. Select what type of module to add from the menu structure by clicking the menu item. The corresponding module will now be added to the pipeline. Remember to give the module a descriptive name in the Settings widget in order to identify the module. Figure A.6 shows the adding of a new module.

A.4.4 Delete a Module

If you want to remove one if the modules in the pipeline network, right click the module in the pipeline network widget and select "Delete" from the context menu. The module will now be permanently deleted from the pipeline. The context menu is shown in figure A.7. Click "Delete" to remove the current module.

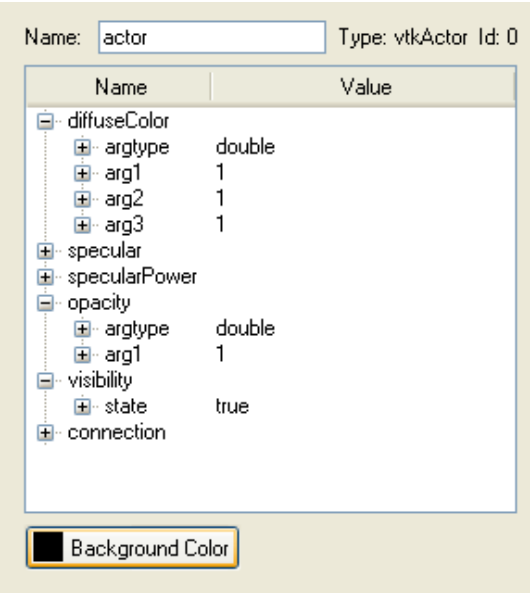


Figure A.8: The Settings widget where a module's attributes can be edited.

A.5 Change a Module's Settings

A plugin-module's settings can be altered in the Settings widget. The tree is activated by clicking one of the modules in the pipeline network widget. The corresponding settings will the be shown in the widget's tree structure.

In order to change a settings for a module, double click an item in the tree which represents the module's state. Enter a new value for an active node in the tree and confirm by pressing enter. The setting will now be changed and the scene will change correspondingly. The Settings widget is shown in figure A.8.

A.6 Walkthrough examples: Create visualizations from scratch

The following three examples show how to easily create both simple and advanced visualizations.

A.6.1 Simple File Loading

1. Add a vtkDataSetReader by right-clicking the Pipeline widget and choosing Add->Sources->vtkDataSetReader. Specify the file to read in the Settings widget by choosing fileName from the tree after activating the module in the Pipeline widget.

2. Add a vtkDataSetMapper from the "Mappers" sub-menu as above.

3. Add a vtkActor from the "Actors" sub-menu

4. Create the connections in the Pipeline widget by dragging a link from the output-area to the input-area of the receiving object:

   vtkDataSetReader to vtkDataSetMapper

   vtkDataSetMapper to vtkActor

5. The visualization should now appear in the main view.

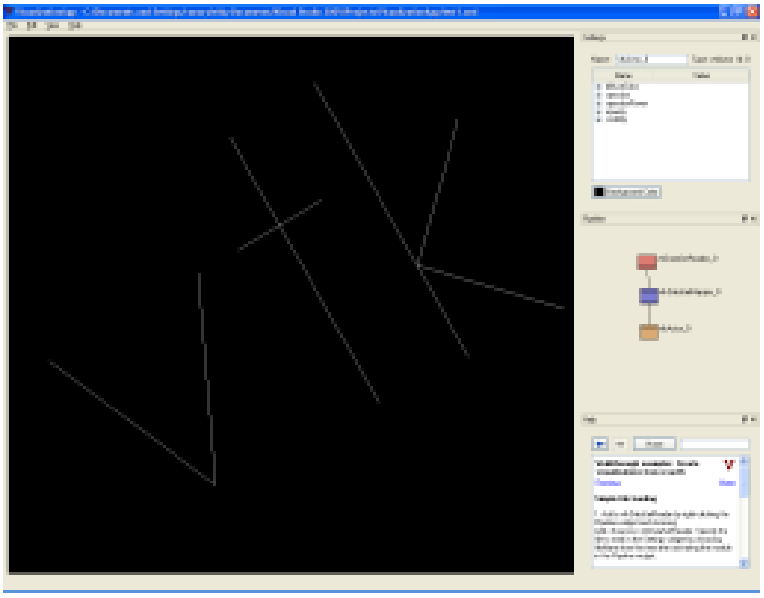


Figure A.9: Figure shows example configuration.

A.6.2 Volume Visualization Using Iso-surfaces

1. Add a vtkVolume16Reader with the default file and attributes from "Sources" menu.
2. Add a vtkContourFilter from "Filters" below the previous module.
3. Add a vtkPolyDataNormals from "Filters".
4. Add a vtkPolyDataMapper from "Mappers".
5. Add a vtkActor module from "Actors".
6. Connect the modules in the order added to the pipeline. When the actor is connected, a scene will be rendered showing the skin of a head from images acquired by a CT scan as shown in figure A.10.
7. To correctly scale the model, select the reader module to activate its settings. Expand the dataSpacing-item. Change the value of arg3 from 1 to .5.
8. Give the skin a more life-like color by selecting the actor module, then change the values of diffuseColor to arg1=1, arg2=0.49, arg3=0.25. Select the opacity-node and set its value to 0.5 to make the skin partly transparent as shown in figure A.11.

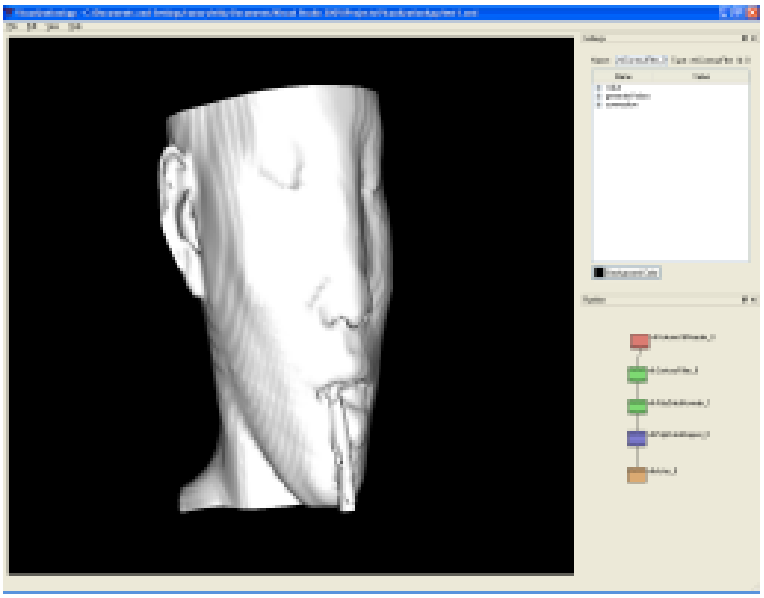


Figure A.10: Figure shows result after connecting the modules.

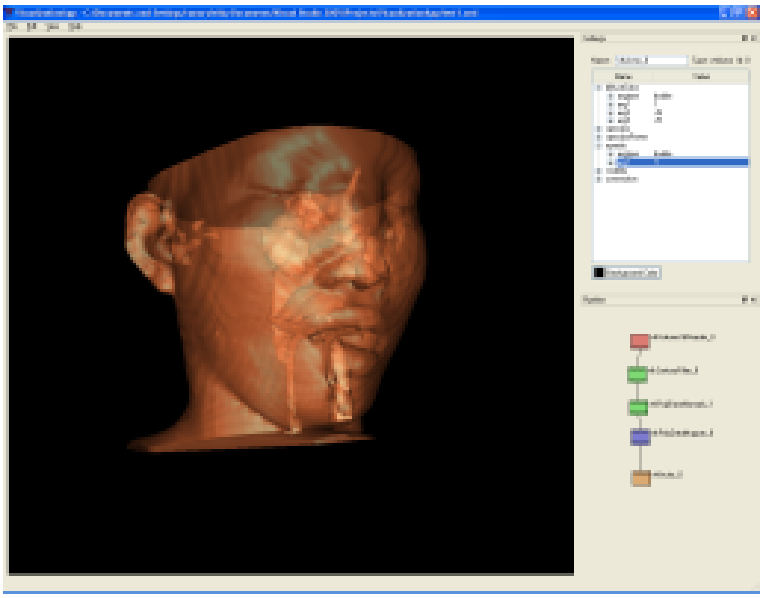


Figure A.11: Figure shows result after assigning color to the skin contour and settings opacity value to 0.5.

A.6.3 Volume Visualization using Ray Casting

1. Add a source vtkSLCReader. Change its attribute fileName from "data/head.slc" to "data/lobster11.slc".

2. Add a source vtkPiecewiseFunction and place it in the network.

3. Add a source vtkColorTransferFunction and place it in the network.

4. Add a filter vtkVolumeProperty and connect the vtkPiecewiseFunction and vtkColorTransferFunction to this module.

5. Add a vtkVolumeRayCastCompositeFunction.

6. Add a vtkVolumeRayCastMapper and connect the vtkSLCReader and vtkVolumeRayCastCompositeFunction to this.

7. Add a vtkVolume actor.

8. Connect the vtkVolumeRayCastMapper and vtkVolumeProperty to the vtkVolume module. The main window will show the resulting visualization as in figure A.12.

9. Click the button "Background Color" in the Settings widget and select a background color that better shows the object visualized as shown in figure A.13.

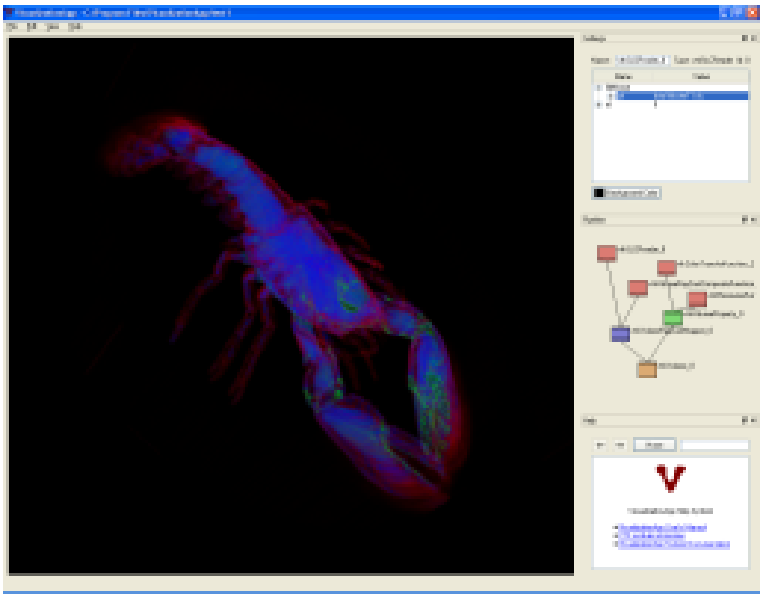


Figure A.12: Figure shows the data set rendered after connecting the modules.

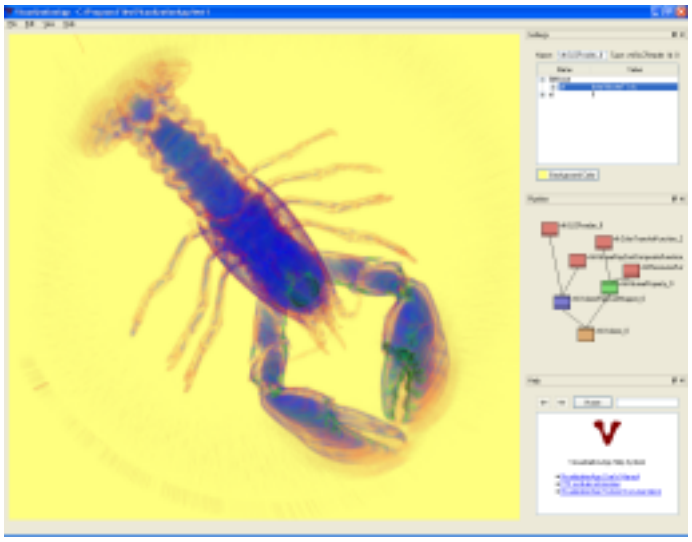


Figure A.13: Same scene after changing background color.

Appendix B

System Documentation

VisualizationApp Reference Manual

v1.0

Generated by Doxygen 1.4.6-NO

Tue Jun 13 11:17:12 2006

Contents

Chapter 1

VisualizationApp Hierarchical Index

1.1 VisualizationApp Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

VisualizationApp Class Index

2.1 VisualizationApp Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

VisualizationApp File Index

3.1 VisualizationApp File List

Here is a list of all files with brief descriptions:

Chapter 4

VisualizationApp Class Documentation

4.1 VConnection Class Reference

Class represents the connection between VTK-modules.

`#include <vconnection.h>`

Public Member Functions

- **VConnection** (const QString &totype, const QString &fromtype, unsigned int toid, unsigned int fromid, QString &**function**)

  A constructor.

- ~**VConnection** ()
- QString **getToType** ()

  toType access function.

- QString **getFromType** ()

  fromType access function.

- unsigned int **getToId** ()

  toId access function

- unsigned int **getFromId** ()

  fromId access function.

- QString **getFunction** ()

  function access function.

- void **setToType** (const QString &string)

  toType assigment function.

- void **setFromType** (const QString &string)

fromType assigment function.

