A visual programming environment for the Visualization Toolkit in Virtual Reality

Henk Dreuning

June 8, 2016
Abstract

Scientific visualization is used by researchers to get insight in correlations and important information that is present in big datasets. The use of virtual reality in combination with visualization further increases insight in complex, 3D structures. However, the creation of visualizations requires programming skills and is prone to errors. This thesis describes the development of an application that provides a virtual environment in which researchers can create a basic visualization pipeline. By guiding the user in the creation of a visualization pipeline, giving the options to manipulate selected parameters and allowing inspection of the resulting visualization in the same environment, it combines the advantages of a visual programming environment with the benefits of scientific visualization in combination with virtual reality. Several visualizations are built using the application, showing the possibilities that the environment provides. The application lays a foundation for a virtual environment in which elaborate visualization pipelines can be composed, simplifying the use of scientific visualization in research.
Contents

1 Introduction 7
   1.1 Related Work ................................................. 7
   1.2 Contribution ................................................. 8

2 Design 9
   2.1 External components ........................................ 9
      2.1.1 The Visualization Toolkit .............................. 9
      2.1.2 Oculus Rift ............................................ 10
      2.1.3 OculusVTK ............................................. 10
   2.2 Requirements ................................................. 11
   2.3 Exposing VTK’s API .......................................... 12
   2.4 Graphical user interface ..................................... 12
   2.5 Interaction .................................................. 13
   2.6 Program architecture ...................................... 14

3 Implementation 17
   3.1 Python wrapping ............................................. 17
   3.2 Introspection of VTK ......................................... 17
      3.2.1 The VTK class tree ..................................... 18
      3.2.2 During program execution ............................... 20
   3.3 The virtual environment ..................................... 20
   3.4 Rendering to the Oculus Rift ............................... 23
   3.5 Interaction .................................................. 23

4 Experiments 25
   4.1 Visualizations ............................................... 25
      4.1.1 Visualization of an arrow .............................. 25
      4.1.2 Visualization of a disk ................................. 26
      4.1.3 Marching man .......................................... 26
      4.1.4 Stream tracer ......................................... 27
   4.2 Performance ................................................ 30

5 Conclusion 33
   5.1 Future work ................................................. 34
      5.1.1 Method arguments ...................................... 34
      5.1.2 Interaction ............................................ 34
      5.1.3 Performance ........................................... 34
      5.1.4 Non-linear pipelines .................................. 34

Appendices 39
   A Button scheme ............................................. 41
   B Test system specifications ................................... 43
Visualization provides a means to get insight in correlations and meaningful phenomena that are present in big datasets, but are difficult to detect using existing algorithms. By visualizing the data, the human mind can be put to use for making connections and discovering important pieces of information that might otherwise have stayed undiscovered. Inspecting a visualization in a virtual environment can make scientific visualization more accessible and interactive. This, together with the immersion that virtual reality (VR) provides, further increases insight in complex, 3D structures.

Visualizing a dataset consists of several steps that transform data from one representation to another, until the desired end result is obtained. These steps can include reading input data from file, applying filters to this data and mapping it to a representation that can be rendered using existing 3D computer graphics techniques.

The Visualization Toolkit (VTK) is a widely used, open source toolkit for creating visualization pipelines [16, 17]. VTK consists of a collection of objects that can be used to compose a visualization pipeline. Since recently, a visualization created with VTK can be viewed in VR with the use of an extension to VTK called OculusVTK. However, OculusVTK does not enable the user to build a visualization pipeline from within the virtual environment, making it inconvenient to use. This thesis addresses this issue.

1.1 Related Work

For classic desktop use, several applications exist that provide a method for composing a visualization pipeline with VTK.

MayaVi is a data visualization software package, which uses VTK at its base [5, 8, 14]. It aims to integrate scientific Python libraries with TVTK, a “traits”-based wrapper for VTK. The functionality of MayaVi is presented both as a Python API and in the form of a graphical user interface. The main goal of MayaVi is to provide easy and interactive visualization of data.

MeVisLab is a cross-platform commercial software framework for image processing and visualization, combining modules from the MeVis Image Processing Library, the Insight Segmentation and Registration Toolkit (ITK) and the Visualization Toolkit (VTK) [1, 12, 15]. The framework can be used to build algorithms and applications, including graphical user interfaces with Qt, with the main focus lying on medical image processing.

The Delft Visualization and Image Processing Development Environment (DeVIDE) is a software framework that facilitates prototyping and experimentation with image processing and visualization algorithms [3, 11]. It incorporates multiple other libraries, including VTK, ITK, numpy
and Matplotlib into one, by making techniques of all frameworks available to the user as DeVIDE modules. These modules can be connected to each other and their variables can be changed at runtime, so the user is able to modify network characteristics and see the result immediately. DeVIDE provides a graphical representation of the visualization pipeline (or network) called the “Graph Editor”, in which the user can interact with the network.

Work has also been done to enable the use of VTK inside virtual environments.

Paul Rajlich created a VTK class named `vtkActorToPF` that translates a VTK actor (an object visible in a scene) into a node that the Iris Performer can use [10, 13]. The Iris Performer can then be used to render the VTK visualization for use in the cave automatic virtual environment (CAVE).

A module was added to VTK lately to add support for viewing visualizations on OpenVR compatible devices [9]. The module was tested on the HTC Vive virtual reality headset.

None of the mentioned solutions provide methods to compose a visualization from within a virtual environment, or viewing a visualization on the Oculus Rift virtual reality headset, which will be the targeted headset for this thesis.

