Virtual Reality Interactions for Scientific Data Visualizations

Jan Schutte

June 8, 2018

Supervisor(s): dr. R.G. Belleman
Signed:
Abstract

In many fields of science large amounts of data are produced, either through simulation or through measurements. When statistical analysis on this data falls short, visualization becomes tactically important for scientific discovery. The open-source application ParaView is used for creating complex scientific visualizations. This application can greatly benefit from having an immersive environment to view visualizations. In this thesis a method is designed and implemented to create an interface where parameters of the ParaView visualization can be edited from within a Virtual Reality environment.
# Contents

1 Introduction

1.1 Related Work

1.1.1 The Visualization Toolkit

1.1.2 ParaView

1.1.3 OpenVR Plugin for ParaView

1.2 Research Question

2 Design

2.1 Hardware

2.2 User Interface

2.3 Building on top of ParaView

2.4 Properties in ParaView

3 Implementation

3.1 Input handling

3.2 Creating the pipeline representation

3.3 Creating the property widgets

3.4 Selection Mechanism

3.5 Updating the visualization

4 Results

4.1 Use case: contour visualization of a CT scan

5 Conclusions

5.1 Future Work

Appendices

A Cloning and building the plugin
Scientific research often involves measuring or generating large amounts of data. This data can be analyzed statistically but is often not possible when, for example, it is unknown what patterns or structures are present in the data [1]. Visualizing this data allows researchers to find structures and patterns in the data by hand and is a vital tool in many fields of science [13].

Some data sets are best visualized in three dimensions in which case it might make sense to view these visualizations in Virtual Reality (VR) where the viewer can benefit from the spatial awareness coming from their depth perception. Together with the rise of consumer grade VR devices, VR has the potential to become an important tool in scientific visualization [3].

The added benefit of an immersive environment is difficult to prove and proofs are often anecdotal [3]. Gruchalla set out to quantify these benefits by performing user studies. Participants were asked to perform the task of planning an oil well in both a desktop and immersive environment. A significant number of participants were not only able to perform the task faster but were also more accurate in their planning.

VR environments also allow for more intuitive interactions. Moving around, observing and picking up objects are all interactions that we already know from our own world and adapting these interactions to the virtual world is almost trivial. This makes interfaces in virtual reality easy to learn by people from all kinds of backgrounds.

Producing visualizations requires the raw data to be transformed into something that makes sense for the visualization. By adding filters points of interest can be highlighted within the data. Editing the properties of these filters is important for the creation of informative visualizations. Current VR methods for viewing visualizations do now allow for editing these properties from inside a VR environment which is the research subject of this thesis.

1.1 Related Work

The work in this thesis is built upon a couple of existing technologies for scientific visualization. The Visualization Toolkit (VTK) is a library of C++ classes that support filtering, visualizing and rendering data. ParaView is an advanced graphical tool for creating scientific visualizations and is built upon the VTK libraries [6].

1.1.1 The Visualization Toolkit

The visualization pipeline is a design approach that the creators of VTK have adapted for creating visualizations and is originally invented by Haber and McNabb [4]. In its core the pipeline consists of three elements; objects that represent data, objects that operate on data and a direc-
tion of data flow between these objects [14].

More concrete the VTK pipeline begins with a source class that is able to read a specific data format. This data can then be optionally filtered using various algorithms that fit the data format that has been read. These two class types act on the data itself, but this data still needs to be displayed by the computer. The mapper classes take the data and turn them into graphics primitives [11]. These graphics primitives can be rendered by the computer to create the final visualization.

The topology together with the parameters of the pipeline are what define a visualization and because editing these parameters is so important, having a good interface to do so is important as well. Editing these properties from a VR context would provide the advantages VR brings while the visualization is being created.

Recently the developers of VTK have made changes to VTK to make its rendering pipeline more suitable for integration with common VR systems [13]. They have achieved this in two ways; by opening up the underlying graphics framework (OpenGL) to other applications and by integrating the commonly used OpenVR library.