- void **setToId** (const unsigned int i)
  toId assigment function.
- void **setFromId** (const unsigned int i)
  fromId assigment function.
- void **setFunction** (const QString &string)
  function assignment function

Private Attributes

- QString **toType**
  String holding the type of module to connect to.
- QString **fromType**
  String holding the type of module to connect from.
- unsigned int **toId**
  Integer holding the ID of the module to connect to.
- unsigned int **fromId**
  Integer holding the ID of the module to connect from .
- QString **function**
  String holding the function to activate when connecting module.

4.1.1 Detailed Description

Class represents the connection between VTK-modules.

VConnection(p. 7) is used in **VMainView**(p. 23) to hold the connection between VTK-modules

Definition at line 20 of file vconnection.h.

4.1.2 Constructor & Destructor Documentation

4.1.2.1 VConnection::VConnection (const QString & *totype*, const QString & *fromtype*, unsigned int *toid*, unsigned int *fromid*, QString & *function*)

A constructor.

Parameters:

totype Holds the type of module to connect to.

fromtype Holds the type of module to connect from.

toid Holds the ID of the module to connect to.

fromid Holds the ID of the module to connect from.

fromid Holds the name of the function to activate.

Definition at line 23 of file vconnection.cpp.

4.1.2.2 VConnection::~VConnection ()

4.1.3 Member Function Documentation

4.1.3.1 unsigned int VConnection::getFromId ()

fromId access function.

Returns:
The id of the module to connect from.

Definition at line 72 of file vconnection.cpp.

References fromId.

4.1.3.2 QString VConnection::getFromType ()

fromType access function.

Returns:
String holding the type of current to connect from.

Definition at line 48 of file vconnection.cpp.

References fromType.

4.1.3.3 QString VConnection::getFunction ()

function access function.

Returns:
The function to activate in the receiver module.

Definition at line 86 of file vconnection.cpp.

References function.

4.1.3.4 unsigned int VConnection::getToId ()

toId access function

Returns:
The id of the module to connect to.

Definition at line 60 of file vconnection.cpp.

References toId.

4.1.3.5 QString VConnection::getToType ()

toType access function.

Returns:
String holding the type of current to connect to.

Definition at line 36 of file vconnection.cpp.

References toType.

4.1.3.6 void VConnection::setFromId (const unsigned int *i*)

fromId assigment function.

Parameters:
i Input int.

Definition at line 135 of file vconnection.cpp.

References fromId.

4.1.3.7 void VConnection::setFromType (const QString & *string*)

fromType assigment function.

Parameters:
string Input string.

Definition at line 111 of file vconnection.cpp.

References fromType.

4.1.3.8 void VConnection::setFunction (const QString & *string*)

function assignment function

Definition at line 145 of file vconnection.cpp.

References function.

4.1.3.9 void VConnection::setToId (const unsigned int *i*)

toId assigment function.

Parameters:
i Input int.

Definition at line 123 of file vconnection.cpp.

References toId.

4.1.3.10 void VConnection::setToType (const QString & *string*)

toType assigment function.

Parameters:
string Input string.

Definition at line 99 of file vconnection.cpp.

References toType.

4.1.4 Member Data Documentation

4.1.4.1 unsigned int VConnection::fromId [private]

Integer holding the ID of the module to connect from .

Definition at line 59 of file vconnection.h.

Referenced by getFromId(), and setFromId().

4.1.4.2 QString VConnection::fromType [private]

String holding the type of module to connect from.

Definition at line 49 of file vconnection.h.

Referenced by getFromType(), and setFromType().

4.1.4.3 QString VConnection::function [private]

String holding the function to activate when connecting module.

Definition at line 64 of file vconnection.h.

Referenced by getFunction(), and setFunction().

4.1.4.4 unsigned int VConnection::toId [private]

Integer holding the ID of the module to connect to.

Definition at line 54 of file vconnection.h.

Referenced by getToId(), and setToId().

4.1.4.5 QString VConnection::toType [private]

String holding the type of module to connect to.

Definition at line 44 of file vconnection.h.

Referenced by getToType(), and setToType().

The documentation for this class was generated from the following files:

- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vconnection.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vconnection.cpp**

4.2 VDomNodeItem Class Reference

The **VDomNodeItem**(p. 12) class defines the item used for the **VDomNodeModel**(p. 16).

`#include <vdomnodeitem.h>`

Collaboration diagram for VDomNodeItem:

Public Member Functions

- **VDomNodeItem** (QDomNode &node, int row, **VDomNodeItem** *parent=0)
  A standard Constructor.
- ~**VDomNodeItem** ()
  Desctructor which cleans the childItems in hash table.
- **VDomNodeItem** * **child** (int i)
  Function defines the child Items of the Item based on the DOM-node child nodes and attributes.
- **VDomNodeItem** * **parent** ()
  parentItem access function.
- QDomNode **node** () const
  domNode access function.
- int **row** ()
  rowNumber access function.

Private Attributes

- QDomNode **domNode**
  The Node being modelled as an Item for the Model.
- QHash< int, **VDomNodeItem** * > **childItems**
  Hash table holding the children of the current Item.
- **VDomNodeItem** * **parentItem**
  The parent of the current Item.
- QDomAttr **domAttr**
  DOM-attribute for use in Model.
- int **rowNumber**
  Row Number assigned to Item for use in the Model.

4.2.1 Detailed Description

The **VDomNodeItem**(p. 12) class defines the item used for the **VDomNodeModel**(p. 16).

The **VDomNodeItem**(p. 12) class defines the item used for the **VDomNodeModel**(p. 16) used for the TreeView used in the settings-widget. It represents a QDomNode read from an XML-file.

Definition at line 20 of file vdomnodeitem.h.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 VDomNodeItem::VDomNodeItem (QDomNode & *node*, int *row*, VDomNodeItem * *parent* = 0)

A standard Constructor.

Parameters:

node The node which the Item is based on.

row The row of the Item in the Model structure.

parent The parent Item associated witht this Item.

Definition at line 23 of file vdomnodeitem.cpp.

References domNode, parent(), parentItem, and rowNumber.

Referenced by child().

Here is the call graph for this function:

4.2.2.2 VDomNodeItem::~VDomNodeItem ()

Desctructor which cleans the childItems in hash table.

Definition at line 36 of file vdomnodeitem.cpp.

References childItems.

4.2.3 Member Function Documentation

4.2.3.1 VDomNodeItem * VDomNodeItem::child (int *i*)

Function defines the child Items of the Item based on the DOM-node child nodes and attributes.

Parameters:

i Index value.

Returns:

Returns the Item with the childItems included

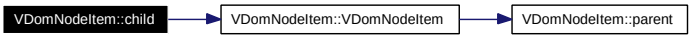
This function examines the DOM node i order to create child items. If the DOM node has child nodes they will be stored as children of this Items. If the node has attributes, theay will also be stores as children in order to structure the elements in the XML-document in the desired way.

Definition at line 97 of file vdomnodeitem.cpp.

References childItems, domNode, and VDomNodeItem().

Referenced by VDomNodeModel::index().

Here is the call graph for this function:



4.2.3.2 QDomNode VDomNodeItem::node () const

domNode access function.

Returns:
The DOM-node used for Item

Definition at line 77 of file vdomnodeitem.cpp.

References domNode.

Referenced by VDomNodeModel::data(), and VDomNodeModel::rowCount().

4.2.3.3 VDomNodeItem * VDomNodeItem::parent ()

parentItem access function.

Returns:
The parent Item associated with this Item.

Definition at line 51 of file vdomnodeitem.cpp.

References parentItem.

Referenced by VDomNodeModel::parent(), and VDomNodeItem().

4.2.3.4 int VDomNodeItem::row ()

rowNumber access function.

Returns:
Row number for Model structure

Definition at line 64 of file vdomnodeitem.cpp.

References rowNumber.

4.2.4 Member Data Documentation

4.2.4.1 QHash<int,VDomNodeItem*> VDomNodeItem::childItems [private]

Hash table holding the children of the current Item.

Definition at line 43 of file vdomnodeitem.h.

Referenced by child(), and ~VDomNodeItem().

4.2.4.2 QDomAttr VDomNodeItem::domAttr [private]

DOM-attribute for use in Model.

Definition at line 55 of file vdomnodeitem.h.

4.2.4.3 QDomNode VDomNodeItem::domNode [private]

The Node being modelled as an Item for the Model.

Definition at line 36 of file vdomnodeitem.h.

Referenced by child(), node(), and VDomNodeItem().

4.2.4.4 VDomNodeItem* VDomNodeItem::parentItem [private]

The parent of the current Item.

Definition at line 49 of file vdomnodeitem.h.

Referenced by parent(), and VDomNodeItem().

4.2.4.5 int VDomNodeItem::rowNumber [private]

Row Number assigned to Item for use in the Model.

Definition at line 61 of file vdomnodeitem.h.

Referenced by row(), and VDomNodeItem().

The documentation for this class was generated from the following files:

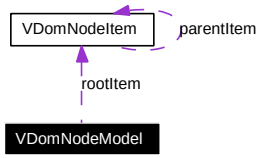
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vdomnodeitem.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vdomnodeitem.cpp**

4.3 VDomNodeModel Class Reference

Model for modelling a DOM-node read from XML for use in a QTreeView.

`#include <vdomnodemodel.h>`

Collaboration diagram for VDomNodeModel:



Signals

- void **parseElementSignal** (QDomElement &e, bool update)
- void **removeDuplicateConnectionsSignal** (const QDomElement &el)
- void **updateConnectionsSignal** ()

Public Member Functions

- **VDomNodeModel** (QDomNode node, QObject *parent=0)

  VDomNodeModel(p. 16) *Constructor. Initializes members and connects signals and slots.*

- **~VDomNodeModel** ()

  Write brief comment for ~VDomNodeModel here. Destructor.

- QVariant **data** (const QModelIndex &index, int role) const

  Returna the data from each Item contained in the QModelIndex object.

- Qt::ItemFlags **flags** (const QModelIndex &index) const

  Sets the flags associated with model.

- bool **setData** (const QModelIndex &index, const QVariant &value, int role=Qt::EditRole)

  Function for setting the values in the underlying data structure when editing the View.

- QVariant **headerData** (int section, Qt::Orientation orientation, int role=Qt::DisplayRole) const

  Sets the Model header data for use in the associated View.

- QModelIndex **index** (int row, int column, const QModelIndex &parent=QModelIndex()) const

  Returning the QModelIndex for a given Item after calling the child-functions of each Item.

- QModelIndex **parent** (const QModelIndex &child) const

  Returns the parent QModelIndex for a child QModelIndex.

- int **rowCount** (const QModelIndex &parent=QModelIndex()) const

  Returns the number of rows in the model.

- int **columnCount** (const QModelIndex &parent=QModelIndex()) const

  Returns the number of columns of the model.

- const QDomNode & **getDomNode** ()

  domNode access function.

Private Attributes

- QDomNode **domNode**

  The DOM node.

- QDomNamedNodeMap **attributes**

  The attributes arranged in a QDomNamedNodeMap.

- **VDomNodeItem** * **rootItem**

  The root item for the model.

- QObject * **parentWidget**

  The model's parent widget.

4.3.1 Detailed Description

Model for modelling a DOM-node read from XML for use in a QTreeView.

Represents a Model for use in a QTreeView based on VDomNodeItems. This Model is one part of the Qt Model/View programming paradigm. The other part used is this system is a QTreeView which works together with this Model in order to display and edit the data.

Definition at line 25 of file vdomnodemodel.h.

4.3.2 Constructor & Destructor Documentation

4.3.2.1 VDomNodeModel::VDomNodeModel (QDomNode *node*, QObject * *parent* = 0)

VDomNodeModel(p. 16) Constructor. Initializes members and connects signals and slots.

Parameters:

node The base node for the model.

parent The parent object.

Definition at line 23 of file vdomnodemodel.cpp.

References domNode, parentWidget, parseElementSignal(), rootItem, and updateConnectionsSignal().

4.3.2.2 VDomNodeModel::~VDomNodeModel ()

Write brief comment for ~VDomNodeModel here. Destructor.

Definition at line 42 of file vdomnodemodel.cpp.

References rootItem.

4.3.3 Member Function Documentation

4.3.3.1 int VDomNodeModel::columnCount (const QModelIndex & *parent* = QModelIndex()) const

Returns the number of columns of the model.

Parameters:
parent The parent QModelIndex.

Returns:
Returns the number of columns for the View to create.

Definition at line 58 of file vdomnodemodel.cpp.

4.3.3.2 QVariant VDomNodeModel::data (const QModelIndex & *index*, int *role*) const

Returna the data from each Item contained in the QModelIndex object.

Parameters:
index The QModelIndex object from which the DOM-node is accessed though.
role The role of the QModelIndex.

Returns:
The data returned as QVariant.

Definition at line 219 of file vdomnodemodel.cpp.

References attributes, and VDomNodeItem::node().

Here is the call graph for this function:

4.3.3.3 Qt::ItemFlags VDomNodeModel::flags (const QModelIndex & *index*) const

Sets the flags associated with model.

Parameters:
index QModelIndex holding the Item information.

Returns:
Qt::ItemFlags.

Definition at line 74 of file vdomnodemodel.cpp.