1.2 Contribution

This thesis describes the design and implementation of a program that aims to combine the advantages of using virtual reality in combination with scientific visualization with the advantages of a visual programming environment for VTK. The ultimate goal of such a project would be to create a virtual environment in which the user can build and view a visualization pipeline and change the variables of the objects that are part of this pipeline.

The research question that follows from the description above is:

*How can an interactive virtual environment be created in which a visualization pipeline can be constructed and modified?*

The following chapter will discuss the design considerations for both the program’s components and its architecture. Chapter 3 will focus on the actual implementation of the environment and chapter 4 will explain the experiments that were performed together with their results. The last chapter will contain the conclusions drawn from the experiments.
2.1 External components

2.1.1 The Visualization Toolkit

The Visualization Toolkit (VTK) is a programming library written in C++, with wrappers for multiple (interpreted) languages like Python, Tcl and Java. It consists of a collection of objects that can be used to compose a visualization pipeline. Each object serves to either represent data in a particular form, to manipulate this data or to create a scene in which the visualization can be rendered. Additionally there are a number of supporting objects that can be used for convenience, outside of the actual visualization pipeline. The composition of a VTK pipeline, including the objects that make up the scene (actors, renderer and window), is shown in Figure 2.1.

![VTK Visualization Pipeline](https://www.cs.uic.edu/~jbell/CS526/Tutorial/Tutorial.html)

Figure 2.1: The composition of a VTK visualization pipeline, source: https://www.cs.uic.edu/~jbell/CS526/Tutorial/Tutorial.html.
The use of VTK in scientific research can pose a number of inconveniences. A researcher that wants to use VTK is obliged to have programming skills. If no programming knowledge is present, a fellow researcher will have to be called in to perform the implementation of the pipeline. A number of reviews and revisions are likely to be needed before the desired result is acquired. This can slow down the overall research progress.

The creation of a pipeline in VTK can be prone to errors. The programmer needs to instantiate the correct objects for its visualization, set the correct parameter values for the end result to be as expected and connect the appropriate input and output ports of various objects to each other. When making these connections, VTK’s class hierarchy needs to be taken into account: one particular VTK object might not produce the correct type of output data, while one of its subclasses does.

The aforementioned inconveniences in the use of VTK for scientific visualization could be solved by introducing a visual programming environment for VTK. By presenting the VTK objects, the building blocks of a pipeline, in a graphical way and presenting the options to change parameters and attributes of these objects in an intuitive way, the need for programming skills can be eliminated. This also gives the opportunity to guide the user in the decisions that need to be made, by only giving the options for connecting objects that result in a correct pipeline.

2.1.2 Oculus Rift

The targeted virtual reality headset is the Oculus Rift, of which up until now different versions have been released. In chronological order these versions are: Development Kit 1 (DK1), Development Kit 2 (DK2) and the Consumer Version 1 (CV1), shown in Figures 2.2a, 2.2b and 2.2c respectively. The CV1 will be the target version for this thesis. The Oculus Rift can be controlled with the Oculus Software Development Kit (SDK).

![Development Kit 1.](a) Development Kit 1. ![Development Kit 2.](b) Development Kit 2. ![Consumer Version 1.](c) Consumer Version 1.


2.1.3 OculusVTK

OculusVTK is a collection of extensions for VTK written in C++, originally developed by Casper van Leeuwen, visualization consultant at SURFsara. OculusVTK mainly consists of two parts. The main feature is the `vtkOculusRenderPass`, which allows a visualization created in VTK to be rendered into an Oculus Rift virtual reality headset. User interaction within the created virtual environment is added by the other main part of OculusVTK, the `vtkInteractorStyleGame`. This VTK extension enables user interaction in the virtual environment using either keyboard or controller input. A gamepad handler running on a separate thread is used to retrieve information provided by the controller. OculusVTK was originally developed for UNIX-like operating systems and the `vtkOculusRenderPass` was later ported to Microsoft Windows by Shabaz Sultan.
2.2 Requirements

The ultimate goal of a project like this would be to create a virtual environment in which the user can build and view any visualization pipeline and change the variables of the objects that are part of this pipeline. The result of any changes would be immediately visible inside the virtual environment, eliminating the need to repeatedly mount and dismount the VR headset. During the building process, the user would be guided in selecting appropriate building blocks, considering the current state of the pipeline. Interaction within the virtual environment should be easily accessible and feel intuitive. Although virtual reality headsets are generally regarded unsuitable for prolonged use, the method of interaction should be comfortable enough to allow usage as long as needed to build (parts of) a pipeline.

Creating a virtual environment as sketched above poses a number of challenges, ranging from exposing the VTK API to the user to UI design to user interaction. The goal for this thesis is to create an environment that serves as a base for the described environment, including aspects of all challenges involved. A user should be able to create a relatively simple, linear pipeline consisting of VTK objects, be guided in the creation by filtering objects that are to be added to the pipeline and be able to change most variables of basic type (integer, floating point). The user should be able to perform these tasks, together with the inspection of the resulting visualization, without leaving the virtual environment.

Summarizing, the functional requirements for the software are that the user must be able to perform the following tasks in the virtual environment:

1. Select VTK objects as building blocks for a visualization pipeline;
2. Build a linear visualization pipeline;
3. Inspect a graphical representation of the visualization pipeline;
4. Modify parameters of VTK objects that are of a basic type: integer or floating point;
5. Inspect the visualization resulting from the composed pipeline.

Also,

6. The VTK objects that a user can pick must be filtered to ones that accept the previous object’s output data as input data.
7. The input method must be comfortable for prolonged use.
8. The user should not have to leave the virtual environment during the creation of the pipeline or inspection of the resulting visualization.
9. The performance of the application while using the graphical user interface must be on a comfortable level.