OpenVR is a library developed by Valve that supports common VR head mounted displays like the Oculus Rift and the HTC Vive. It also integrates with SteamVR’s ecosystem [13].

1.1.2 ParaView

ParaView is built by the developers of VTK. It is visualization software that uses VTK to create advanced visualization through a desktop application. The main benefit of ParaView is that the user does not need to know how to program in C++ to create visualizations.

In figure 1.2 the desktop environment of ParaView is shown. On the left is the pipeline browser and below it the property panel is displayed. The visualization takes the center view.
**ParaView** allows you to create a visualization pipeline without the need to know about what mapper, actor or rendering classes are required. A stage in **ParaView**’s pipeline is a bundle of various VTK pipeline stages. A stage is either a *source* or *filter* class that is matched with the right *mapper* and *actor* classes. This way any stage in the pipeline of **ParaView** is able to generate geometry for the final visualization.

### 1.1.3 OpenVR Plugin for ParaView

The developers at Kitware have taken the first steps into getting **ParaView** and VTK ready for VR [13]. Their plugin allows users to view visualizations from the desktop environment in VR using the OpenVR library. The plugin has already defined some interactions; movement around the visualization, grabbing parts of the visualization, scaling the visualization, taking measurements and adding a clipping plane to the visualization.

![Existing VR menu with added pipeline browser option.](image)

These interactions are important for inspecting the current visualization but do now allow for changes to the visualization. Adding new **ParaView** sources or filters as well as editing the parameters of these sources/filters is not possible.

Building the interactions for editing the pipeline properties on top of this plugin is a lot easier as the plugin already enables rendering the visualization in VR out of the box.

### 1.2 Research Question

This thesis describes the implementation of universal interaction methods for editing properties of the **ParaView** visualization pipeline. The research question is thus as follows:

*How can a universal interface for editing the pipeline properties of **ParaView** be created for Virtual Reality?*
In this chapter the design of the VR interface is described as well as how the implementation has to fit into the existing code.

## 2.1 Hardware

This thesis will base its design on the two most common VR headsets, the *Oculus Rift* and *HTC Vive*. These headsets follow a similar design, they have a headset and two hand controllers that are tracked in 3D space. With the help of a graphically powerful computer, the headset can render high fidelity stereoscopic imagery. The two controllers are an important part of interacting with the virtual interface that has been created here.

![Oculus Rift](image1.png) ![HTC Vive](image2.png)

**Figure 2.1: Oculus Rift**  **Figure 2.2: HTC Vive**

## 2.2 User Interface

The additions to the *ParaView* plugin will add an overview of the current visualization pipeline similar to the one shown in the desktop environment (figure 2.3). The user will be able to select these pipeline stages with one of the controllers and will be presented with the properties of that pipeline stage similar to the properties in the desktop environment (figure 2.4). Then for each different type of property a 3D interface will be created. When the property is edited the visualization will be updated accordingly.
The visual representation of the user interface will be built using the geometric primitives available in VTK. Other options would be to use a shared OpenGL context to create the UI in OpenGL or use an OpenGL supported library or game engine to draw this interface. This would require not only familiarity with the rather large ParaView and VTK source code, but also the code of that library. Due to time constraints this option was not chosen.

2.3 Building on top of ParaView

The plugin that was created by the developers of ParaView defines two new classes, one that handles the UI elements presented in the desktop environment and one that connects ParaView to OpenVR. The latter class called vtkPVOpenVRHelper keeps track of state changes of the VR environment, the UI and event handling. This is the starting point to create new interactions.

To prevent bloating this class it has been decided that the pipeline interface will be created from a separate class named pvOpenVRPipelineBrowser. This class determines the structure of the pipeline currently defined by ParaView and creates a visual representation in the VR space.

The vtkPVOpenVRHelper also keeps track of a list of selectable actors, these are the actors that a user can interact with by pointing to them with their left controller. These are stages of the pipeline and the visual representations of the properties of these pipeline items. When the vtkPVOpenVRHelper class registers a click of the left controller trigger, the vtkPVOpenVRHelper will check if the pointer is intersecting with one of the selectable objects.