4.3.3.4 const QDomNode & VDomNodeModel::getDomNode ()

domNode access function.

Returns:
The member QDomNode.

Definition at line 337 of file vdomnodemodel.cpp.

References domNode.

Referenced by VSettings::writeModuleName().

4.3.3.5 QVariant VDomNodeModel::headerData (int *section*, Qt::Orientation *orientation*, int *role* = `Qt::DisplayRole`) const

Sets the Model header data for use in the associated View.

Parameters:
section Section index value.

orientation Qt::Orientation value for View.

role Represents the role of each Item in the Model.

Returns:
QVariant representing any value set in the function.

Definition at line 99 of file vdomnodemodel.cpp.

4.3.3.6 QModelIndex VDomNodeModel::index (int *row*, int *column*, const QModelIndex & *parent* = `QModelIndex()`) const

Returning the QModelIndex for a given Item after calling the child-functions of each Item.

Parameters:
row Row value.

column Column value.

parent Parent of the Item to index.

Returns:
A QModelIdex object holding the internal pointer representing the QDomNode.

Definition at line 133 of file vdomnodemodel.cpp.

References VDomNodeItem::child(), and rootItem.

Here is the call graph for this function:

4.3.3.7 QModelIndex VDomNodeModel::parent (const QModelIndex & *child*) const

Returns the parent QModelIndex for a child QModelIndex.

Parameters:
child The child QModelIndex.

Returns:
The parent QModelIndex.

Definition at line 191 of file vdomnodemodel.cpp.

References VDomNodeItem::parent(), and rootItem.

Here is the call graph for this function:

4.3.3.8 void VDomNodeModel::parseElementSignal (QDomElement & *e*, bool *update*) [signal]

Referenced by VDomNodeModel().

4.3.3.9 void VDomNodeModel::removeDuplicateConnectionsSignal (const QDomElement & *el*) [signal]

4.3.3.10 int VDomNodeModel::rowCount (const QModelIndex & *parent* = QModelIndex()) const

Returns the number of rows in the model.

Parameters:
parent The parent QModelIndex.

Returns:
The number of rows.

Definition at line 161 of file vdomnodemodel.cpp.

References attributes, VDomNodeItem::node(), and rootItem.

Here is the call graph for this function:

4.3.3.11 bool VDomNodeModel::setData (const QModelIndex & *index*, const QVariant & *value*, int *role* = Qt::EditRole)

Function for setting the values in the underlying data structure when editing the View.

Parameters:

index The QModelIndex to be edited from View.

value The value received from the View.

role The role of the QModelIndex to be edited.

Returns:

Boolean which states if the editing of the Item was successful.

Definition at line 266 of file vdomnodemodel.cpp.

4.3.3.12 void VDomNodeModel::updateConnectionsSignal () [signal]

Referenced by VDomNodeModel().

4.3.4 Member Data Documentation

4.3.4.1 QDomNamedNodeMap VDomNodeModel::attributes [private]

The attributes arranged in a QDomNamedNodeMap.

Definition at line 63 of file vdomnodemodel.h.

Referenced by data(), and rowCount().

4.3.4.2 QDomNode VDomNodeModel::domNode [private]

The DOM node.

Definition at line 57 of file vdomnodemodel.h.

Referenced by getDomNode(), and VDomNodeModel().

4.3.4.3 QObject* VDomNodeModel::parentWidget [private]

The model's parent widget.

Definition at line 75 of file vdomnodemodel.h.

Referenced by VDomNodeModel().

4.3.4.4 VDomNodeItem* VDomNodeModel::rootItem [private]

The root item for the model.

Definition at line 69 of file vdomnodemodel.h.

Referenced by index(), parent(), rowCount(), VDomNodeModel(), and ~VDomNodeModel().

The documentation for this class was generated from the following files:

- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vdomnodemodel.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vdomnodemodel.cpp**

4.4 VMainView Class Reference

Main View class represents the main window including the VTK module intialization functions.

`#include <vmainview.h>`

Public Slots

- virtual void **fileOpen** ()

  Defines actions to be taken upon file open .

- void **fileNew** ()

  Create a new empty XML-file with the default contents.

- virtual void **fileExit** ()

  Handles shutdown actions.

- void **createModules** ()

  Function reads through the DOM-tree to find the module-element then passes the element to function parseElement in order to create the VTK scene.

- void **connectModules** ()

  Connect the different VTK modules stored in its respective lists based on the list of connections.

- void **clearScene** ()

  Clears the scene of all VTK objects and connections.

- void **printConnections** ()

  Prints the connections contained in the connections-list.

- void **renderScene** ()

  VTK function calls to render all "Props" in the scene.

- void **removeDuplicateConnections** (const QDomElement &el)

  Removes duplicate connections in XML DOM-element.

- void **parseElement** (QDomElement &el, bool update)

  Function that parses each VTK module DOM-element in order to instantiate VTK objects and call the property functions corresponding to the patterns defined in XML document.

Signals

- void **updatePipeline** ()

Public Member Functions

- **VMainView** (QWidget *parent=0)

  Constructor for initializing the manu items and VTK sub-system.

- ~**VMainView** ()
  Destructor deletes VTK objects.
- void **setHighlightedModule** (const **VModule** &module)
- QString & **getCurrentFilename** ()
  currentFilename access function.
- QList< vtkObject * > & **getSources** ()
  Sources access function. Returns the list of instantiated sources.
- QList< vtkObject * > & **getFilters** ()
  Filters access function. Returns the list of instantiated filters.
- QList< vtkObject * > & **getMappers** ()
  Mappers access function. Returns the list of instantiated mappers.
- QList< vtkObject * > & **getActors** ()
  Actors access function. Returns the list of instantiated actors.
- QList< **VConnection** * > & **getConnections** ()
  Connections access function. Returns the list of instantiated connections.

Private Attributes

- vtkPropCollection * **props**
  Collection of VTK "Props".
- QList< vtkObject * > **sources**
  List of VTK Sources.
- QList< vtkObject * > **filters**
  List of VTK Filters.
- QList< vtkObject * > **mappers**
  List of VTK Mappers.
- QList< vtkObject * > **actors**
  List of VTK Actors.
- QList< **VConnection** * > **connections**
  List of connection objects.
- QString **currentFilename**
  String holding the currently active file name to read the XML document from.
- vtkRenderer * **ren**
  The VTK renderer object.
- vtkGenericRenderWindowInteractor * **interact**

VTK render window interactor object.

- vtkFrustumCoverageCuller * **culler**
  Used for culling VTK actors.

4.4.1 Detailed Description

Main View class represents the main window including the VTK module intialization functions.

Class represents the main window including menubar, statusbar and VTK functionality. The VTK modules are intialized through functions parsing the XML-document that represents the scene and visuaization pipeline.

Definition at line 42 of file vmainview.h.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 VMainView::VMainView (QWidget * *parent* = 0)

Constructor for initializing the manu items and VTK sub-system.

Parameters:
parent Parent widget.

Definition at line 60 of file vmainview.cpp.

References connectModules(), createModules(), currentFilename, fileExit(), fileNew(), fileOpen(), VSettings::getModuleName(), props, ren, renderScene(), and updatePipeline().

Here is the call graph for this function:

4.4.2.2 VMainView::~VMainView ()

Destructor deletes VTK objects.

Definition at line 141 of file vmainview.cpp.

References clearScene(), props, and ren.

4.4.3 Member Function Documentation

4.4.3.1 void VMainView::clearScene () [slot]

Clears the scene of all VTK objects and connections.

Definition at line 508 of file vmainview.cpp.

References sources.

Referenced by createModules(), and ~VMainView().

4.4.3.2 void VMainView::connectModules () [slot]

Connect the different VTK modules stored in its respective lists based on the list of connections.

Definition at line 284 of file vmainview.cpp.

References connections, and sources.

Referenced by fileNew(), fileOpen(), and VMainView().

4.4.3.3 void VMainView::createModules () [slot]

Function reads through the DOM-tree to find the module-element then passes the element to function parseElement in order to create the VTK scene.

Definition at line 245 of file vmainview.cpp.

References clearScene(), and currentFilename.

Referenced by fileNew(), fileOpen(), and VMainView().

4.4.3.4 void VMainView::fileExit () [virtual, slot]

Handles shutdown actions.

Definition at line 557 of file vmainview.cpp.

Referenced by VMainView().

4.4.3.5 void VMainView::fileNew () [slot]

Create a new empty XML-file with the default contents.

Definition at line 223 of file vmainview.cpp.

References connectModules(), createModules(), currentFilename, getCurrentFilename(), renderScene(), and updatePipeline().

Referenced by VMainView().

4.4.3.6 void VMainView::fileOpen () [virtual, slot]

Defines actions to be taken upon file open .

Definition at line 157 of file vmainview.cpp.

References connectModules(), createModules(), currentFilename, props, renderScene(), and updatePipeline().

Referenced by VMainView().

4.4.3.7 QList< vtkObject * > & VMainView::getActors ()

Actors access function. Returns the list of instantiated actors.

Returns:
The VTK "actor" objects.

Definition at line 1325 of file vmainview.cpp.

References actors.

4.4.3.8 QList< VConnection * > & VMainView::getConnections ()

Connections access function. Returns the list of instantiated connections.

Returns:
The list of **VConnection**(p. 7) objects.

Definition at line 1336 of file vmainview.cpp.

References connections.

4.4.3.9 QString & VMainView::getCurrentFilename ()

currentFilename access function.

Returns:
The current filename of loaded XML-file.

Definition at line 1281 of file vmainview.cpp.

References currentFilename.

Referenced by VSettings::buildWidgets(), VPipeline::clearConnections(), VPipeline::createNewConnection(), fileNew(), VPipeline::initModules(), VPipeline::numberOfConnections(), VPipeline::removeModule(), VSettings::setBackgroundColor(), VSettings::VSettings(), VSettings::writeModuleName(), and VPipeline::writePosition().

4.4.3.10 QList< vtkObject * > & VMainView::getFilters ()

Filters access function. Returns the list of instantiated filters.

Returns:
The VTK "filter" objects.

Definition at line 1303 of file vmainview.cpp.

References filters.

4.4.3.11 QList< vtkObject * > & VMainView::getMappers ()

Mappers access function. Returns the list of instantiated mappers.

Returns:
The VTK "mapper" objects.

Definition at line 1314 of file vmainview.cpp.

References mappers.

4.4.3.12 QList< vtkObject * > & VMainView::getSources ()

Sources access function. Returns the list of instantiated sources.

Returns:
The VTK "source" objects.

Definition at line 1292 of file vmainview.cpp.

References sources.

4.4.3.13 void VMainView::parseElement (QDomElement & *e*, bool *update*) [slot]

Function that parses each VTK module DOM-element in order to instantiate VTK objects and call the property functions corresponding to the patterns defined in XML document.

Parameters:
e The DOM-element to be parsed.

update States whether the element is to be instantiated or just updated if it already exists.

Definition at line 573 of file vmainview.cpp.

References sources.

4.4.3.14 void VMainView::printConnections () [slot]

Prints the connections contained in the connections-list.

Definition at line 473 of file vmainview.cpp.

References connections.

Referenced by removeDuplicateConnections().

4.4.3.15 void VMainView::removeDuplicateConnections (const QDomElement & *e*) [slot]

Removes duplicate connections in XML DOM-element.

Parameters:
e DOM-element.

Definition at line 1249 of file vmainview.cpp.

References connections, and printConnections().

4.4.3.16 void VMainView::renderScene () [slot]

VTK function calls to render all "Props" in the scene.

Definition at line 490 of file vmainview.cpp.

References props.

Referenced by fileNew(), fileOpen(), and VMainView().

4.4.3.17 void VMainView::setHighlightedModule (const VModule & *module*)

4.4.3.18 void VMainView::updatePipeline () [signal]

Referenced by fileNew(), fileOpen(), and VMainView().

4.4.4 Member Data Documentation

4.4.4.1 QList<vtkObject*> VMainView::actors [private]

List of VTK Actors.

Definition at line 103 of file vmainview.h.

Referenced by getActors().

4.4.4.2 QList<VConnection*> VMainView::connections [private]

List of connection objects.

Definition at line 109 of file vmainview.h.

Referenced by connectModules(), getConnections(), printConnections(), and removeDuplicate-Connections().

4.4.4.3 vtkFrustumCoverageCuller* VMainView::culler [private]

Used for culling VTK actors.

Definition at line 135 of file vmainview.h.

4.4.4.4 QString VMainView::currentFilename [private]

String holding the currently active file name to read the XML document from.

Definition at line 116 of file vmainview.h.

Referenced by createModules(), fileNew(), fileOpen(), getCurrentFilename(), and VMainView().

4.4.4.5 QList<vtkObject*> VMainView::filters [private]

List of VTK Filters.

Definition at line 91 of file vmainview.h.

Referenced by getFilters().

4.4.4.6 vtkGenericRenderWindowInteractor* VMainView::interact [private]

VTK render window interactor object.

Definition at line 129 of file vmainview.h.

4.4.4.7 QList<vtkObject*> VMainView::mappers [private]

List of VTK Mappers.

Definition at line 97 of file vmainview.h.

Referenced by getMappers().

4.4.4.8 vtkPropCollection* VMainView::props [private]

Collection of VTK "Props".

Definition at line 79 of file vmainview.h.

