Creating interactive visualization pipelines in Virtual Reality

Daan Kruis

June 9, 2017

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

Scientific visualization is the transformation of data into a visual representation, with the goal of obtaining new insights into the data. The Visualization Toolkit (VTK) is a large C++ library that has over 2000 classes used for various visualizations. Visualizing data in virtual reality enables researchers to study the data in even more detail. This thesis describes an application that can be used for creating a contour filter visualization pipeline while in a virtual reality environment. It allows the user to change the contour value and see the result inside the virtual reality environment. Some experiments were ran to determine the average frames per second as opposed to the number of triangles in the resulting visual representation of the data. The result of this thesis is an application that forms the basis of possible future research into a virtual environment in which different visualization pipelines can be created, edited and viewed.
## Contents

1 Introduction
   1.1 Related Work ......................................................... 7
   1.2 Research Question .................................................. 8

2 Design
   2.1 Hardware ..................................................................... 9
   2.2 Visualization Toolkit ................................................. 11
   2.3 Requirements .............................................................. 11
      2.3.1 Oculus Touch support ............................................ 13
      2.3.2 VTK introspection ................................................ 13
      2.3.3 Graphical user interface design ................................ 14

3 Implementation .................................................................. 15
   3.1 VTK introspection ....................................................... 15
   3.2 A graphical user interface in VTK ................................... 16
   3.3 Interaction with Oculus Touch controllers ...................... 17

4 Experiments ...................................................................... 21
   4.1 Experimenting goal ..................................................... 21
   4.2 Experimenting method .................................................. 21

5 Results ............................................................................. 25

6 Conclusions ...................................................................... 27
   6.1 Future Work .................................................................. 29
      6.1.1 Oculus Rift rendering in python ................................ 29
      6.1.2 Full Oculus touch controller support ....................... 29
      6.1.3 General widget interaction ...................................... 29
Scientific visualization is the process of transforming data into a visual representation. This representation can then be interpreted by humans to obtain new insights into the data which may otherwise not have been found. Scientific visualization is used heavily in many different sciences, like physics and medical science.

The Visualization Toolkit (VTK) is an extensive, object-oriented library containing thousands of classes used to create a wide variety of different visualizations. To create a visualization multiple steps have to be taken before the input data is transformed into the desired visualization. Together all these different steps (like reading the data, mapping it to a 3D model, applying textures etc.) form the visualization pipeline. The concept of the visualization pipeline was devised by Haber and McNabb. A visualization pipeline is a set of different general case operations that together result in the desired data visualization.

The reason to do scientific visualization in virtual reality is because doing scientific visualization in virtual reality as opposed to a regular 3D rendering window adds two new effects that allow for a more in depth inspection of the 3D object: stereo vision and motion parallax. Stereo vision is the extraction of three dimensional information by comparing the image data of a single scene viewed from two different vantage points. Motion Parallax is the effect that objects that are closer appear to move faster than objects that are further away, when using two viewpoints that move in the same way. These effects give the user a better perception of depth and distance to the object, which allows the user to obtain better insights into the visualized data than if a regular 3D rendering window was used.

The concept of scientific visualization in virtual reality is not new. There are many fields in which this is already applied, like in big data analysis and atmospheric research. However, virtual reality is not only used for scientific visualization of very specific topics, but also for more general purpose scientific visualization tools that use virtual reality because of the extra insights it provides. Virtual reality for use in scientific visualization is evidently an increasingly...
popular topic which makes it an interesting subject for this thesis.

Since May 2016 VTK has a new set of classes that allow for rendering visualizations in an Oculus Rift virtual reality headset [10]. These classes allow the user to view the result of the pipeline in virtual reality, but not much more. There is no interaction possible and no way to change the visualization pipeline from within the virtual reality environment. The goal for this project is to create a virtual reality environment where a full visualization pipeline can be created, edited and viewed. To allow for interaction in this environment the Oculus Touch controllers will be used.

1.1 Related Work

There are applications that present a regular graphical interface to build a visualization pipeline.

ParaView is an application developed by the same company as the Visualization Toolkit. It allows users to build a visualization pipeline in a graphical user interface (Figure 1.2). The main guiding requirements of ParaView are support for an efficient workflow and support for the visualization and analysis of large datasets [1]. Since September 2016 ParaView also supports rendering to an Oculus Rift or an HTC Vive [9]. However, this support only allows the user to view the visualization from one starting point. This means that, while it is technically possible to walk around an entire object and see every important detail, looking at large objects requires a lot of walking space for the user. There is no other way to rotate or move the object or the camera. Furthermore, ParaView does not enable the user to change the visualization pipeline and its parameters from within the virtual reality environment, which is one of the main focuses for this thesis.