The various objects that are selectable with the left controller pointer are all of the same base class, this abstract class (pvOpenVRSelectable) requires derived classes to implement a selection function. This function is called when that item has been selected with the pointer and it defines the specific interaction of that class.
2.4 Properties in ParaView

There are several different types of properties a source or filter pipeline stage can have, which makes it hard to create a general user interface for VR. The desktop environment of ParaView does hide some properties that are available if the filters are directly coded with VTK. This thesis focuses only on the properties that are editable from the desktop environment. A partial overview of the possible properties are shown in figure 2.6, in bold are the two domains that have been implemented in this thesis.

![Property Hierarchy Diagram](image)

Figure 2.6: Partial property hierarchy in ParaView.

In order to create a general structure where all of these properties can easily be represented in a 3D VR environment, an object oriented approach is used. Property widgets all extend from a base class, and for each type of property and each domain of a property a derivation of that class can be created. Each derivation of the base class adds more specific functionality to the widget. This hierarchy would closely follow the property hierarchy as shown in figure 2.6.
CHAPTER 3

Implementation

This chapter describes the implementation of how the input is handled, how the various widgets have been constructed and how the visualization pipeline is updated.

3.1 Input handling

The OpenVR plugin for ParaView already has some inputs bound to actions. For the pipeline interactions only one extra button was required: the left trigger which is bound to the selection of our selectable objects. In the OpenVR plugin input events are handled by a class called vtkOpenVRInteractorStyle. This class extends from vtkInteractorStyle3D that defines methods of interaction with the visualization in a 3D environment such as panning, rotating and translating [10]. The vtkOpenVRInteractorStyle adds OpenVR specific implementations of these interaction methods.

The vtkOpenVRInteractorStyle is modeled after the command/observer design pattern [8]. This means that the interactor class subscribes to events coming from various sources such as the OpenVR controllers. It also defines a callback which is called when these events trigger. In the case of the vtkOpenVRInteractorStyle the callback listens for button presses and position updates of the controllers in 3D space.

![Event hierarchy diagram]

Figure 3.1: Event hierarchy.

The events the vtkOpenVRInteractorStyle receives are platform dependent. The developers of the OpenVR plugin have decided that new events need to be specified for the functionality of
the plugin. For instance, a button press event is transformed to a VTKIS_MENU event that triggers opening or closing the main menu. This way vtkPVOpenVRHelper does not define specific buttons to actions which makes it easier to create platform independent interactions by only creating a new interaction style. Following this design a new event called VTKIS_POINTER_SELECT has been created, which vtkPVOpenVRHelper listens to, to start the pointer selection process.

3.2 Creating the pipeline representation

In ParaView and VTK widgets are classes used for interaction with a complex operation [9]. A widget is usually a combination of a property, a visual representation and a way to interact with that property. For the pipeline stages and the accompanying properties we aim to create similar classes that wrap all of the desired behaviour.

Creating the VR representation of the pipeline starts with finding the internal representation of the pipeline. ParaView uses XML files to import classes from VTK [5], these XML files determine what properties are associated with that class and how to edit them. This XML representation is available through ParaView classes derived from QT classes such as pqPipelineSource. The UI in ParaView is created using QT5 which is a platform independent library for desktop user interfaces. We use these QT objects to build our VR representation.

The server manager in ParaView (pqServerManagerModel) is a singleton that keeps track of all visualization components in the current visualization [7]. Through the interfaces of this class we can iterate over the list of source and filter classes that are currently in the pipeline. It also defines which pipeline stage takes input from another stage and to which it outputs its data. For each of the pipeline stages a new vtkPVOpenVRPipelineNodeWidget object is created. This class is extended from the pvOpenVRSelectable class and it defines the interactions and the graphical representation of the pipeline stage in VR.

A list of the edges between the vtkPVOpenVRPipelineNodeWidgets is also created, which is used to draw the connections between the pipeline stages.