Referenced by fileOpen(), renderScene(), VMainView(), and ~VMainView().

4.4.4.9 vtkRenderer* VMainView::ren [private]

The VTK renderer object.

Definition at line 123 of file vmainview.h.

Referenced by VMainView(), and ~VMainView().

4.4.4.10 QList<vtkObject*> VMainView::sources [private]

List of VTK Sources.

Definition at line 85 of file vmainview.h.

Referenced by clearScene(), connectModules(), getSources(), and parseElement().

The documentation for this class was generated from the following files:

- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vmainview.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vmainview.cpp**

4.5 VModule Class Reference

Class representing a VTK module read from XML-document for use in **VPipeline**(p. 42).

`#include <vmodule.h>`

Public Member Functions

- **VModule** ()
  A constructor.
- void **setPath** (const QPainterPath &path)
  Sets the QPainterPath associated with the module.
- void **setToolTip** (const QString &toolTip)
  Sets the tooltip string associated with the module.
- void **setPosition** (const QPoint &position)
  Sets the current position of the module in the Pipeline Widget.
- void **setColor** (const QColor &color)
  Sets the color associated with the module.
- void **setType** (const QString &string)
  Sets the attribute string myType.
- void **setName** (const QString &name)
  Sets the attribute string myName.
- void **setIndex** (const int &i)
  Sets the attribute int myIndex .
- void **setOutputSelected** (bool b)
  myOutpusSelected assignment function.
- void **setInputSelected** (bool b)
  Sets the attribute bool myInputSelected.
- void **setOutputHighlighted** (bool b)
  Sets the attribute bool myOutputHighlighted.
- void **setInputHighlighted** (bool b)
  Sets the state of myInputHighlighted.
- void **setDescription** (const QString &desc)
  Sets the attribute string myDescription.
- QPainterPath **path** () const
  myPath access function.

- QPoint **position** () const
  myPosition access funtion.
- QColor **color** () const
  myColor access function.
- QString **toolTip** () const
  myToolTip access function.
- QString **type** () const
  myType access function.
- QString **name** () const
  myName acccess function.
- QString **description** () const
  myDescription access function
- int **index** () const
  myIndex access function.
- bool **outputSelected** () const
  myOutputSelected access function.
- bool **inputSelected** () const
  myInputSelected access function.
- bool **inputHighlighted** () const
  myInputHighlighted access function
- bool **outputHighlighted** () const
  myOutputHighlighted access function

Private Attributes

- QPainterPath **myPath**
  The QPainterPath which is the graphical representation of a module.
- QPoint **myPosition**
  The point representing the position of the module.
- QColor **myColor**
  The color of the module.
- QString **myToolTip**
  The ToolTip string.
- QString **myType**

String representing the type of the current module.

- QString **myName**
  String representing the name of the current module.
- QString **myDescription**
  String representing the description of the current module.
- int **myIndex**
  Integer representing the module index.
- bool **myOutputSelected**
  Boolean which states whether the module's output is selected.
- bool **myInputSelected**
  Boolean which states whether the module's inputs is selected.
- bool **myOutputHighlighted**
  Boolean which states whether the module's output is highlighted.
- bool **myInputHighlighted**
  Boolean which states whether the module's inputs is highlighted.

4.5.1 Detailed Description

Class representing a VTK module read from XML-document for use in **VPipeline**(p. 42).
This class holds the necessary information to represent the modules in the Pipeline widget
Definition at line 19 of file vmodule.h.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 VModule::VModule ()

A constructor.
Definition at line 14 of file vmodule.cpp.

4.5.3 Member Function Documentation

4.5.3.1 QColor VModule::color () const

myColor access function.

Returns:
The color of the module

Definition at line 192 of file vmodule.cpp.
References myColor.
Referenced by VPipeline::paintEvent().

4.5.3.2 QString VModule::description () const

myDescription access function

Returns:
Decription of the module

Definition at line 249 of file vmodule.cpp.

References myDescription.

Referenced by VPipeline::paintEvent().

4.5.3.3 int VModule::index () const

myIndex access function.

Returns:
Index of the current module

Definition at line 227 of file vmodule.cpp.

References myIndex.

Referenced by VPipeline::mousePressEvent().

4.5.3.4 bool VModule::inputHighlighted () const

myInputHighlighted access function

Returns:
States whether the module's input is highlighted

Definition at line 295 of file vmodule.cpp.

References myInputHighlighted.

Referenced by VPipeline::paintEvent().

4.5.3.5 bool VModule::inputSelected () const

myInputSelected access function.

Returns:
States whether the module's input is selected.

Definition at line 272 of file vmodule.cpp.

References myInputSelected.

Referenced by VPipeline::paintEvent().

4.5.3.6 QString VModule::name () const

myName acccess function.

Returns:
String representing the name of the module.

Definition at line 238 of file vmodule.cpp.

References myName.

Referenced by VPipeline::mousePressEvent().

4.5.3.7 bool VModule::outputHighlighted () const

myOutputHighlighted access function

Returns:
States whehter the module's output is highlighted

Definition at line 284 of file vmodule.cpp.

References myOutputHighlighted.

Referenced by VPipeline::paintEvent().

4.5.3.8 bool VModule::outputSelected () const

myOutputSelected access function.

Returns:
States whether the module's output is selected.

Definition at line 260 of file vmodule.cpp.

References myOutputSelected.

Referenced by VPipeline::mousePressEvent(), and VPipeline::paintEvent().

4.5.3.9 QPainterPath VModule::path () const

myPath access function.

Returns:
the QPainterPath myPath.

Definition at line 170 of file vmodule.cpp.

References myPath.

Referenced by VPipeline::paintEvent().

4.5.3.10 QPoint VModule::position () const

myPosition access funtion.

Returns:
The postition of the module.

Definition at line 181 of file vmodule.cpp.

References myPosition.

Referenced by VPipeline::insideInput(), VPipeline::insideModule(), VPipeline::insideOutput(), VPipeline::moveModuleTo(), and VPipeline::paintEvent().

4.5.3.11 void VModule::setColor (const QColor & *color*)

Sets the color associated with the module.

Parameters:
color QColor associated with the module.

Definition at line 62 of file vmodule.cpp.

References myColor.

Referenced by VPipeline::createModule().

4.5.3.12 void VModule::setDescription (const QString & *desc*)

Sets the attribute string myDescription.

Parameters:
desc Input string.

Definition at line 110 of file vmodule.cpp.

References myDescription.

Referenced by VPipeline::createModule().

4.5.3.13 void VModule::setIndex (const int & *i*)

Sets the attribute int myIndex .

Parameters:
i Input int.

Definition at line 85 of file vmodule.cpp.

References myIndex.

Referenced by VPipeline::createModule().

4.5.3.14 void VModule::setInputHighlighted (bool *b*)

Sets the state of myInputHighlighted.

Parameters:
b Input boolean.

Definition at line 157 of file vmodule.cpp.

References myInputHighlighted.

4.5.3.15 void VModule::setInputSelected (bool *b*)

Sets the attribute bool myInputSelected.

Parameters:
b Input boolean.

Definition at line 133 of file vmodule.cpp.

References myInputSelected.

4.5.3.16 void VModule::setName (const QString & *string*)

Sets the attribute string myName.

Parameters:
string Input string

Definition at line 98 of file vmodule.cpp.

References myName.

Referenced by VPipeline::createModule().

4.5.3.17 void VModule::setOutputHighlighted (bool *b*)

Sets the attribute bool myOutputHighlighted.

Parameters:
b Input boolean.

Definition at line 145 of file vmodule.cpp.

References myOutputHighlighted.

4.5.3.18 void VModule::setOutputSelected (bool *b*)

myOutpusSelected assignment function.

Parameters:
b Input boolean.

Definition at line 121 of file vmodule.cpp.

References myOutputSelected.

4.5.3.19 void VModule::setPath (const QPainterPath & *path*)

Sets the QPainterPath associated with the module.

Parameters:
path QPainter path associated with the module.

Definition at line 27 of file vmodule.cpp.

References myPath.

Referenced by VPipeline::createModule().

4.5.3.20 void VModule::setPosition (const QPoint & *position*)

Sets the current position of the module in the Pipeline Widget.

Parameters:
position Input position.

Definition at line 51 of file vmodule.cpp.

References myPosition.

Referenced by VPipeline::createModule(), and VPipeline::moveModuleTo().

4.5.3.21 void VModule::setToolTip (const QString & *toolTip*)

Sets the tooltip string associated with the module.

Parameters:
toolTip Tooltip string.

Definition at line 38 of file vmodule.cpp.

References myToolTip.

Referenced by VPipeline::createModule().

4.5.3.22 void VModule::setType (const QString & *string*)

Sets the attribute string myType.

Parameters:
string Input string.

Definition at line 74 of file vmodule.cpp.

References myType.

Referenced by VPipeline::createModule().

4.5.3.23 QString VModule::toolTip () const

myToolTip access function.

Returns:
The module's ToolTip string.

Definition at line 203 of file vmodule.cpp.

References myToolTip.

4.5.3.24 QString VModule::type () const

myType access function.

Returns:
String representing the type of the module.

Definition at line 215 of file vmodule.cpp.

References myType.

4.5.4 Member Data Documentation

4.5.4.1 QColor VModule::myColor [private]

The color of the module.

Definition at line 69 of file vmodule.h.

Referenced by color(), and setColor().

4.5.4.2 QString VModule::myDescription [private]

String representing the description of the current module.

Definition at line 89 of file vmodule.h.

Referenced by description(), and setDescription().

4.5.4.3 int VModule::myIndex [private]

Integer representing the module index.

Definition at line 94 of file vmodule.h.

Referenced by index(), and setIndex().

4.5.4.4 bool VModule::myInputHighlighted [private]

Boolean which states whether the module's inputs is highlighted.

Definition at line 114 of file vmodule.h.

Referenced by inputHighlighted(), and setInputHighlighted().

4.5.4.5 bool VModule::myInputSelected [private]

Boolean which states whether the module's inputs is selected.

Definition at line 104 of file vmodule.h.

Referenced by inputSelected(), and setInputSelected().

4.5.4.6 QString VModule::myName [private]

String representing the name of the current module.

Definition at line 84 of file vmodule.h.

Referenced by name(), and setName().

4.5.4.7 bool VModule::myOutputHighlighted [private]

Boolean which states whether the module's output is highlighted.

Definition at line 109 of file vmodule.h.

Referenced by outputHighlighted(), and setOutputHighlighted().

4.5.4.8 bool VModule::myOutputSelected [private]

Boolean which states whether the module's output is selected.

Definition at line 99 of file vmodule.h.

Referenced by outputSelected(), and setOutputSelected().

4.5.4.9 QPainterPath VModule::myPath [private]

The QPainterPath which is the graphical representation of a module.

Definition at line 57 of file vmodule.h.

Referenced by path(), and setPath().

4.5.4.10 QPoint VModule::myPosition [private]

The point representing the position of the module.

Definition at line 63 of file vmodule.h.

Referenced by position(), and setPosition().

4.5.4.11 QString VModule::myToolTip [private]

The ToolTip string.

Definition at line 74 of file vmodule.h.

Referenced by setToolTip(), and toolTip().

4.5.4.12 QString VModule::myType [private]

String representing the type of the current module.

Definition at line 79 of file vmodule.h.

Referenced by setType(), and type().

The documentation for this class was generated from the following files:

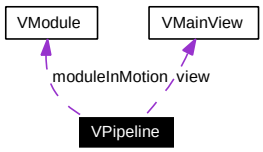
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vmodule.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vmodule.cpp**

4.6 VPipeline Class Reference

VPipeline(p. 42) is a class that represents the contents of the Pipeline widget in the application.

`#include <vpipeline.h>`

Collaboration diagram for VPipeline:



Signals

- void **updateSettings** (const QString &type, const int &id)

Public Member Functions

- **VPipeline** (QWidget *parent, **VMainView** &viewRef)

  Cosntructor which sets attributes and creates the grapichal representations of the modules.

Protected Member Functions

- bool **event** (QEvent *event)

  Handles events.

- void **resizeEvent** (QResizeEvent *event)

  Handles the resize event.

- void **paintEvent** (QPaintEvent *event)

  Handles the paint event.

- void **mousePressEvent** (QMouseEvent *event)

  Handles the event when a mouse button is pressed.

- void **mouseMoveEvent** (QMouseEvent *event)

  Handles the mouse move event.

- void **mouseReleaseEvent** (QMouseEvent *event)

  Handles the event when a mouse button is released.

- void **writePosition** (const QString &type, const int &index, const QPoint &point)

  Function that records the position of a module and writes its coordinated to the XML-document.

- void **contextMenuEvent** (QContextMenuEvent *event)

  Context menu event handler.

- void **createNewConnection** (const QString &toName, int toId, const QString &fromName, int fromId)

  A slot that is activated when a new connection is created in the Pipeline widget by dragging a link between two modules.

- void **createMenu** ()

  Generate the menu used as a context menu and in the top menu bar in the main view. The menu contains the actions for adding a new module based on existing templates for modules supported by the system and an actiopn for deleting modules. The templates for adding a new module are located in the file template/ModuleTemplates.xml and the menu created is generated from this file.