![Figure 1.2: An example visualization in ParaView](http://www.paraview.org)

DeVIDE, or the Delft Visualization and Image processing Development Environment, was designed to speed-up the prototyping phase of research: to decrease the time used in the cycle of analyzing, designing, implementing and testing [3]. It provides many different functions, but
the most closely related function to this thesis is the graph editor it supplies (Figure 1.3). The graph editor provide a graphical representation of the current network of different visualizations and allows the user to connect the different parts of the visualization pipeline together. A similar setup is desired in this project, only it has to be usable in Virtual Reality.

![Figure 1.3: An example visualization in DeVIDE](https://graphics.tudelft.nl/devide-movie-and-image-gallery/)

1.2 Research Question

This thesis describes the implementation of a virtual reality application that allows its user to create, edit and view a visualization pipeline. The most important aspects are the graphical user interface, the interaction with this graphical user interface and the introspection into VTK that allows for creating and editing of a visualization pipeline. The leading research question in this thesis is:

How can a fully interactive virtual reality application be created that allows the user to construct and control a visualization pipeline from within a virtual environment?

The next chapter describes the different design requirements and choices that were made; Chapter 3 gives an insight into the implementation of the application; Chapter 4 and 5 talk about the experiments that were undertaken to measure the usability of the application and show these results and chapter 6 will discuss these results and draw conclusions based on this discussion.
CHAPTER 2

Design

In this chapter the design of the application is discussed. It starts with information about the different hardware that is used, then it talks about the Visualization Toolkit and finally it discusses the requirements for the application and the choices that were made.

2.1 Hardware

The virtual reality headset that is used in this project is the Oculus Rift, or more precisely the Oculus Rift Consumer Version 1 (CV1) (Figure 2.1). Besides the headset it has two sensors that track the movement of the headset and, if available, the Oculus Touch controllers.

![Figure 2.1: The Oculus Rift CV1 and one of its sensors.](https://www.ifixit.com/Teardown/Oculus+Rift+CV1+Teardown/60612)

To interact with the virtual environment the Oculus Touch controllers are used (Figure 2.2). Besides the regular controller elements like buttons, triggers and analog sticks, the Oculus Touch controllers offer position and orientation tracking, allowing the user to see their hands in virtual reality almost in the same way as they are in real life, provided the application they are using implements this.

2.2 Visualization Toolkit

The Visualization Toolkit (or VTK) is a library that contains over 2000 classes that can be used in a visualization pipeline. Some classes are meant as different stages in the visualization pipeline, while others are intended as additional classes to assist in the visualization process. The

\[ \text{Source: } \text{https://www.ifixit.com/Teardown/Oculus+Rift+CV1+Teardown/60612} \]
library is written in C++, but there are wrappers for multiple different languages like Python and Java.

The average VTK visualization pipeline consists of a data reader, then zero or more filters that transform the data, one or more mappers to turn the data into graphical objects, an actor for each of these objects which contains the relevant properties and finally a renderer and a render window to display the different objects (Figure 2.3).

---

**Figure 2.2:** The Oculus Touch controllers. The rings provide the position and orientation tracking.

**Figure 2.3:** The stages for an average VTK visualization pipeline.
2.3 Requirements

Using VTK is not a simple task. The user requires extensive knowledge of the library and needs to have at least rudimentary programming skills. Creating a visualization pipeline is a delicate process. The user has to know what the different parameters for each class mean and how to properly set these. Furthermore, the right output has to be connected to the right input to create a functional visualization pipeline.

All this together makes visualizing data using VTK difficult if you do not have the right knowledge, often leading to researchers requesting visualizations from their colleagues that are more experienced in the use of VTK. To combat this problem this thesis suggests a virtual environment that allows the user to visually construct a visualization pipeline and edit the different parameters. Furthermore, it should give the user recommendations based on the output of the previous stages, to remove the need for extensive knowledge of VTK.