The targeted operating system is Microsoft Windows and the application should be usable with the Oculus Rift Consumer Version 1 (CV1).

The stated requirements pose three main problems to be solved:

- How can the API of VTK be exposed to the user, while ensuring that the pipelines built with it are valid?
- How can a graphical user interface be created in the virtual environment, while simultaneously being able to show the visualization built by the user?
- How can user interaction be added to the virtual environment?

The following three sections explain the decisions made regarding these problems. The last section provides an overview of the application’s architecture.
2.3 Exposing VTK’s API

In order to expose the API of VTK to the user in a graphical manner, a method must be found that can automatically determine all classes in VTK. Some characteristics of these classes must be determined to know which classes are actually usable for the end user. There must be a means to determine which VTK objects can be connected to other VTK objects to ensure valid pipeline configurations. Finally, it must be able to find out which manipulations can be performed on VTK objects using its methods.

The introspection capabilities of interpreted languages like Python provide means to perform the desired operations. VTK has wrappers for Python, Tcl and Java. Python was chosen as the language to use.

2.4 Graphical user interface

A graphical user interface can generally speaking be implemented in several ways. For use with the application at hand, two requirements must be met: the method must work in combination with VTK and it must work in combination with the \texttt{vtkOculusRenderPass}. Also the difficulty of implementation is taken into account. Some options and the reasons why they are or are not suitable for the application are discussed.

For classic desktop use, GUI toolkits like Tk or Qt can be used to build a user interface. VTK integrates a number of these toolkits, so their use in combination with VTK poses no problems. They cannot be used in combination with the \texttt{vtkOculusRenderPass} however, as the render pass needs to integrate the GUI into the OpenGL context it uses to render to the Oculus Rift. GUI toolkits either make use of a graphics library that cannot be integrated with OpenGL, or do not allow other applications to use their graphics context. GUI toolkits would make the implementation of a user interface relatively easy.

Game engines can be used to create graphical user interfaces as well. Whether or not a game engine interferes with VTK is dependent on the internal workings of the engine. The render pass is not ready to be used in combination with a game engine. Further changes to the render pass would be necessary to facilitate this, if at all possible. The implementation of a GUI in a game engine would be on a relatively high level.

Another option is to create a UI in OpenGL. Because VTK also uses OpenGL to render the visualization that the user created, the two applications can change OpenGL states without each other’s notice. This can cause the applications to interfere. After slight modifications, the render pass should be able to display a GUI built with OpenGL. A disadvantage of this option would be that the implementation is on a relatively low level.

VTK itself can be used to create a user interface. Although several 2D and 3D widgets exist, they are not meant for use in a virtual environment. Using (simple) shapes like planes and cubes that VTK provides “source” objects for, a user interface more appropriate for VR can be built. This method poses no problems regarding the render pass, as it is created to render VTK visualizations. A disadvantage of this option is that it, as opposed to GUI toolkits and game engines, requires a relatively low level implementation.

Table 2.1 systematically shows the requirements that were and were not met by the considered methods. With VTK itself being the only reasonable option, this was chosen as the method to build the graphical user interface.
Table 2.1: Requirements that were met and implementational difficulty for all considered methods to build a graphical user interface.

<table>
<thead>
<tr>
<th></th>
<th>GUI toolkit</th>
<th>Game engine</th>
<th>OpenGL</th>
<th>VTK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not interfere with VTK</td>
<td>Yes</td>
<td>Depends on internal workings</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Works with vtkOculusRenderPass</td>
<td>No</td>
<td>No</td>
<td>After modification</td>
<td>Yes</td>
</tr>
<tr>
<td>Implementation difficulty</td>
<td>Easy</td>
<td>Easy</td>
<td>Hard</td>
<td>Hard</td>
</tr>
</tbody>
</table>

2.5 Interaction

Several options for the method of interaction can be considered. Their advantages and disadvantages are described here, leading to the decision that was made.

The most general and easy to implement method for interaction is traditional keyboard input. This would not require additional software or add implementation complexity on the level of the application itself, and input of various forms, like (floating point) numbers and text, is possible in the same way as for a normal desktop program. However, reaching the correct keys on the keyboard can be difficult with a VR headset mounted on one’s head, especially when the person in question is not able to “touch type”. This forms a major drawback for use in a VR environment.

Another option is the use of a gamepad (see Figure 2.3a), which is easy for the user to keep in its hands with a VR headset mounted. The analog sticks allow for easy movement through the virtual environment, and the gamepad is suitable for prolonged use. The integration of a gamepad into the application requires some programming logic on the application level and possibly the use of an external library, but the extra effort needed is still relatively small. The input of text and numbers is difficult using a gamepad.

The most advanced option that was considered is the Leap Motion Controller, a USB peripheral device that is mounted on the front of the VR headset, as shown in Figure 2.3b [4]. The controller registers the position and orientation of the user’s hands, and makes this information available to the developer via an API. This way, a method of interaction is possible that utilizes the 3-dimensional characteristics of the virtual environment. This can result in a highly intuitive method of interaction, given the similarities with interaction in the real world. However, the implementation is more complex than that of the other input methods considered, and the input of numbers and text, as well as movement though the environment can be difficult. Finally, the other options will provide better stability than this one.

(b) A Leap Motion Controller mounted on the front of an Oculus Rift CV1, source: http://store-us.leapmotion.com/products/universal-vr-mount-pre-order.

Figure 2.3: Two considered methods of interaction: a gamepad (Xbox controller) on the left and the Leap Motion Controller on the right.
Table 2.2 summarizes the advantages and disadvantages of each of the considered methods of interaction.