- void **renewModuleIds** (QDomElement &element)

  Function examines the DOM-structure to renew module IDs so that they are incremental for the parsing function in **VMainView**(p. 23) *to work.*

- void **clearConnections** (const QString &name, int id)

  Function which clears all connections stored in a module.

- int **numberOfConnections** (const QString &name, int id)

  Function that returns the number of connections a module recieves.

Private Slots

- void **createNewSource** ()

  A slot that is activated when a new source is created from the context menu. Calls the createModule function.

- void **createNewFilter** ()

  A slot that is activated when a new filter is created from the context menu. Calls the createModule function.

- void **createNewMapper** ()

  A slot that is activated when a new mappers is created from the context menu. Calls the createModule function.

- void **createNewActor** ()

  A slot that is activated when a new actor is created from the context menu. Calls the createModule function.

- void **initModules** ()

  This function examines the XML-document represented as a DOM tree in order to find the modules and their positions so that they can be drawn in the widget.

- void **handleAddModuleAction** (QAction *action)

  Function handles an action send when a menu item is selected in order to add a new Module. When a new module of a given type is activated from the menu the function adds an element with the new module in the XML representation.

- void **removeModule** ()

  Function removes a module from the XML - document based on the delete action from the menu.

Private Member Functions

- void **createModule** (const QPainterPath &path, const QString &toolTip, const QPoint &pos, const QColor &color, const QString &type, const int &index, const QString &name, const QString &description)

  Function creates a graphical module representation in the Pipeline widget .

- int **moduleAt** (const QPoint &pos)

  Returns the index in the list of modules of the module found at the position passed as an argument to this function.

- void **moveModuleTo** (const QPoint &pos)

  Function handling the moving of a module to a new position in the widget.

- bool **insideModule** (const **VModule** &module, const QPoint &pos)

  Function checks whether a given mouse position is inside the module or not.

- bool **insideInput** (const **VModule** &module, const QPoint &pos)

  Function checks if a position is inside a module input field or not.

- bool **insideOutput** (const **VModule** &module, const QPoint &pos)

  Function checks if a position is inside a module output field or not.

- QPoint **initialModulePosition** (const QDomElement &el)

  Function for placing the modules is the right place in the widget.

- QPoint **randomModulePosition** ()

  Returns a random position in the case of adding a new module or when the position has not yet been written to file.

- QColor **initialModuleColor** ()

  Write brief comment for initialModuleColor here.

- QColor **randomModuleColor** ()

  Returns a random color for a module.

Private Attributes

- QList< **VModule** > **modules**

  List of modules.

- QActionGroup * **moduleActionGroup**

  Action group for actions with the same action handler slot.

- QPainterPath **sourcePath**

  The graphical source path pattern.

- QPainterPath **filterPath**

  The graphical filter path pattern.

- QPainterPath **mapperPath**

  The graphical mapper path pattern.

- QPainterPath **actorPath**

  The graphical actor path pattern.

- QPoint **previousPosition**

  The previous position of a module.

- QPoint **cursorPosition**

  The cursor's position.

- **VModule** * **moduleInMotion**

  Write brief comment for modules here.

- QString **selectedModuleName**

  The name of a selected module.

- int **selectedModuleId**

  The ID of a selected module.

- QString **newModuleType**

  The type of a new module.

- QMenu * **contextMenu**

  The context menu which is activated by clicking the right mouse button over the pipeline widget.

- QMenu * **pipelineEdit**

  Edit menu which is added to the main view's menu bar.

- **VMainView** & **view**

  A reference to the main view object.

- const QSize **moduleSize**

  The Size of a module.

4.6.1 Detailed Description

VPipeline(p. 42) is a class that represents the contents of the Pipeline widget in the application.

The pipeline widget consists of a graphical network representation of the VTK visualization pipeline. This includes functionality for connecting module and also delting and adding new modules to the network

Definition at line 23 of file vpipeline.h.

4.6.2 Constructor & Destructor Documentation

4.6.2.1 VPipeline::VPipeline (QWidget * *parent*, VMainView & *viewRef*)

Cosntructor which sets attributes and creates the grapichal representations of the modules.

Parameters:

parent Parent Widget.

viewRef Reference to the main view.

Definition at line 23 of file vpipeline.cpp.

References actorPath, createMenu(), filterPath, mapperPath, moduleSize, selectedModuleId, selectedModuleName, and sourcePath.

Here is the call graph for this function:

4.6.3 Member Function Documentation

4.6.3.1 void VPipeline::clearConnections (const QString & *name*, int *id*) [protected]

Function which clears all connections stored in a module.

Parameters:

name Name of Module.

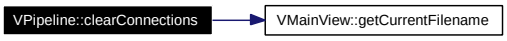
id The module's id.

Definition at line 1121 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by mousePressEvent().

Here is the call graph for this function:



4.6.3.2 void VPipeline::contextMenuEvent (QContextMenuEvent * *event*) [protected]

Context menu event handler.

Parameters:

event The context menu event.

Definition at line 834 of file vpipeline.cpp.

References contextMenu.

4.6.3.3 void VPipeline::createMenu () [protected]

Generate the menu used as a context menu and in the top menu bar in the main view. The menu contains the actions for adding a new module based on existing templates for modules supported by the system and an actiopn for deleting modules. The templates for adding a new module are located in the file template/ModuleTemplates.xml and the menu created is generated from this file.

Definition at line 702 of file vpipeline.cpp.

References contextMenu, moduleActionGroup, pipelineEdit, removeModule(), and view.

Referenced by VPipeline().

4.6.3.4 void VPipeline::createModule (const QPainterPath & *path*, const QString & *toolTip*, const QPoint & *pos*, const QColor & *color*, const QString & *type*, const int & *index*, const QString & *name*, const QString & *description*) [private]

Function creates a graphical module representation in the Pipeline widget .

Parameters:

path QPainterPath used to paint the module in the widget.

toolTip Tooltip string associated with the module.

pos The module's position.

color The module's color.

type The module's type.

index The module's index number.

name The module's name.

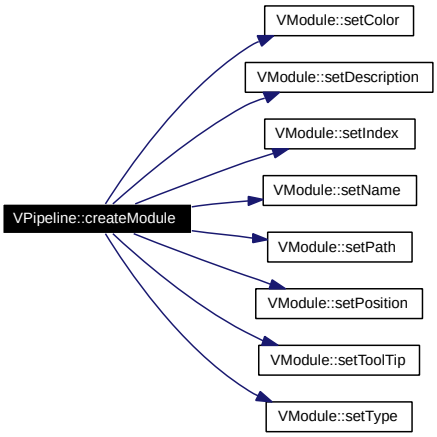
description A brief textual description of the module.

Definition at line 471 of file vpipeline.cpp.

References modules, VModule::setColor(), VModule::setDescription(), VModule::setIndex(), VModule::setName(), VModule::setPath(), VModule::setPosition(), VModule::setToolTip(), and VModule::setType().

Referenced by createNewActor(), createNewFilter(), createNewMapper(), and createNewSource().

Here is the call graph for this function:



4.6.3.5 void VPipeline::createNewActor () [private, slot]

A slot that is activated when a new actor is created from the context menu. Calls the createModule function.

Definition at line 381 of file vpipeline.cpp.

References actorPath, createModule(), and randomModulePosition().

4.6.3.6 void VPipeline::createNewConnection (const QString & *toName*, int *toId*, const QString & *fromName*, int *fromId*) [protected]

A slot that is activated when a new connection is created in the Pipeline widget by dragging a link between two modules.

The function adds a new connection in the XML representation based on the connecting of modules in the Pipeline widget

Definition at line 393 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by mousePressEvent().

Here is the call graph for this function:

4.6.3.7 void VPipeline::createNewFilter () [private, slot]

A slot that is activated when a new filter is created from the context menu. Calls the createModule function.

Definition at line 362 of file vpipeline.cpp.

References createModule(), filterPath, and randomModulePosition().

4.6.3.8 void VPipeline::createNewMapper () [private, slot]

A slot that is activated when a new mappers is created from the context menu. Calls the createModule function.

Definition at line 372 of file vpipeline.cpp.

References createModule(), mapperPath, and randomModulePosition().

4.6.3.9 void VPipeline::createNewSource () [private, slot]

A slot that is activated when a new source is created from the context menu. Calls the createModule function.

Definition at line 351 of file vpipeline.cpp.

References createModule(), randomModulePosition(), and sourcePath.

4.6.3.10 bool VPipeline::event (QEvent * *event*) [protected]

Handles events.

Parameters:
event The event received.

Returns:
Boolean value.

Definition at line 64 of file vpipeline.cpp.

References moduleAt(), and modules.

Here is the call graph for this function:

4.6.3.11 void VPipeline::handleAddModuleAction (QAction * *action*) [private, slot]

Function handles an action send when a menu item is selected in order to add a new Module. When a new module of a given type is activated from the menu the function adds an element with the new module in the XML representation.

Parameters:
action The action activated when a menu item is selected. Received from the action group moduleActionGroup.

Definition at line 920 of file vpipeline.cpp.

4.6.3.12 QColor VPipeline::initialModuleColor () [private]

Write brief comment for initialModuleColor here.

Returns:
Write description of return value here.

Definition at line 567 of file vpipeline.cpp.

References modules.

4.6.3.13 QPoint VPipeline::initialModulePosition (const QDomElement & *el*) [private]

Function for placing the modules is the right place in the widget.

Parameters:
el The DOM-element that contains the information of the position of the module.

Returns:
The position of the module as stored in XML-document.

Definition at line 534 of file vpipeline.cpp.

References randomModulePosition().

Here is the call graph for this function:

4.6.3.14 void VPipeline::initModules () [private, slot]

This function examines the XML-document represented as a DOM tree in order to find the modules and their positions so that they can be drawn in the widget.

Definition at line 591 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by resizeEvent().

4.6.3.15 bool VPipeline::insideInput (const VModule & *module*, const QPoint & *pos*) [private]

Function checks if a position is inside a module input field or not.

Parameters:
module The module to check against.
pos The position to check against.

Returns:
Boolean stating whether the position is inside a module'e input field or not.

Definition at line 875 of file vpipeline.cpp.

References moduleSize, and VModule::position().

Referenced by mousePressEvent().

Here is the call graph for this function:

4.6.3.16 bool VPipeline::insideModule (const VModule & *module*, const QPoint & *pos*) [private]

Function checks whether a given mouse position is inside the module or not.

Parameters:
module The module to check against.

pos The postition to check against.

Returns:
Boolean stating whether the position is inside a module or not.

Definition at line 852 of file vpipeline.cpp.

References moduleSize, and VModule::position().

Here is the call graph for this function:

4.6.3.17 bool VPipeline::insideOutput (const VModule & *module*, const QPoint & *pos*) [private]

Function checks if a position is inside a module output field or not.

Parameters:
module The module to check against.
pos The position to check against.

Returns:
Boolean stating whether the position is inside a module'e output field or not.

Definition at line 899 of file vpipeline.cpp.

References moduleSize, and VModule::position().

Referenced by mousePressEvent().

Here is the call graph for this function:

4.6.3.18 int VPipeline::moduleAt (const QPoint & *pos*) [private]

Returns the index in the list of modules of the module found at the position passed as an argument to this function.

Parameters:
pos The position of the mouse pointer when pressing right mouse button

Returns:
The index of the module found at this mouse position. For use in the list of modules used in Pipeline.

Definition at line 497 of file vpipeline.cpp.

References modules.

Referenced by event(), mousePressEvent(), and mouseReleaseEvent().

4.6.3.19 void VPipeline::mouseMoveEvent (QMouseEvent * *event*) [protected]

Handles the mouse move event.

Parameters:

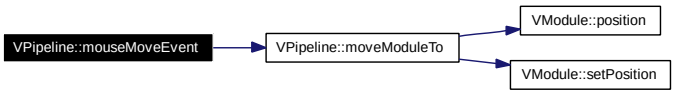
event The mouse event to be handled.

The funtcition control the motion of the modules in the Pipeline widget

Definition at line 296 of file vpipeline.cpp.

References cursorPosition, moduleInMotion, and moveModuleTo().

Here is the call graph for this function:



4.6.3.20 void VPipeline::mousePressEvent (QMouseEvent * *event*) [protected]

Handles the event when a mouse button is pressed.

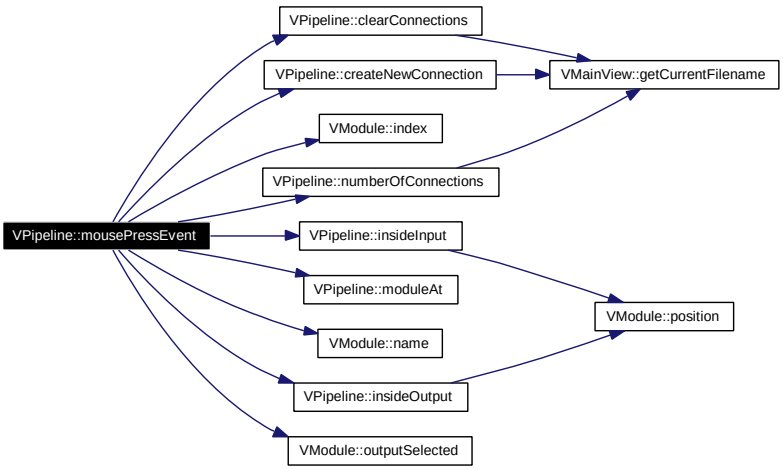
Parameters:

event The mouse event. The function handles the mouse event when a button is pressed. In this case this involves setting the attributes used for moving the modules around in the widget and connecting two widgets.