All this should be doable from inside the virtual reality environment and any changes to the pipeline should immediately be processed, to allow the user to keep the virtual reality headset on. The interaction will be done using the Oculus Touch controllers, which as opposed to a regular mouse and keyboard setup, allows for three dimensional interaction. This three dimensional interaction can be used to enable the user to “pick up” the different building blocks forming a visualization pipeline and move and connect them using three dimensional movement.

Using virtual reality does bring along the problem that virtual reality headsets are not intended for prolonged use. Several health issues could arise during prolonged use, like motion sickness or disorientation [12]. So it is important to design the application in such a way that the use does not strain the user when using it for a longer time. The controls should be simple and should not require the user to move their hands and body haphazardly. Furthermore, it should be usable both while sitting and standing.

To achieve these requirements, several challenges arise. While VTK does support rendering to the Oculus Rift, there is no support whatsoever for the Oculus Touch controllers. This will have to be manually build into VTK. Furthermore, to create an arbitrary pipeline the different classes and their methods will have to be exposed to the application. And, finally, the graphical user interface will have to be designed in such a way that it works with VTK and the objects it produces.

2.3.1 Oculus Touch support

Every visualization pipeline ends with a renderer. The renderer is linked to a render window and this render window is then linked to a render window interactor. It is in this interactor that the main rendering loop takes place. In a regular VTK visualization the final function call is the call to start the event loop. This event loop handles certain input events and then calls the function that renders the visualization.

For this application it is desirable to add support for the Oculus Touch controllers that works with the concept of the event loop as it is currently present in VTK.

2.3.2 VTK introspection

Exposing the classes and their methods to the application is not as straightforward as it might seem. As mentioned, the library is written in C++, which is a compiled language, and because of that it is impossible to access information about the different classes and the underlying hierarchical structure at runtime. The way to solve this problem is to build the application in an interpreted language. Interpreted languages execute the lines of code as they are read, meaning that the underlying structure is not lost at runtime.
As was mentioned before, VTK offers python wrappings and since python is an interpreted language it can be used to achieve the desired introspection into VTK. However, not every class in VTK has python wrappings, due to python not being able to handle anything that contains pointers. Therefore, one requirement to the usage of python is the availability of the Oculus rendering classes in the python wrappers. These classes, however, are not wrapped in python. This poses a serious problem, because they are needed if we want to be able to use the application with the Oculus Rift in any way.

The reason as to why these classes are not wrapped is unclear. The suspicion is that it has something to do with the usage of pointers in these classes, because, as mentioned before, python, as opposed to C++, does not have these. A possible, though untested, solution could be to create wrapper classes around the Oculus rendering classes using C++ in which no pointers whatsoever are present and then wrap these classes in python and use them to render to an Oculus Rift from a python application. This is, however, beyond the scope of this thesis, though an interesting topic for further research.

2.3.3 Graphical user interface design

The graphical user interface is an integral part of the application and determines the user-friendliness of the product. There are several ways of creating a user interface. In related work performed by Dreuning, four of these options have been discussed and arguments have been provided for the use of each method. Important for each method is how well it can be used together with VTK and, more specifically, with the Oculus Rendering of VTK [5].

The first option would be to use one of the many GUI toolkits in existence. These provide an easy way to implement a GUI. The most obvious choice would be to use the Qt toolkit, because VTK has built-in support for this toolkit [8]. However, these toolkits cannot be used in combination with the Oculus rendering of VTK, because this requires the toolkit to be integrated into the OpenGL context that renders to the Oculus.

The second option would be to use a game engine, like Unity, to create a graphical user interface. However, since VTK is not designed to be used with game engines, it would depend on the game engine if it is at all possible to integrate it with VTK.

The third option would be to use OpenGL to create a graphical user interface. Since VTK uses OpenGL itself to render its visualizations the integration should pose no problems. However, the downside of using OpenGL to implement a graphical user interface is that is would be a very low-level implementation, making it a tedious process.

The final option would be to use VTK itself to create a graphical user interface. This has, obviously, flawless integration with VTK and the Oculus rendering of VTK. VTK offers a range of widgets, which could prove useful in the implementation of a graphical user interface. These widgets would have to be adapted to be used properly with the Oculus Touch controllers, but would make the implementation easier than OpenGL would make it. Furthermore, basic shapes like spheres and cubes exist which can be used to further create an interactive environment.