The general difficulty for use of a keyboard in VR and the implementation and stability remarks associated with the Leap Motion Controller caused these two methods to be regarded inappropriate for the application described in this thesis. The relative ease of integration and suitability for prolonged use of a gamepad positioned this method of interaction in favour of the others and was therefore chosen as the method of interaction to use.

<table>
<thead>
<tr>
<th></th>
<th>Keyboard</th>
<th>Gamepad</th>
<th>Leap Motion Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suitable for VR</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Suitable for prolonged use</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Easy to input numbers/text</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Implementational difficulty</td>
<td>Easy</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Stability</td>
<td>Good</td>
<td>Good</td>
<td>Bad</td>
</tr>
</tbody>
</table>

Table 2.2: Advantages and disadvantages of each of the considered methods of interaction.

2.6 Program architecture

The decisions made in the previous three sections result in an application that is built from several parts. An overview of the program’s architecture is given.

The application roughly consists of three layers: the backend, the viewers and the UI, as shown in Figure 2.4. The backend consists of several components that provide the methods used for the introspection of VTK and other information that is ultimately shown in the UI. These include classes that represent the user-built pipeline, the separate VTK objects in that pipeline, a file browser and information about buttons used in the UI.

The viewers form the layer between the backend and the UI. The UI, in combination with the user created visualization, is rendered using the `vtkOculusRenderPass`, which in turn relies on VTK and the Oculus SDK. User interaction is handled by the interactor, which gets input data from the gamepad via a dedicated handler.
Figure 2.4: Schematic overview of the application’s architecture.
This chapter discusses the implementation of the VR application. First some general information is provided, after which the introspection of VTK, the UI, the rendering to the Oculus Rift and the implementation of the method of interaction will be covered in their respective sections.

As mentioned before, the introspection capabilities of the Python programming language are used to determine the characteristics of VTK objects that are needed to, in the end, provide the user with the options that VTK offers in a graphical manner. Not only the introspection of VTK, but also the rest of the application was implemented in Python.

3.1 Python wrapping

Both the `vtkOculusRenderPass`, other components of OculusVTK (`gamepadHandler` and `vtkInteractorStyleGame`) and the Oculus SDK are written in C++. To use these components in the application Python wrappers have to be generated, for which several options exist.

- VTK uses its own wrapping system to generate Python wrappers for VTK objects. This wrapping system can also be used to wrap VTK objects that extend the VTK source code, like the `vtkOculusRenderPass`.
- A wrapper can be generated using a library such as Boost.Python [2].

To ensure compatibility with the rest of VTK, the VTK wrapping system was used to create wrappers for the `vtkOculusRenderPass`. The use of this system poses one inconvenience however: the initialization of the Oculus Rift is done outside of the render pass, while the Oculus SDK is also written in C++ and is therefore not accessible in Python. To resolve this, a method called `InitOVR` was added to the `vtkOculusRenderPass` object, which handles the initialization of the VR headset. Finally, the VTK wrapping system makes this method available in Python.

3.2 Introspection of VTK

The introspection of VTK consists of several parts. From a user perspective, two actions require introspection of VTK: adding an element to the pipeline and editing an existing pipeline element. When adding a new VTK object to the pipeline, the options should be limited to only those VTK objects that accept the output of the previous object in the pipeline as input. Also, the user should be able to filter the objects it can choose from by category. When editing a pipeline object, the editable parameters of that VTK object must be shown. Ultimately, this results in a number of introspection tasks that need to be performed:

- Select VTK objects with the correct number of input and/or output ports.
• Given the output of the previous VTK object in the pipeline, select the possible succeeding VTK objects.

• Categorize VTK objects.

• Retrieve, parse and use the methods of a VTK object to change parameters.

The approach used for each of these subjects will be discussed.

### 3.2.1 The VTK class tree

At the beginning of the execution of the application, before the virtual environment is created, a class tree is built for VTK. This class tree starts at `vtkAlgorithm` and recursively traverses all of its subclasses. For each VTK class, a `TreeObject` is created that represents the class. Several characteristics of the class are determined and stored in the wrapper object:

- Whether the VTK class is a **concrete** or an **abstract** class. VTK classes that are defined as being `abstract` in the C++ source code are not defined as such in the Python wrapper. Instead, the `__init__` method ("constructor") of the Python VTK object raises an exception when called. This means that no reflective method can be used to find out if a VTK class is `abstract` or not. Instead, an instance of the class is created inside a try-except block, with the catch of an exception indicating that the class is `abstract`.

- If the VTK class is `abstract`, store whether it is implemented, i.e. if one or more of its subclasses is `concrete`. Using the information gathered about which classes are `abstract` and which are `concrete` in combination with the recursive nature of building the class tree, this characteristic can be easily determined.

- If the class is `concrete`, parse its methods. Python’s built-in function `dir` is used to retrieve the list of methods of a VTK object. Due to the systematic naming of methods in VTK, each method’s name matches one of the patterns shown in Table 3.1.

<table>
<thead>
<tr>
<th>Method type</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set methods</td>
<td><code>Set&lt;property&gt;</code></td>
</tr>
<tr>
<td>SetTo methods</td>
<td><code>Set&lt;property&gt;To&lt;value&gt;</code></td>
</tr>
<tr>
<td>OnOff methods</td>
<td><code>&lt;property&gt;On/Off</code></td>
</tr>
<tr>
<td>Get methods</td>
<td><code>Get&lt;property&gt;</code></td>
</tr>
</tbody>
</table>

All `SetTo methods` that operate on the same property are stored together in one structure. The same is done for the `OnOff methods`. Finally, the `Get methods` are associated with the `Set`, `SetTo` or `OnOff` methods that operate on the same property.