Definition at line 196 of file vpipeline.cpp.

References clearConnections(), createNewConnection(), VModule::index(), insideInput(), insideOutput(), moduleAt(), modules, VModule::name(), numberOfConnections(), and VModule::outputSelected().

Here is the call graph for this function:



4.6.3.21 void VPipeline::mouseReleaseEvent (QMouseEvent * *event*) [protected]

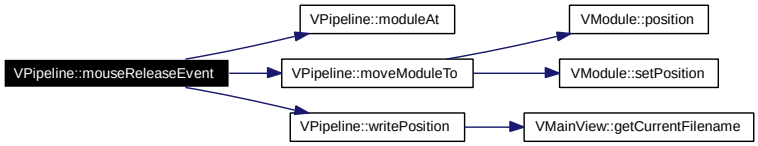
Handles the event when a mouse button is released.

Parameters:
event The mouse event to be handled.

Definition at line 338 of file vpipeline.cpp.

References moduleAt(), moduleInMotion, modules, moveModuleTo(), and writePosition().

Here is the call graph for this function:



4.6.3.22 void VPipeline::moveModuleTo (const QPoint & *pos*) [private]

Function handling the moving of a module to a new position in the widget.

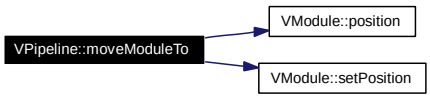
Parameters:
pos The new position of the module.

Definition at line 515 of file vpipeline.cpp.

References moduleInMotion, VModule::position(), previousPosition, and VModule::setPosition().

Referenced by mouseMoveEvent(), and mouseReleaseEvent().

Here is the call graph for this function:



4.6.3.23 int VPipeline::numberOfConnections (const QString & *name*, int *id*) [protected]

Function that returns the number of connections a module recieves.

Parameters:
name Name of Module.
id The module's id.

Definition at line 1183 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by mousePressEvent().

Here is the call graph for this function:

4.6.3.24 void VPipeline::paintEvent (QPaintEvent * *event*) [protected]

Handles the paint event.

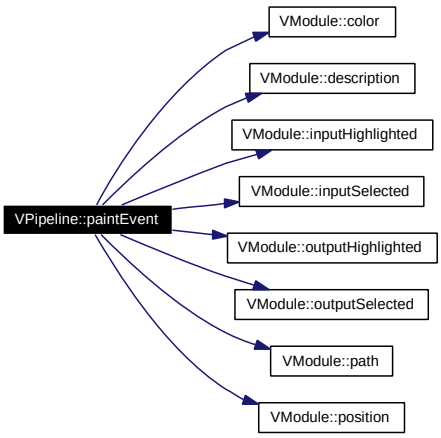
Parameters:
event The paint event.

This is where the graphical painting takes place.

Definition at line 101 of file vpipeline.cpp.

References VModule::color(), VModule::description(), VModule::inputHighlighted(), VModule::inputSelected(), modules, moduleSize, VModule::outputHighlighted(), VModule::outputSelected(), VModule::path(), and VModule::position().

Here is the call graph for this function:



4.6.3.25 QColor VPipeline::randomModuleColor () [private]

Returns a random color for a module.

Returns:
A random color.

Definition at line 579 of file vpipeline.cpp.

4.6.3.26 QPoint VPipeline::randomModulePosition () [private]

Returns a random position in the case of adding a new module or when the position has not yet been written to file.

Returns:
A random position for the module.

Definition at line 554 of file vpipeline.cpp.

Referenced by createNewActor(), createNewFilter(), createNewMapper(), createNewSource(), and initialModulePosition().

4.6.3.27 void VPipeline::removeModule () [private, slot]

Function removes a module from the XML - document based on the delete action from the menu.

Definition at line 1001 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by createMenu().

4.6.3.28 void VPipeline::renewModuleIds (QDomElement & *element*) [protected]

Function examines the DOM-structure to renew module IDs so that they are incremental for the parsing function in **VMainView**(p. 23) to work.

Parameters:
element The document element to assign new ids to.

Definition at line 1083 of file vpipeline.cpp.

4.6.3.29 void VPipeline::resizeEvent (QResizeEvent * *event*) [protected]

Handles the resize event.

Parameters:
event The resizing event.

Definition at line 83 of file vpipeline.cpp.

References initModules().

4.6.3.30 void VPipeline::updateSettings (const QString & *type*, const int & *id*) [signal]

4.6.3.31 void VPipeline::writePosition (const QString & *type*, const int & *index*, const QPoint & *point*) [protected]

Function that records the position of a module and writes its coordinated to the XML-document.

Parameters:
type The type of module (for identification).

index The module's index (for identification).

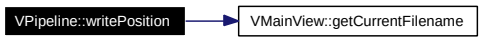
point The position of the module which is to be written to file.

Definition at line 655 of file vpipeline.cpp.

References VMainView::getCurrentFilename(), and view.

Referenced by mouseReleaseEvent().

Here is the call graph for this function:



4.6.4 Member Data Documentation

4.6.4.1 QPainterPath VPipeline::actorPath [private]

The graphical actor path pattern.

Definition at line 103 of file vpipeline.h.

Referenced by createNewActor(), and VPipeline().

4.6.4.2 QMenu* VPipeline::contextMenu [private]

The context menu which is activated by clicking the right mouse button over the pipeline widget.

Definition at line 138 of file vpipeline.h.

Referenced by contextMenuEvent(), and createMenu().

4.6.4.3 QPoint VPipeline::cursorPosition [private]

The cursor's position.

Definition at line 113 of file vpipeline.h.

Referenced by mouseMoveEvent().

4.6.4.4 QPainterPath VPipeline::filterPath [private]

The graphical filter path pattern.

Definition at line 93 of file vpipeline.h.

Referenced by createNewFilter(), and VPipeline().

4.6.4.5 QPainterPath VPipeline::mapperPath [private]

The graphical mapper path pattern.

Definition at line 98 of file vpipeline.h.

Referenced by createNewMapper(), and VPipeline().

4.6.4.6 QActionGroup* VPipeline::moduleActionGroup [private]

Action group for actions with the same action handler slot.

Definition at line 83 of file vpipeline.h.

Referenced by createMenu().

4.6.4.7 VModule* VPipeline::moduleInMotion [private]

Write brief comment for modules here.

Definition at line 118 of file vpipeline.h.

Referenced by mouseMoveEvent(), mouseReleaseEvent(), and moveModuleTo().

4.6.4.8 QList<VModule> VPipeline::modules [private]

List of modules.

Definition at line 78 of file vpipeline.h.

Referenced by createModule(), event(), initialModuleColor(), moduleAt(), mousePressEvent(), mouseReleaseEvent(), and paintEvent().

4.6.4.9 const QSize VPipeline::moduleSize [private]

The Size of a module.

Definition at line 153 of file vpipeline.h.

Referenced by insideInput(), insideModule(), insideOutput(), paintEvent(), and VPipeline().

4.6.4.10 QString VPipeline::newModuleType [private]

The type of a new module.

Definition at line 133 of file vpipeline.h.

4.6.4.11 QMenu* VPipeline::pipelineEdit [private]

Edit menu which is added to the main view's menu bar.

Definition at line 143 of file vpipeline.h.

Referenced by createMenu().

4.6.4.12 QPoint VPipeline::previousPosition [private]

The previous position of a module.

Definition at line 108 of file vpipeline.h.

Referenced by moveModuleTo().

4.6.4.13 int VPipeline::selectedModuleId [private]

The ID of a selected module.

Definition at line 128 of file vpipeline.h.

Referenced by VPipeline().

4.6.4.14 QString VPipeline::selectedModuleName [private]

The name of a selected module.

Definition at line 123 of file vpipeline.h.

Referenced by VPipeline().

4.6.4.15 QPainterPath VPipeline::sourcePath [private]

The graphical source path pattern.

Definition at line 88 of file vpipeline.h.

Referenced by createNewSource(), and VPipeline().

4.6.4.16 VMainView& VPipeline::view [private]

A reference to the main view object.

Definition at line 148 of file vpipeline.h.

Referenced by clearConnections(), createMenu(), createNewConnection(), initModules(), numberOfConnections(), removeModule(), and writePosition().

The documentation for this class was generated from the following files:

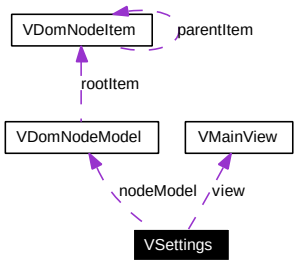
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vpipeline.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vpipeline.cpp**

4.7 VSettings Class Reference

This class represents the Settings widget that display information about an active module and handles changes to this data.

`#include <vsettings.h>`

Collaboration diagram for VSettings:



Public Slots

- void **buildWidgets** (const QString &type, const int &id)

  This function generated the sub-widgets in **VSettings**(p. 59) *based on the module. This includes the tree view and the labels and line edit.*

- void **writeModuleName** ()

  Writes the module name fetched from the QLineEdit widget to the XML-file.

- void **setBackgroundColor** ()

  Function sets the scene's background color to the output of a QColorDialog.

Public Member Functions

- **VSettings** (QWidget *parent, **VMainView** &viewRef)

  A constructor for initializing widget elements.

- ~**VSettings** ()

  Destructor.

- **VMainView** & **getView** ()

  view access function.

- QLineEdit & **getModuleName** ()

  moduleName access function

- **VDomNodeModel** & **getNodeModel** ()

  nodeModel access function.

Protected Member Functions

- bool **event** (QEvent *event)
  Event handler.
- void **resizeEvent** (QResizeEvent *event)
  Handles the resize event.
- void **paintEvent** (QPaintEvent *event)
  Handles the paint event.
- void **mousePressEvent** (QMouseEvent *event)
  Handles the mouse pressed event.
- void **mouseMoveEvent** (QMouseEvent *event)
  Handles the mouse move event.
- void **mouseReleaseEvent** (QMouseEvent *event)
  Handles the mouse release event.

Private Attributes

- **VDomNodeModel** * **nodeModel**
  The Model used as basis for the widget's QTreeView.
- **VMainView** & **view**
  A reference to the main view object.
- QTreeView * **tree**
  The tree view.
- QLabel * **nameLabel**
  A label showing the module name.
- QLineEdit * **moduleName**
  An editable text widget for editing the desciption of the module .
- QLabel * **moduleId**
  A label for showing the module's ID.
- QLabel * **moduleType**
  A label for showing the module type.
- QGridLayout * **gridLayout**
  A grid layout used to lay out the different widgets.
- QColor **backgroundColor**
  A grid layout used to lay out the different widgets.

- QToolButton **backgroundColorButton**

  Background color chooser button.

- QPixmap **bgPixmap**

  Pixmap for backgroundcolor button.

- QIcon **buttonIcon**

  Icon for backgroundcolor button.

4.7.1 Detailed Description

This class represents the Settings widget that display information about an active module and handles changes to this data.

The **VSettings**(p. 59) class consists of a QTreeView which is a part of the Model/View programming paradigm used to make the tre representation of a module's attribute. The class also includes som simple QLineEdit and QLabels for displaying information about the module.

Definition at line 24 of file vsettings.h.

4.7.2 Constructor & Destructor Documentation

4.7.2.1 VSettings::VSettings (QWidget * *parent*, VMainView & *viewRef*)

A constructor for initializing widget elements.

Parameters:

parent DThe parent widget.

viewRef A reference to the main view object

Definition at line 22 of file vsettings.cpp.

References backgroundColor, backgroundColorButton, VMainView::getCurrentFilename(), gridLayout, moduleId, moduleName, moduleType, nameLabel, nodeModel, tree, and view.

Here is the call graph for this function:

4.7.2.2 VSettings::~VSettings ()

Destructor.

Definition at line 85 of file vsettings.cpp.

4.7.3 Member Function Documentation

4.7.3.1 void VSettings::buildWidgets (const QString & *type*, const int & *id*) [slot]

This function generated the sub-widgets in **VSettings**(p. 59) based on the module. This includes the tree view and the labels and line edit.

Parameters:

type Module type string.

id Module ID int.

Definition at line 171 of file vsettings.cpp.

References VMainView::getCurrentFilename(), moduleId, moduleName, moduleType, nodeModel, tree, and view.

4.7.3.2 bool VSettings::event (QEvent * *event*) [protected]

Event handler.

Parameters:

event The event.

Returns:

Boolean.

Definition at line 100 of file vsettings.cpp.

4.7.3.3 QLineEdit & VSettings::getModuleName ()

moduleName access function

Returns:

Line Edit reference

Definition at line 316 of file vsettings.cpp.

References moduleName.

Referenced by VMainView::VMainView().

4.7.3.4 VDomNodeModel & VSettings::getNodeModel ()

nodeModel access function.

Returns:

VDomNodeModel(p. 16) reference.

Definition at line 329 of file vsettings.cpp.

References nodeModel.

4.7.3.5 VMainView & VSettings::getView ()

view access function.

Returns:
View reference.

Definition at line 303 of file vsettings.cpp.

References view.

4.7.3.6 void VSettings::mouseMoveEvent (QMouseEvent * *event*) [protected]

Handles the mouse move event.