The first and second options would require too much alteration to be useful in the current application and the third option would take more time than the fourth option would. Therefore, the decision was made to implement the graphical user interface in VTK itself.
CHAPTER 3

Implementation

This Chapter discusses the implementation of the different aspects described in the previous chapter. It starts off with a discussion about the desired, but unavailable VTK introspection using python, then it explains the different parts of the Oculus Touch interaction and it ends with a description of how the graphical user interface was created using VTK.

3.1 VTK introspection

As mentioned in the previous chapter the Oculus rendering classes of VTK are not wrapped in python. This makes it impossible to have full introspection of VTK in the application and also render the application to the Oculus Rift.

Instead a simplification was made, with the idea of possible expansion should a newer version of VTK be able to properly wrap the Oculus rendering classes.

Instead of focusing on the entirety of VTK, just a single, though often used, scientific visualization technique was chosen: the contour filter. The contour filter produces isosurfaces, or isolines depending on the dimensionality of the input [11]. While this is a smaller project than was originally intended it still poses enough challenges in the implementation.

The visualization pipeline of a contour filter consists of five steps. The first step is the data reader. Depending on what type of file it has to read the application either uses a \texttt{vtkStructuredPointsReader}, or a \texttt{vtkXMLImageDataReader} object. The first reads files in the \texttt{.vtk} format and the second reads files in the \texttt{.vti} format.

The second step is the contour filter itself. Some parameters are set to a standard position: the computing of normals and scalars is turned on, while the computing of gradients is turned off. The initial contour value is set to 0, but this parameter can later be edited through the graphical user interface. The range that this parameter can take is based on the range of the input data coming from the data reader.

The triangles produced by the contour filter are then passed to a \texttt{vtkPolyDataMapper}, which maps the triangle information to actual geometric objects. The scalar range, which determines the coloring of the object is set from the lowest to the highest possible contour value, so that the color follows a rainbow spectrum where the lowest value is red and the highest value is blue.

The fourth step is an actor of the \texttt{vtkActor} type. The actor is what controls the various geometric properties, like position and orientation, of the three dimensional object generated by the mapper.

The pipeline ends with a renderer, which is a \texttt{vtkOculusRenderer} when rendering to an Oculus.
Rift is desired. The renderer uses the three dimensional data to actually draw the object to the screen, or screens in case of an Oculus Rift.

Together, these five steps result in a three dimensional object that is dependent on the supplied contour value.

3.2 A graphical user interface in VTK

As mentioned in the previous chapter the graphical user interface is created using existing VTK objects. There are five different things that need a representation in the graphical user interface. The graphical user interface should display the different stages of the pipeline; it should have a representation for the connection between two stages; it should have an interactable object to change the contour value; it should show the position of the users hands; and it should show the actual visualization.

To represent the different stages of the pipeline textured, outlined boxes, with dimensions of 0.25x0.25x0.05, are used. The texture is used to place the name of the stage on the box. Originally the intention was to use the \texttt{vtkTextActor3D} object to display the name, but this object didn’t scale properly. Later, the \texttt{vtkVectorText} object was discovered, and while this would help with the extensibility of the application, it was decided to keep using the textures. There are four boxes in total instead of five. This is because the rendering stage is not explicitly visualized, but instead implicitly connected as soon as the rest of the stages are properly connected. The boxes start stacked behind each other (Figure 3.2a) but eventually all end up at \(z = 0\) (Figure 3.2b). They can only move over \(x\) and \(y\).

![Figure 3.2: The different positions for the stages.](Image)

The connection between the different stages is visualized using an arrow. The arrow spans from the middle of the right side of the output stage to the middle of the left side of the input stage and it stays in this position if you move one of the connected stages (Figure 3.3). Arrows can only be created between stages that are allowed to connect to each other. Besides being a visual indicator that two stages are connected, the back-end of the stages is also only connected once the arrow is created (and removed if the arrow is removed). While this does not per se serve any goal in this particular application, since only one way of connecting the stages is possible, it is useful for extensibility purposes, should the possibility arise to create different visualization pipelines.
To change the contour value a \texttt{vtkSliderWidget} is used (Figure 3.3). This slider is attached to the contour filter box and starts out with a range from 0 to 1. As soon as the data reader stage is connected to the contour filter stage the slider range is updated to match the scalar range of the input data. The slider value starts, same as the contour filter it is attached to, at 0.