In order to utilize the obtained methods to change parameters of a VTK object, it needs to be called. Therefore, the method’s parameter types need to be known. VTK’s Python wrapping system adds a “docstring” to each of a VTK object’s methods. This docstring contains one or more method signatures. Regular expressions and string manipulations are used to find these signatures and extract the descriptions of return- and parameter types out of them. Python’s built-in `eval` function is used to transform the string describing the types into a tuple of actual Python types.

Using the derived parameter and return types, the default value of a property that is affected by an object’s method is determined. This is shown to the user in the UI. When the user changes the property’s value via the UI, the method’s name and argument types are used to actually set the new value in the VTK object.
At this moment only parameters that are of integer or floating point type are made available to the user, as described in the next section. The described methodology to extract methods, their signatures and parameter- and return types is capable of detecting other types as well, like booleans, VTK class types and with additional manual mapping also strings and tuples. In order to use these types, support should be added to the UI and the application’s backend.

- Categorize the class. The categorization of a VTK class is based on its name. VTK uses the “camelcase” capitalization scheme for its classes, starting each word with a capital letter, except the first word. A list of common words is assembled by splitting the names of all classes on capital letters and counting the number of occurrences of the resulting words. The words with the most occurrences serve as the categories. Figure 3.1 shows an example of this methodology for a set of three VTK classes.

The categories only have to be determined once, before the first time the application is run, as they will not change as long as no other version of VTK is installed. The categories are thus stored in a file in JSON format, and read when the application is run. At this time, the categories have to be entered manually into the JSON file, based on the list of common words generated by a separate script.

To categorize a particular class, the words occurring in its name are checked to correspond to one of the predetermined categories. If so, the class is set to belong to that category. One VTK class can belong to multiple categories.

![VTK classes diagram](image.png)

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>2</td>
</tr>
<tr>
<td>Mapper</td>
<td>2</td>
</tr>
<tr>
<td>Volume</td>
<td>1</td>
</tr>
<tr>
<td>Unstructured</td>
<td>1</td>
</tr>
<tr>
<td>Poly</td>
<td>1</td>
</tr>
<tr>
<td>Data</td>
<td>1</td>
</tr>
<tr>
<td>Structured</td>
<td>1</td>
</tr>
<tr>
<td>Grid</td>
<td>1</td>
</tr>
<tr>
<td>Geometry</td>
<td>1</td>
</tr>
<tr>
<td>Filter</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>Classes in category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>vtkStructuredGridGeometryFilter</td>
</tr>
<tr>
<td></td>
<td>vtkUnstructuredGridVolumeMapper</td>
</tr>
<tr>
<td>Mapper</td>
<td>vtkPolyDataMapper</td>
</tr>
<tr>
<td></td>
<td>vtkUnstructuredGridVolumeMapper</td>
</tr>
</tbody>
</table>

Figure 3.1: An example of the process to determine VTK class categories, using only three VTK classes. The names of the classes are first split into words. The most common words (“Grid” and “Mapper”) form the categories.
3.2.2 During program execution

After the VTK class tree has been built, two actions can be performed that either make use of the information stored in the tree, or require additional introspection of VTK objects.

Adding an object to the pipeline. The user is presented with a list of concrete VTK classes from which a new pipeline object can be chosen. Next to that list the categories are shown, serving as filters for the VTK classes to choose from. The VTK classes that pass the category filter can not necessarily be appended to the pipeline. Therefore, these classes are tested to contain the correct number of input ports (zero if no items are currently in the pipeline, more otherwise) and to accept the output data of the previous object in the pipeline as their input data.

The output of one VTK object is set as the input of another object using output ports and input connections. Although the methods that are used to connect two VTK objects provide information about the types of their parameters and return value, this information is too general to determine if the two objects can be connected. To resolve this, each object that passes the category filters is tested to accept the output port of the previous object in the pipeline by actually performing the connect operation on dummy nodes of the same types. An error handler class is set up to observe VTK’s “ErrorEvent” and “WarningEvent”, which serve as an indication that the input port and output connection of two classes cannot be connected. The result is cached for future use.

Editing an existing object. When editing an existing VTK object in the pipeline, the methods that were parsed are used to show a list of editable parameters and their current values. After changing a parameter’s value, the corresponding “set” method is called to commit the change and the “get” method is used to confirm and store the new value.

3.3 The virtual environment

The information extracted from VTK is shown to the user in a graphical user interface at the appropriate time. This section will discuss the graphical way in which this information is presented.

The virtual environment consists of three visible items. A visualization of the current state of the pipeline, a UI plane which shows a graphical user interface and the visualization created by the user itself. An overview of the scene is shown in Figure 3.2.
The starting point for performing actions for the user is the visualization of the pipeline itself. Initially, a single block with a plus sign is shown, which the user selects to start the addition of a new pipeline object. The UI plane shows two lists, one with categories that can be turned on and off to make a selection of VTK classes of the categories of interest. The other list shows the VTK classes that result from the selection. VTK classes only show up in this list if they belong to each of the selected categories (so the category filter works as an ‘AND’ filter), if they have the correct number of input ports and if they accept the output from the previous object in the pipeline as input (if any). The user selects a VTK class from the list to instantiate it and append the VTK object to the pipeline. Both the visualization of the pipeline and the visualization resulting from it are updated accordingly.

With one or more objects in the pipeline, the user chooses to either add more objects to the pipeline using the block with a plus sign, or to edit an existing object. When an existing object is selected, the UI plane shows a list of methods that can be called to change the object’s parameters. Selection of an OnOff method or a value of a SetTo method toggles or sets the selected parameter or value. Selecting a SetValue method switches the UI shown on the UI plane to one of three UIs: the integer editing UI, floating point editing UI or the filebrowser UI for parameters of types integer, floating point and string respectively.