To prevent the pipeline widgets from overlapping a force directed graph drawing algorithm is used to reposition the nodes [2]. The algorithm assigns positive forces between nodes that have a connection, and a negative force between all nodes that do not. By applying these forces and then recalculating these forces for some number of iterations an equilibrium is reached where the nodes no longer apply forces to each other. If enough iterations are applied no nodes should overlap.

3.3 Creating the property widgets

When a pipeline stage has been selected with the pointer, the pipeline representation is hidden and the properties for that stage are displayed. To interact with the properties of filter or source objects a vtkSMProxy is used. The vtkSMProxy class is used by ParaView to handle setting and getting the properties of pipeline objects. It also provides an iterator that allows us to iterate over all properties of that pipeline object.
The proxy can have properties that are not relevant for the end-user, these properties are not visible in the desktop environment and are also skipped in the VR environment.

To determine the data type of the property we are dealing with, the property is cast to one of the type specific property classes. In this thesis two of these properties are implemented, the integer vector and double vector. These vectors, or n-dimensional arrays, can have varying applications that all have different representations. How these properties are used is determined by the domain, the domain defines how many dimensions the vector has and the possible value space for that vector. For example an integer vector property can have a \texttt{vtkSMBooleanDomain}, defining that property as a Boolean having one dimension and a value range of 0 and 1.

Once the type of property and the domain of that property have been determined, the corresponding widget for that property can be created. From the two domains that have been implemented here two widgets are created: a boolean widget and a double range widget. The former widget is represented by a box which is colored red when the boolean is false and green if it is true. The latter widget is represented by a slider, a sphere along a line whose location indicates the current value and the length of the line the value range.

3.4 Selection Mechanism

To create the graphics for the user interface some geometric primitives are used that are available in \textit{VTK} itself. This requires that for each primitive a source, mapper and actor need to be defined and added to the renderer.

The geometry of the widget classes is used to determine the bounding box of that widget. When the user tries to select either a pipeline stage or a property the location of the pointer is checked against the bounding boxes of all selectable actors. The selection event is sent to the first object that collides with the pointer.

When sending the selection event the coordinates of the pointer are also provided so that the widget knows where inside its bounding box the pointer was located at the time of selection. This is used by the double range slider widget to determine where on the slider range the selection was made.

3.5 Updating the visualization

When a property is edited, the visualization must be updated. If the properties of a pipeline stage have been updated, all pipeline stages after that stage need to be updated as well. \textit{ParaView}'s desktop environment marks all properties that have been edited as “dirty” and when the user applies these changes all objects with “dirty” flags are updated. All stages in the pipeline after the edited stages are also updated. This mechanism is replicated in the VR environment, but the
“accept” happens automatically after each edit. When the user edits a property, the property is marked as “dirty”, the object is updated, a new visualization is generated and becomes visible to the user.
Using the implementation described in the previous chapter it was possible to create a working VR user interface that can be used to edit the boolean and double range properties for any pipeline stage created in ParaView. The pipeline representation is shown in figure 4.2. When the user points to one of the boxes of a pipeline stage with the tip of the controller pointer (figure 4.1) and presses the trigger button the properties for that stage are shown (figure 4.3). The boxes of the boolean property and the pipeline stages have been made transparent to make it more clear when the tip of the pointer is inside the box or behind it.
4.1 Use case: contour visualization of a CT scan

The two widgets that have been implemented allow for some useful interactions, as an example we use a CT scan of a human head from the *Marching Cubes* paper by Lorensen and Cline [12]. The data set is a 3D map of the density of a whole human head. By applying a contour filter to the data source different sections of the head can be viewed, soft tissue such as skin or more dense material such as bones and teeth. The contour filter has an isosurface property which determines which densities are rendered [12].

Finding the right value for this property is vital to determine what is wrong with the patient. The double range slider widget allows the user to quickly find the value that shows the obstruction in the head of the patient.