Parameters:
event The event to be handled.

Definition at line 145 of file vsettings.cpp.

4.7.3.7 void VSettings::mousePressEvent (QMouseEvent * *event*) [protected]

Handles the mouse pressed event.

Parameters:
event The event to be handled.

Definition at line 134 of file vsettings.cpp.

4.7.3.8 void VSettings::mouseReleaseEvent (QMouseEvent * *event*) [protected]

Handles the mouse release event.

Parameters:
event The event to be handled.

Definition at line 156 of file vsettings.cpp.

4.7.3.9 void VSettings::paintEvent (QPaintEvent * *event*) [protected]

Handles the paint event.

Parameters:
event The event to be handled.

Definition at line 123 of file vsettings.cpp.

4.7.3.10 void VSettings::resizeEvent (QResizeEvent * *event*) [protected]

Handles the resize event.

Parameters:
event The event to be handled.

Definition at line 112 of file vsettings.cpp.

4.7.3.11 void VSettings::setBackgroundColor () [slot]

Function sets the scene's background color to the output of a QColorDialog.

Definition at line 248 of file vsettings.cpp.

References VMainView::getCurrentFilename(), and view.

4.7.3.12 void VSettings::writeModuleName () [slot]

Writes the module name fetched from the QLineEdit widget to the XML-file.

Definition at line 210 of file vsettings.cpp.

References VMainView::getCurrentFilename(), VDomNodeModel::getDomNode(), moduleName, nodeModel, and view.

4.7.4 Member Data Documentation

4.7.4.1 QColor VSettings::backgroundColor [private]

A grid layout used to lay out the different widgets.

Definition at line 99 of file vsettings.h.

Referenced by VSettings().

4.7.4.2 QToolButton VSettings::backgroundColorButton [private]

Background color chooser button.

Definition at line 104 of file vsettings.h.

Referenced by VSettings().

4.7.4.3 QPixmap VSettings::bgPixmap [private]

Pixmap for backgroundcolor button.

Definition at line 109 of file vsettings.h.

4.7.4.4 QIcon VSettings::buttonIcon [private]

Icon for backgroundcolor button.

Definition at line 114 of file vsettings.h.

4.7.4.5 QGridLayout* VSettings::gridLayout [private]

A grid layout used to lay out the different widgets.

Definition at line 94 of file vsettings.h.

Referenced by VSettings().

4.7.4.6 QLabel* VSettings::moduleId [private]

A label for showing the module's ID.

Definition at line 84 of file vsettings.h.

Referenced by buildWidgets(), and VSettings().

4.7.4.7 QLineEdit* VSettings::moduleName [private]

An editable text widget for editing the desciption of the module .

Definition at line 79 of file vsettings.h.

Referenced by buildWidgets(), getModuleName(), VSettings(), and writeModuleName().

4.7.4.8 QLabel* VSettings::moduleType [private]

A label for showing the module type.

Definition at line 89 of file vsettings.h.

Referenced by buildWidgets(), and VSettings().

4.7.4.9 QLabel* VSettings::nameLabel [private]

A label showing the module name.

Definition at line 74 of file vsettings.h.

Referenced by VSettings().

4.7.4.10 VDomNodeModel* VSettings::nodeModel [private]

The Model used as basis for the widget's QTreeView.

Definition at line 59 of file vsettings.h.

Referenced by buildWidgets(), getNodeModel(), VSettings(), and writeModuleName().

4.7.4.11 QTreeView* VSettings::tree [private]

The tree view.

Definition at line 69 of file vsettings.h.

Referenced by buildWidgets(), and VSettings().

4.7.4.12 VMainView& VSettings::view [private]

A reference to the main view object.

Definition at line 64 of file vsettings.h.

Referenced by buildWidgets(), getView(), setBackgroundColor(), VSettings(), and writeModuleName().

The documentation for this class was generated from the following files:

- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vsettings.h**
- My Documents/Visual Studio 2005/Projects/VisualizationApp/**vsettings.cpp**

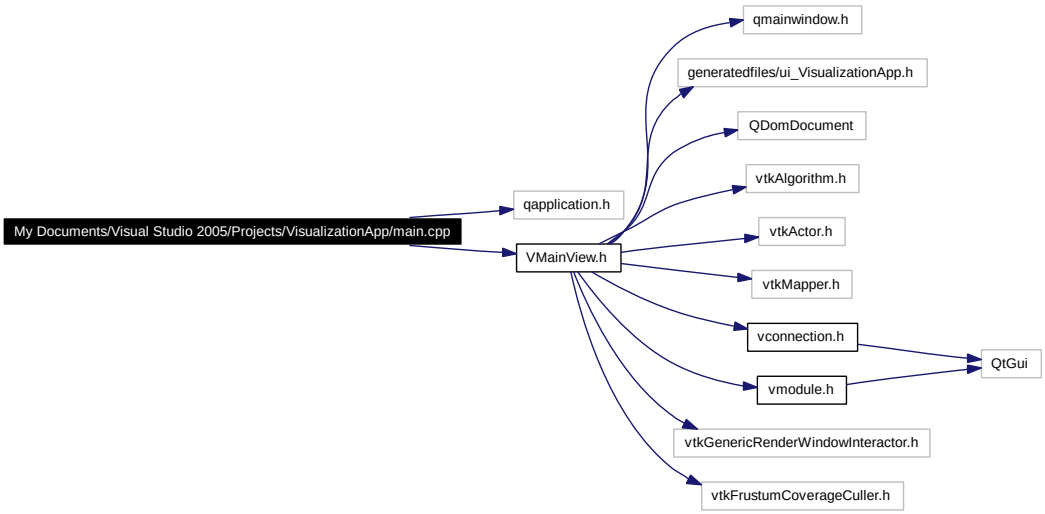
Chapter 5

VisualizationApp File Documentation

5.1 My Documents/Visual Studio 2005/Projects/VisualizationApp/main.cpp File Reference

```
#include <qapplication.h>
#include "VMainView.h"
```

Include dependency graph for main.cpp:



Functions

- int **main** (int argc, char **argv)

5.1.1 Function Documentation

5.1.1.1 int main (int *argc*, char ** *argv*)

Definition at line 7 of file main.cpp.

5.2 My Documents/Visual Studio 2005/Projects/VisualizationApp/resource.h File Reference

5.3 My Documents/Visual Studio 2005/Projects/VisualizationApp/vconnection.cpp File Reference

```
#include "vconnection.h"
```

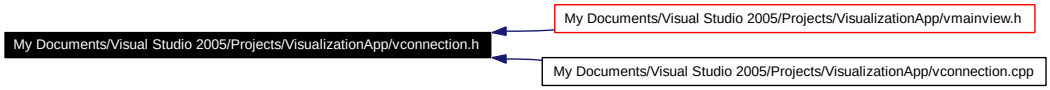
Include dependency graph for vconnection.cpp:

5.4 My Documents/Visual Studio 2005/Projects/VisualizationApp/vconnection.h File Reference

```
#include <QtGui>
```

Include dependency graph for vconnection.h:

This graph shows which files directly or indirectly include this file:



Classes

- class **VConnection**

  Class represents the connection between VTK-modules.

5.5 My Documents/Visual Studio 2005/Projects/VisualizationApp/vdomnodeitem.cpp File Reference

```
#include "vdomnodeitem.h"
```

Include dependency graph for vdomnodeitem.cpp:

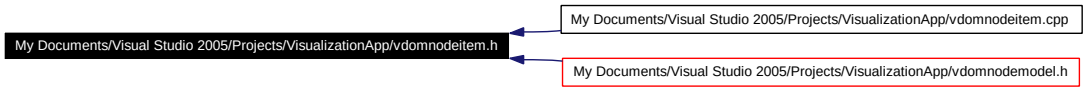
5.6 My Documents/Visual Studio 2005/Projects/VisualizationApp/vdomnodeitem.h File Reference

```
#include <QHash>
```

```
#include <QDomDocument>
```

Include dependency graph for vdomnodeitem.h:

This graph shows which files directly or indirectly include this file:



Classes

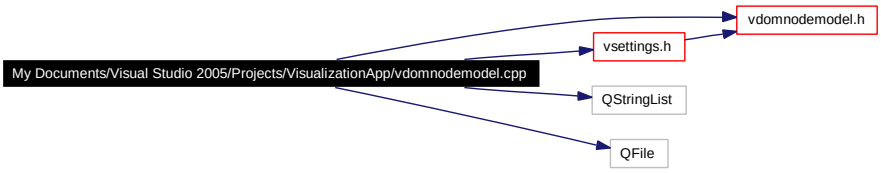
- class **VDomNodeItem**

  The **VDomNodeItem**(p. 12) *class defines the item used for the* **VDomNodeModel**(p. 16).

5.7 My Documents/Visual Studio 2005/Projects/VisualizationApp/vdomnodemodel.cpp File Reference

```
#include "vdomnodemodel.h"
#include "vsettings.h"
#include <QStringList>
#include <QFile>
```

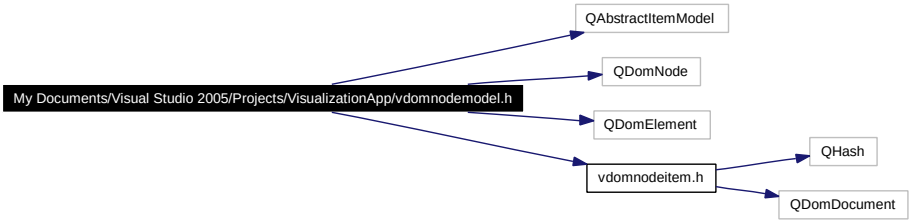
Include dependency graph for vdomnodemodel.cpp:



5.8 My Documents/Visual Studio 2005/Projects/VisualizationApp/vdomnodemodel.h File Reference

```
#include <QAbstractItemModel>
#include <QDomNode>
#include <QDomElement>
#include "vdomnodeitem.h"
```

Include dependency graph for vdomnodemodel.h:



This graph shows which files directly or indirectly include this file:

Classes

- class **VDomNodeModel**

  Model for modelling a DOM-node read from XML for use in a QTreeView.

5.9 My Documents/Visual Studio 2005/Projects/VisualizationApp/vmainview.cpp File Reference

```
#include <qapplication.h>
#include <qfiledialog.h>
#include "vpipeline.h"
#include "vsettings.h"
#include <QDomDocument>
#include <QTextStream>
#include "VMainView.h"
#include <vtkRenderer.h>
#include <vtkRenderWindow.h>
#include "vtkCylinderSource.h"
#include <vtkPolyDataMapper.h>
#include "vtkDataSetReader.h"
#include "vtkDataSetMapper.h"
#include "vtkPropCollection.h"
#include "vtkVolumeProperty.h"
#include "vtkStructuredPointsReader.h"
#include "vtkPiecewiseFunction.h"
#include "vtkColorTransferFunction.h"
#include "vtkVolumeRayCastCompositeFunction.h"
#include "vtkVolumeRayCastMapper.h"
#include "vtkFixedPointVolumeRayCastMapper.h"
#include "vtkQuadric.h"
#include "vtkSampleFunction.h"
#include "vtkProperty.h"
#include "vtkVolume16Reader.h"
#include "vtkContourFilter.h"
#include "vtkPolyDataNormals.h"
#include "vtkOutlineFilter.h"
#include "vtkStripper.h"
#include "vtkImageMapToColors.h"
#include "vtkImageActor.h"
#include "vtkLookupTable.h"
#include "vtkBoxWidget.h"
#include "vtkInteractorStyleTrackballCamera.h"
```