Figure 3.3 shows the UI for editing an integer value. The user can raise or lower the value by one with the “up” and “down” buttons on the gamepad. Two other buttons can be used to increase or decrease the step size. Increasing or decreasing the step size once multiplies or divides the current step by ten, respectively. This way the user can quickly enter both small (close to zero) and very large numbers. See appendix A for the exact button configuration.
The UI for editing a floating point number is shown in Figure 3.4. Two vertical bars are used to annotate the part of the number that is currently being edited (the “8” in Figure 3.4). This is either the integer part of the number, or one of the digits after the decimal point. The integer part of the floating point number can be edited in the same way as in the UI for editing an integer. For the other digits, the step size cannot be changed, because the small range of possible values (0 to 9) makes this unnecessary. When the marker is moved beyond the rightmost digit, a zero is appended automatically. This enables the user to input floating point numbers with arbitrary precision.

The floating point number that is edited is internally represented as a string, so that manipulation of the integer part and individual digits is possible. When the new value is acknowledged by the user, the string representation is transformed back into a true floating point value.

The filebrowser UI, shown in Figure 3.5, consists of a list of files and directories in the current active directory. The user can traverse directories and select a file or directory as input. The path to the chosen file or directory is used as a parameter of string type.
The UI plane and all UIs that can be shown on it are created using basic shapes that VTK provides source objects for. The UI plane itself and the buttons and boxes in the UIs are created from \texttt{vtkPlaneSource}s, in some cases with textures applied to them. Text is shown using VTK’s \texttt{vtkTextWidget}.

A separate class in the UI layer of the application, the \texttt{UIManager}, is responsible for changing the UIs that are visible on the UI plane. User interaction triggers events in the backend of the program, which are passed through to the \texttt{UIManager}, after which appropriate action is taken.

### 3.4 Rendering to the Oculus Rift

The virtual environment, including the visualization created by the user, is rendered to the Oculus Rift using the \texttt{vtkOculusRenderPass}. The version of the render pass that was written for Microsoft Windows uses the Oculus SDK version 0.8. This makes the render pass usable in combination with the Oculus Rift DK2, but not with the targeted CV1. Parts of the render pass are rewritten to conform to the API of Oculus SDK version 1.4, so that it can be used in combination with the targeted version of the Oculus Rift.

### 3.5 Interaction

The \texttt{gamepadHandler} bundled with \texttt{OculusVTK} was written to work with UNIX-like operating systems, using features specifically for that platform. Since the target operating system is now Microsoft Windows, this handler class is rewritten to work with the target platform.

The Simple DirectMedia Layer (SDL) version 2.0.4 is used to retrieve gamepad input in the form of events. Upon instantiation, the \texttt{gamepadHandler} class spawns a thread that continuously checks for controller input. The state of the gamepad is stored in a data structure and is retrieved and processed by \texttt{vtkInteractorStyleGame}’s \texttt{OnTimer} method, which in turn is called every time that the application’s renderer’s \texttt{Render} method is called. The button mapping scheme used by \texttt{vtkInteractorStyleGame} is changed to conform to that used by SDL.
4.1 Visualizations

Several pipelines were built using the application, in order to assess if the features of VTK that are exposed to the user can be successfully used and which operations normally performed while creating pipelines cannot currently be performed in the virtual environment. It also assesses if the stated requirements for the application are met and provides other insights into the use of the application.

4.1.1 Visualization of an arrow

The visualization of an arrow requires a simple, linear pipeline, that makes use of a VTK object generating source data and a mapper object. No tuning of parameters is needed. The part of the corresponding visualization pipeline that must be built inside the virtual environment is shown in Figure 4.1a. The arrow can be visualized without problems, by simply appending the required objects to the pipeline in the virtual environment. The resulting visualization is shown in Figure 4.1b.

![Figure 4.1: The visualization of an arrow.](image)

(a) The pipeline used to visualize the arrow.

(b) The resulting visualization, together with the pipeline and (currently empty) UI inside the virtual environment.

Figure 4.1: The visualization of an arrow.
4.1.2 Visualization of a disk

A disk is visualized, which requires a visualization pipeline similar to that of the arrow, but uses a different source object. Several parameters are changed, both of integer and floating point type: the inner and outer radius of the disk are set to 0.3 and 0.75 respectively, and to obtain a nice circular looking disk, the circumferential resolution of the *vtkDiskSource* is set to 100. The part of the visualization pipeline that must be built inside the virtual environment is shown in Figure 4.2a. The visualization pipeline for the disk can be built without problems. The integer and floating point values can be changed by using the interfaces designed for that purpose. The resulting visualization is shown in Figure 4.2b.

![Diagram showing the pipeline used to visualize a disk. Parameters of VTK objects that were changed are shown in rectangles under the object.](image1)

(a) The pipeline used to visualize a disk. Parameters of VTK objects that were changed are shown in rectangles under the object.

![Image showing the visualized disk, together with the pipeline and UI inside the virtual environment. The UI shows the outer radius of the *vtkDiskSource* object being edited.](image2)

(b) The visualized disk, together with the pipeline and UI inside the virtual environment. The UI shows the outer radius of the *vtkDiskSource* object being edited.

Figure 4.2: The visualization of a disk.