To do the same in VR without the UI that has been implemented, the user would have to remove the VR headset, enter in a new value on the desktop GUI, apply the edit, put the VR headset back on and view the change in VR. Because this is a slow process, seeing changes in the visualization becomes hard due to change blindness. Change blindness is the phenomenon where one cannot see large changes that are normally easily noticeable. It is caused when there is a long interruption in a visual scene [15].

Figure 4.4: Skin
Figure 4.5: Skull
Figure 4.6: Obstruction
The goal of this thesis was to find a method of creating widgets to edit ParaView pipeline properties. The implementation has shown that this is possible by building on top of OpenVR plugin made by the developers of VTK and ParaView. The final product allows users to edit double range and boolean properties of any ParaView pipeline stage. More VR widgets can be created from the basis of this thesis. While a working implementation has been created, it is far from ideal and more work is to be done to make the interface more user friendly.

At first the design for the UI was a little more involved, showing both pipeline and properties at the same time. This turned out to be quite a hit on the frame rate, causing great discomfort. Adding a lot of simple geometry like cubes to the renderer seems to drastically hit the performance, this is weird as often the visualizations contain many more polygons than these boxes. More research needs to be done to determine why this performance hit is happening.

When editing complex visualizations it is possible that it takes a couple of seconds to generate the new visualization. This causes the rendering thread of the VR environment to hang for a couple of seconds. OpenVR is able to detect when ParaView is not responsive and it renders an overlay while it is unresponsive. This overlay hides the visualization while the generation is happening which causes change blindness between the new and old visualization. The solution to this problem would be to generate the new visualization on a different thread and then replacing the visualization once it is done generating.

VTK has some built in widgets that can be used to transform various types of data, one of these widgets is the `vtkSliderWidget` which has a 3D representation. Re-using this widget would have prevented re-implementing the slider for the double range property widget created here. The reason this was not done was because a VTK widget requires alterations to the interactor style and since the VR interactor style has not implemented all features a regular VTK widget requires it does not work. Making changes to the large VR interactor style class would have been the better solution, but due to time constraints the decision was made to simply create the interface directly on top of the current interactor style.

To determine the representation of the visualization pipeline the plugin makes use of some QT derived classes in ParaView such as `pqPipelineSource`, this does however make the plugin dependent on the desktop environment to create the VR environment. Being independent from QT would allow the VR environment to exist on its own and not be obstructed by the development of QT interfaces.
5.1 Future Work

Due to time constraints it was not possible to implement the class structure as outlined in the design chapter. The classes for the two widgets have a lot of duplicate code which should be refactored and moved to more abstract classes. Currently the widgets for the boolean and double range are derived from the same base class; the `pvOpenVRSelectable` class. If a new widget needs to be added that supports, for instance, integer ranges a lot of the functionality would have to be copied. It would be better to have a base class for ranges from which the double and integer range widget classes can be derived from. To do this a feature from C++ called templates can be used. This allows a class to be defined without specifying a specific data type. This can be used to create one generic widget class for the range domain that supports both an integer and double data types.

When above refactorings have been done, it should become much easier to implement new property types and domain widgets. Common property domains that can easily be implemented are the enumeration and integer range domains. These two domains do not differ greatly from the already implemented domains.

In the current implementation of the pipeline browser and property widgets are simply placed in front of the user on a flat horizontal plane facing up. User experience can be improved by placing the widgets on a surface closer to the user in a vertical orientation and by rotating the widgets to directly face the user.


Appendices
The implementation is hosted by the Gitlab of the Kitware developers at [https://gitlab.kitware.com/jan.o.schutte/paraview](https://gitlab.kitware.com/jan.o.schutte/paraview). This repository is a fork of the ParaView repository located at [https://gitlab.kitware.com/paraview/paraview](https://gitlab.kitware.com/paraview/paraview). All additions are made in a separate branch called openvr_controles.

To build the plugin follow the build instructions for ParaView and make sure the following cmake flags are set to on:

- PARAVIEW_USE_OPENVR
- PARAVIEW_BUILD_PLUGIN_OpenVR
- Module_vtkRenderingOpenVR