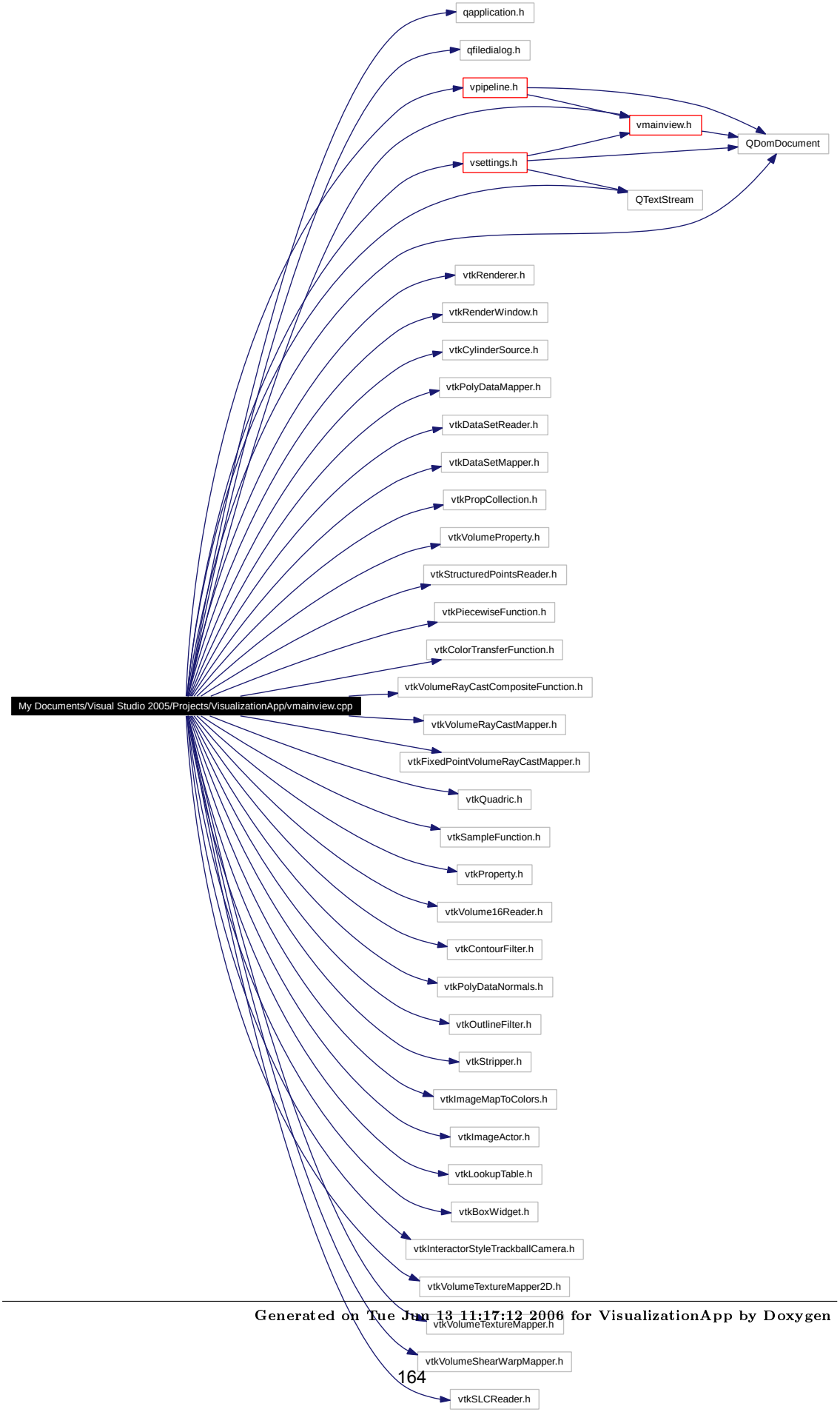
```
#include "vtkVolumeTextureMapper2D.h"

#include "vtkVolumeTextureMapper.h"

#include "vtkVolumeShearWarpMapper.h"

#include "vtkSLCReader.h"
```

Include dependency graph for vmainview.cpp:


My Documents/Visual Studio 2005/Projects/VisualizationApp/vmainview.cpp
qapplication.h
qfiledialog.h
vpipeline.h
vmainview.h
QDomDocument
vsettings.h
QTextStream
vtkRenderer.h
vtkRenderWindow.h
vtkCylinderSource.h
vtkPolyDataMapper.h
vtkDataSetReader.h
vtkDataSetMapper.h
vtkPropCollection.h
vtkVolumeProperty.h
vtkStructuredPointsReader.h
vtkPiecewiseFunction.h
vtkColorTransferFunction.h
vtkVolumeRayCastCompositeFunction.h
vtkVolumeRayCastMapper.h
vtkFixedPointVolumeRayCastMapper.h
vtkQuadric.h
vtkSampleFunction.h
vtkProperty.h
vtkVolume16Reader.h
vtkContourFilter.h
vtkPolyDataNormals.h
vtkOutlineFilter.h
vtkStripper.h
vtkImageMapToColors.h
vtkImageActor.h
vtkLookupTable.h
vtkBoxWidget.h
vtkInteractorStyleTrackballCamera.h
vtkVolumeTextureMapper2D.h
vtkVolumeTextureMapper.h
vtkVolumeShearWarpMapper.h
vtkSLCReader.h

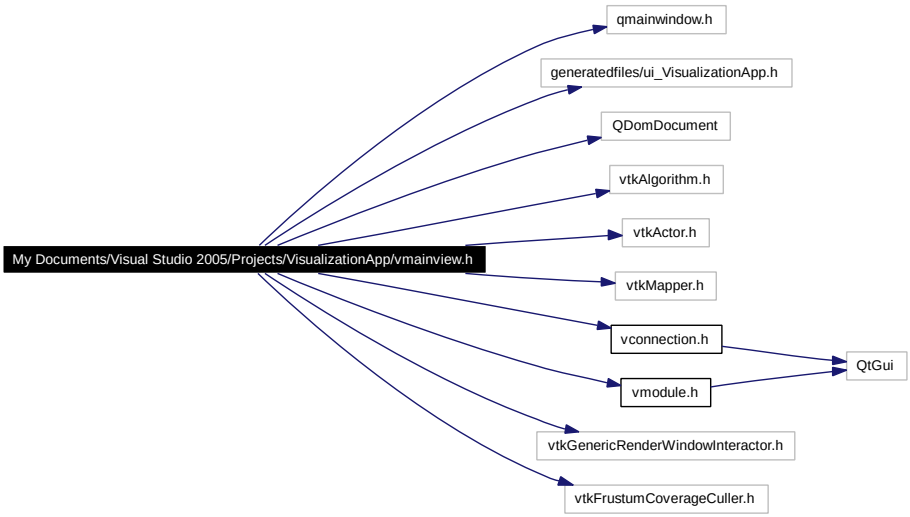
Namespaces

- namespace **std**

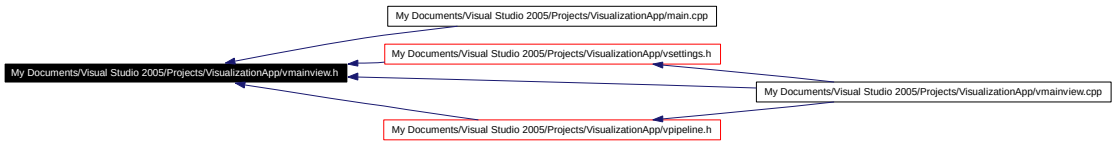
5.10 My Documents/Visual Studio 2005/Projects/VisualizationApp/vmainview.h File Reference

```
#include "qmainwindow.h"
#include "generatedfiles/ui_VisualizationApp.h"
#include <QDomDocument>
#include "vtkAlgorithm.h"
#include "vtkActor.h"
#include "vtkMapper.h"
#include "vconnection.h"
#include "vmodule.h"
#include "vtkGenericRenderWindowInteractor.h"
#include "vtkFrustumCoverageCuller.h"
```

Include dependency graph for vmainview.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **VMainView**

  Main View class represents the main window including the VTK module intialization functions.

5.11 My Documents/Visual Studio 2005/Projects/VisualizationApp/vmodule.cpp File Reference

```
#include "vmodule.h"
```

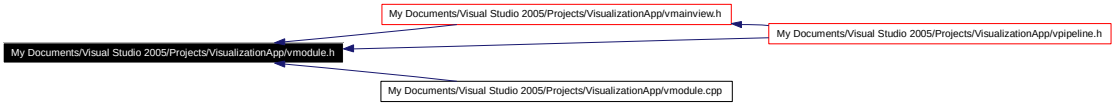
Include dependency graph for vmodule.cpp:

5.12 My Documents/Visual Studio 2005/Projects/VisualizationApp/vmodule.h File Reference

`#include <QtGui>`

Include dependency graph for vmodule.h:

This graph shows which files directly or indirectly include this file:



Classes

- class **VModule**

  Class representing a VTK module read from XML-document for use in **VPipeline**(p. 42).

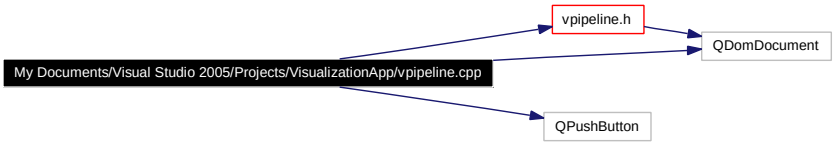
5.13 My Documents/Visual Studio 2005/Projects/VisualizationApp/vpipeline.cpp File Reference

```
#include "vpipeline.h"
```

```
#include <QDomDocument>
```

```
#include <QPushButton>
```

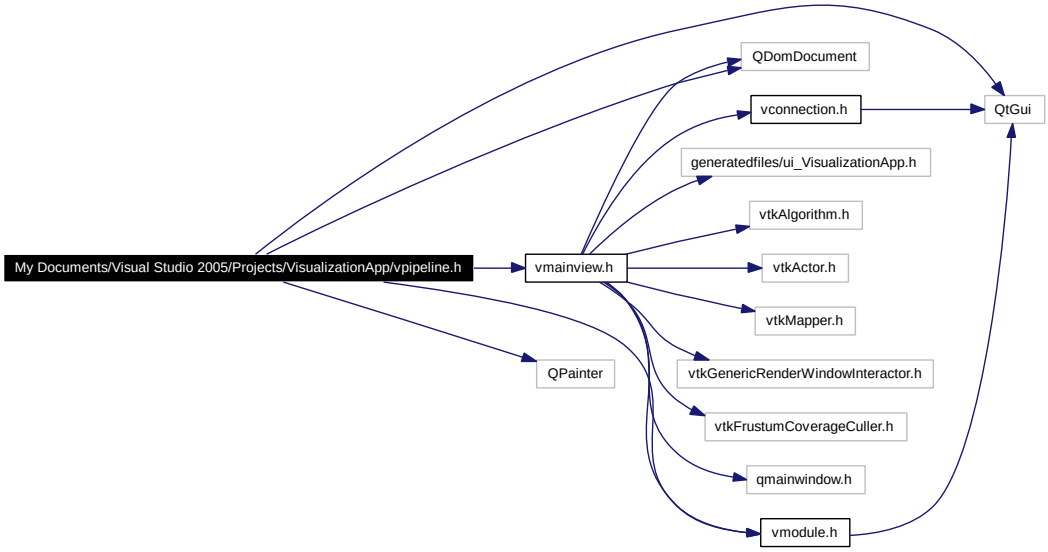
Include dependency graph for vpipeline.cpp:



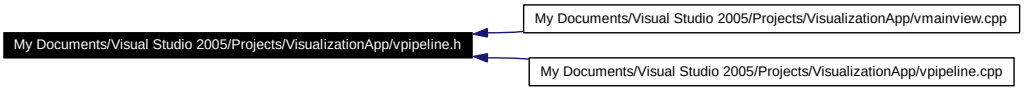
5.14 My Documents/Visual Studio 2005/Projects/VisualizationApp/vpipeline.h File Reference

```
#include <QtGui>
#include <QDomDocument>
#include <QPainter>
#include "vmodule.h"
#include "vmainview.h"
```

Include dependency graph for vpipeline.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **VPipeline**

  VPipeline(p. 42) *is a class that represents the contents of the Pipeline widget in the application.*

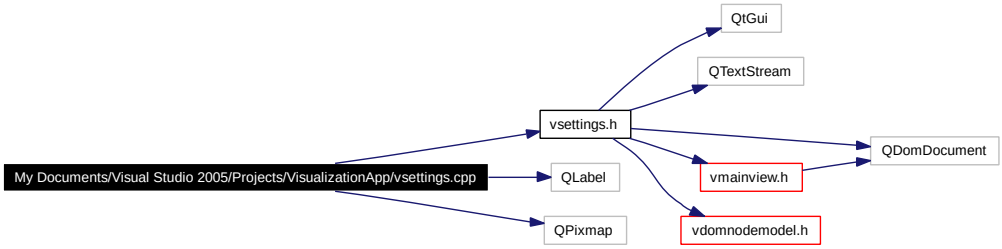
5.15 My Documents/Visual Studio 2005/Projects/VisualizationApp/vsettings.cpp File Reference

```
#include "vsettings.h"
```

```
#include <QLabel>
```

```
#include <QPixmap>
```

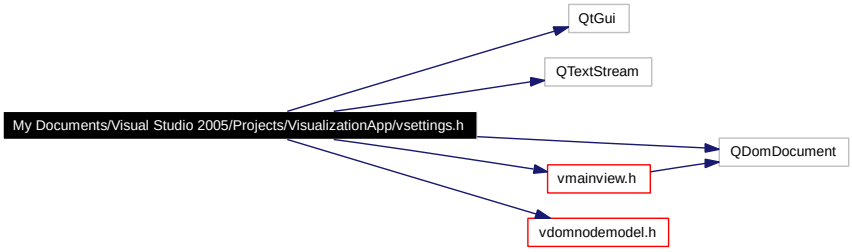
Include dependency graph for vsettings.cpp:



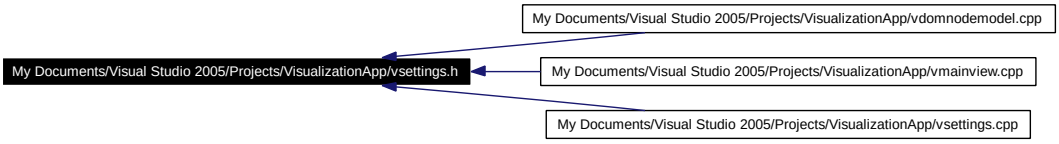
5.16 My Documents/Visual Studio 2005/Projects/VisualizationApp/vsettings.h File Reference

```
#include <QtGui>
#include <QTextStream>
#include <QDomDocument>
#include "vmainview.h"
#include "vdomnodemodel.h"
```

Include dependency graph for vsettings.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **VSettings**

  This class represents the Settings widget that display information about an active module and handles changes to this data.