4.1.3 Marching man

In this experiment a CT scan of the head of a boy is visualized, using a DICOM dataset from the Midas Repository [7]. For this visualization, a DICOM reader is needed, for which the path to the folder containing the DICOM data must be set. The contour filter object is used to create an isosurface, followed by a mapper object. Scalar visibility is turned off in the mapper object, so no coloring is applied. The part of the visualization pipeline that must be built inside the virtual
environment is shown in Figure 4.3a. This visualization cannot be fulfilled through the virtual environment. Creating and appending the required objects poses no problems, nor do setting the directory name and scalar visibility parameters. However, the method that should be used to set the contour value for the \texttt{vtkContourFilter} object requires two parameters: a contour value (floating point) and an index for the contour to set to the value for (integer). Because this method requires two parameters, it is excluded from the graphical user interface. For illustrative purposes, a small extension to the application was made to append the extra parameter at the time the \texttt{SetValue} method is called. This allows the method to be used in the user interface as if it only requires a floating point value. The resulting visualization is shown in Figure 4.3b. Compared to the arrow and disk, the visualization of the head is large in scale. This resulted in two observations made during the inspection of the result: the UI and visualization of the pipeline are now relatively small, so alternately inspecting the result and making changes in the UI requires a lot of movement through the environment, and the size of the visualization makes the movement through the 3D world feel slow.

![Diagram](image1)
(a) The pipeline used for the visualization. Parameters of VTK objects that were changed are shown in rectangles under the object.

![Image](image2)
(b) The visualized DICOM dataset: a boy’s head. The UI can be seen in the right bottom corner, relatively small compared to the visualization of the head. A contour value of 500.0 was set via the graphical user interface.

Figure 4.3: The visualization of the CT scan of a boy’s head.

### 4.1.4 Stream tracer

In this experiment a stream tracer is used to create streamlines. The points of a \texttt{vtkPlaneSource} are used as starting points (seeds) for the streamlines. These points are provided to the stream
tracer by giving the output of the plane source as an argument to vtkStreamTracer’s SetSource-Connection method. The course that the lines take is determined by a vector field that is created from a PLOT3D data file from Midas [6]. The vtkStreamTracer uses these two pieces of information to create the streamlines after which a mapper completes the pipeline. The part of the visualization pipeline that must be built inside the virtual environment is shown in Figure 4.4a. It is not possible to compose this pipeline using the application. Creating and connecting the objects in the upper part of the pipeline as shown in Figure 4.4a poses no problems, but setting the source object for the stream tracer that is used to generate starting points is not possible. This would require the UI to both expose methods that require VTK objects (or their output) as parameters to the user and to provide a method to either create or select a VTK object to use as that parameter. For illustrative purposes, the visualization resulting from the stream tracer pipeline is shown in Figure 4.4b. This visualization was not built with the application, but directly in VTK.
(a) The pipeline used for the visualization. Parameters of VTK objects that were changed are shown in rectangles under the object. The \texttt{vtkPlaneSource}’s output should be given as an argument to \texttt{vtkStreamTracer}’s \texttt{SetSourceConnection} method.

(b) The streamlines generated by the stream tracer using a plane source for starting points and a PLOT3D data file to create the vector field. All streamlines start from the (invisible) plane on the lower left corner of the image. The maximum propagation, which influences the maximum length of the streamlines, was set to 200. This visualization is not created using the application.

Figure 4.4: The visualization of streamlines created with a stream tracer.
4.2 Performance

Previous chapters pointed out that some introspection and other VTK related actions are performed during the usage span of the application. Because rendering a visualization to the Oculus Rift CV1 on its own is already a performance intensive task, the effects of the additional VTK related actions on the performance of the application are measured.

The performance of the application is expressed in the number of frames that are rendered to the display in the Oculus Rift per second (FPS). The measurements were performed during the creation of the visualizations of an arrow, disk and a boy’s head, as explained in the previous section. Each visualization was created five times. The last element (\texttt{vtkPolyDataMapper}) was not added to the pipeline in order to be able to inspect the performance implications of using the UI and visualized pipeline, rather than that of rendering the created visualization. The value of the “scalar visibility” parameter in the visualization that uses a contour filter is left unchanged. The input was emulated and not performed by an actual gamepad, in order to keep input consistent across multiple measurements. The relevant specifications of the test system used for the experiments are listed in appendix B.

Figure 4.5 shows the distribution of the measured number of frames per second during the creation of the visualizations. During 45% of the elapsed time the framerate was approximately 58 FPS. During the rest of the time the framerate was kept between the limits of 54 to 61 FPS. A small increase is visible at 60 FPS compared to the values surrounding it.

In order to determine at which moments the framerate reaches its lower limit, Figure 4.6 shows the measured performance while creating the “marching man” pipeline. The shown framerates are the averages of 5 measurements. The colored dots on the graph mark the input received by the program. Looking at the graph’s progress, a number of peaks and drops stand out. Around the values of 18, 31 and 42 seconds and after 52 seconds on the horizontal axis, the framerate is approximately 60 FPS. At these times, no input actions are performed, leaving the application in an “idle” state, in which nothing other than the rendering of the image is done. These small pauses explain the jump at 60 FPS in Figure 4.5. Two drops stand out, the first being at 40 seconds, which can be explained by the contour value being set there, which causes VTK to use some processing power to create the contour. The second drop is at 47 seconds, which
corresponds to the time that the “mapper” category filter is turned on (marked by the activation keypress) and the VTK classes passing the filter are tested to fit onto the previous object in the pipeline (the \texttt{vtkContourFilter}).

![Performance during the creation of the "marching man" visualization pipeline](image)

Figure 4.6: Performance measurements while creating a visualization pipeline using a DICOM dataset and a contour filter. The dots on the lines indicate user input. All values are averages of 5 measurements.

Although small drops in framerate are visible at performance intensive moments, the overall framerate is not heavily affected by the actions related to the introspection of VTK performed in the background. The overall performance of the program will more heavily depend on the rendering of the visualization created by the user.
The goal for this project was to develop an application that serves as a foundation for a virtual environment in which a visualization pipeline can be created using VTK. The exposure of VTK’s API, the creation of a graphical user interface and the use of an appropriate method of interaction formed the three main problems that assemble to the goal. Several experiments were performed to assess if the stated goal and requirements are met.

The pipelines built during the experiments show that appending VTK objects to a linear pipeline, using SetTo, OnOff and selected (integer and floating point) SetValue methods to change VTK objects’ parameters work successfully. Visualization pipelines that only need this functionality to be built, can be successfully composed.

However, not all linear visualization pipelines can be built using the application. The reasons for that do not particularly depend on the type of pipeline that is built, so no concrete classification is possible for which pipelines can, and which cannot be composed. It merely depends on whether or not smaller operations that need to be performed are possible via the UI. The reason for not being able to modify a parameter is that the use of a method is necessary that is not available via the UI. The reason for this can be one of two (or both):

The method requires more than one argument. The UI currently only supports the use of methods that require one argument. In general, these methods influence a parameter of the VTK object they belong to, of which the meaning is apparent from the method’s name. Adding support for methods that require multiple arguments can be done in multiple ways, each introducing new challenges.

The task of setting the values for method arguments can be given to the user. This means a mechanism must be built into the UI to enable the user to manipulate multiple values, for each of the respective arguments. The meaning of each of the edited arguments must be made clear to the user in order for him to understand what value should be set. However, the docstring used to parse method names and argument types does not always provide an explanation or name of the arguments. Even if a name of or explanation about the arguments can be provided, the user would need rather extensive knowledge of VTK in order to use the arguments.

An alternative would be to move the task of filling in arguments that do not directly relate to a parameters value to the application. The automatic addition of the index of the contour value in one of the experiments can serve as an example of this. This means that the application must be extended with specific support for certain methods, either using predetermined values for arguments, or adding some mechanism to decide which arguments to use depending on the current state of the pipeline. Either way, this would require a lot of manual work on the application developer’s side.
The method requires an argument of an unsupported type. This type can be a VTK class or any other type, like a tuple or function. Support for non VTK related types requires an extension to the UI that gives the user the option to set a value for it. Arguments of a VTK class type require the creation or selection of an existing VTK object, or the output of one of its methods. This could be added to the application by letting the user create a VTK object at the time it wants to give it as an argument to a method, or by letting the user choose an existing VTK object.

A graphical visualization of the pipelines that can be composed can be inspected by the user, as well as the visualization resulting from the built pipeline. The user experience of this inspection could be further improved by the addition of the possibility to resize and move the UI plane and visualized pipeline, as well as a variable movement speed. The method of interaction proved suitable for the operations that needed to be performed and would provide enough options for the improvements just described. At no time there is the need to leave the virtual environment.

The performance experiments show that the introspection tasks performed in the background have only a limited effect on performance. The overall performance of the program will mostly depend on the processing power needed to render the visualization created by the user. As long as this is kept within comfortable limits, so will the overall framerate.

Although limited in its use for real world applications, the created program serves as a good basis for further development. Additions to the UI and backend can be easily made when making use of the design ideas currently present in the program architecture. Also, a foundation is laid for further use of the characteristics of VTK currently determined. For example, the classtree that was built can be helpful when enabling the use of VTK objects as arguments. Instead of the exact type specified by the documentation, the user may want to create an object from a subclass of this type to use as an argument. The classtree provides a means to list and inspect these subclasses.

5.1 Future work

5.1.1 Method arguments

As pointed out previously, support for multiple arguments can be added to the application, as well as support for other types of parameters than supported now, including VTK objects. The way this is presented to the user and the division of work between the user and the application are subjects for research.

5.1.2 Interaction

The usability of the user interface and visualization of the pipeline can be further improved, for example by making it possible to move or resize them. Movement through the virtual environment can be improved by introducing a variable movement speed. Options for adding interaction with the resulting visualization can be explored, possibly in combination with another input device.

5.1.3 Performance

Although the background activities involving VTK introspection did not pose a problem for the performance of the application, more processing work could be moved to the beginning of the execution of the program. Giving in on memory usage, the performance may be slightly improved.

5.1.4 Non-linear pipelines

Another option for future work is the addition of support for non-linear pipelines. Next to the need for changes in the backend of the application, a method for visualizing the pipeline will
have to be determined. The 3-dimensional characteristics of the virtual environment can be put to use even more when it comes to showing branches in the visualization pipeline created by the user.
Appendices
Figure A.1: The used button scheme on the Xbox 360 controller, source: https://en.wikipedia.org/wiki/Xbox_360_controller.
## Test system specifications

Table B.1: Relevant specifications of the test system.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i5-6600K, quad core, 3.5GHz (3.9 GHz turbo boost)</td>
</tr>
<tr>
<td>GPU</td>
<td>Asus strix GTX 970, 1,114GHz (1,253GHz boost), 4GB GDDR5 memory (7010 MHz)</td>
</tr>
<tr>
<td>RAM memory</td>
<td>8GB DDR4, 2133 MHz</td>
</tr>
<tr>
<td>Storage</td>
<td>Samsung 850 EVO 250GB SSD</td>
</tr>
<tr>
<td>VR headset</td>
<td>Oculus Rift CV1</td>
</tr>
<tr>
<td>Display</td>
<td>Full HD (1920x1080) LCD Display</td>
</tr>
<tr>
<td>Operating system</td>
<td>Microsoft Windows 10</td>
</tr>
</tbody>
</table>