The interaction with all these object will be done using the Oculus Touch controllers. How this interaction was done will be discussed in the next section, but the position of the users hands will have to be visualized to give the user the required feedback to interact with the world. To do this two spheres are created that follow the position of the Oculus Touch controllers. While this is adequate for this application, the Oculus Touch controllers have support for different hand poses and orientations. If interaction would be desired that uses these poses, like pointing to an object for example, then it would be advised to upgrade the spheres to actual hand models that change their pose based on the pose of the users hands.

Finally, once every stage is connected properly the visualization is displayed (Figure 3.4). The object is first scaled so that the largest of the width and height is scaled to 1. Then the object is placed so that the center is at $z = 0$, the lowest $y$ position of the object is the same as the lowest $y$ position of all the boxes and the leftmost $x$ position of the object is 0.25 away from the rightmost $x$ position of all the boxes. When moving the slider, the contour filter, and therefore the visualization, is updated right away.

### 3.3 Interaction with Oculus Touch controllers

As mentioned, the Oculus Touch controllers are desired as the input devices for the interaction with the graphical user interface.

To add Oculus Touch support to VTK the status of the controllers has to be read and processed at a certain interval. To do this two possible approaches were considered:

The first approach is to create a separate thread on which the controllers are read and the input is processed all in a continuous loop. The results from this thread will then be passed on to the different objects that have been interacted with.

The second approach is to create a new event loop, similar to the event loop that was mentioned in the previous chapter, that will still call the render function at the end of each loop iteration, but will also read and process the input of the controllers.

The advantage of the first approach is that it does not impact the rendered frames per second,
but a disadvantage is that strange rendering glitches could occur, if the two loops are not syn-
chronized properly. This is because changing something to the scene in the input handling loop,
while the rendering loop is at the rendering step, might cause objects to be rendered improperly.
This can be solved by properly synchronizing the two loops, but then it would be almost the
same as using the second approach.

This second approach has the advantage that there will not be any unexpected rendering
 glitches, but the disadvantage is that, because all the input is handled in the event loop, the ren-
dered frames per second could decrease. However, while exploring this method it was concluded
that the impact on the frames per second was minimal and did not pose any inconvenience.
Therefore, the second approach was chosen.

Even though the spheres were mentioned as part of the graphical user interface, they are
technically not a part of it. At the start of the event loop two spheres are created at the current
position of the Oculus Touch controllers. Then, during each step of the loop the center of the
\texttt{vtk-SphereSource} for each of the spheres was changed to the position of the touch controllers relative
to the position of the camera, to allow the hands to move along with the movement of the camera.

The Oculus Touch controllers provide two types of input: the tracking state for the position
and orientation of the controllers; and the input state for the current state of the different
buttons. The loop maintains the these two states for the current and the previous frame, to be
able to see the change in the input state and the displacement of the controllers between the two
frames.

The input is processed in two ways. Some input is processed in the event loop itself, meaning
that it will always do the same, independent of which application uses it. And some input
generates events that can be caught by the application to do what the programmer desires. The
specific button lay-out for this application is shown in Figure 3.5.

Both rotation of the visualization and translation of the camera are handled in the event loop.
To rotate the visualization two vectors are determined: the vector from the previous position of
the right hand to the current position of the right hand and the vector from the center of the
visualization to the camera. The cross product is taken of these two vectors to determine the
vector which will be rotated about. The length of the right hand displacement vector is used as
a scaling factor to determine the rotation angle.

Rotation should only be available while the visualization is being shown, which is why the `vtkOculusRenderWindowInteractor` has a public `vtkActor` attribute in which the visualization actor will be stored if it is visible.

Translation of the camera uses the same displacement vector as the rotation did, only now it uses the left hand controller. This displacement is multiplied by a constant factor and added to the current translation of the camera. The resulting vector is then set as the new translation for the camera.

The rest of the interaction is handled via newly created events. VTK allows users to create their own events using the `vtkCommand::UserEvent` plus a certain integer. The most extensive interaction is required for the right hand trigger (the trigger pressed with the middle finger). This trigger is used for four different events. It invokes the `vtkCommand::UserEvent + 1` when the trigger is first pressed, `+ 2` while the trigger is being held down and `+ 3` when the trigger is being released. Furthermore, it invokes `vtkCommand::UserEvent + 4` if the invocation of `+ 1` does not set its abort flag to 1. This last event is necessary to distinguish between grabbing the slider widget and one of the boxes.

When the trigger is pressed the slider widget’s SelectAction function is called. The original hope was that it would be possible to use VTK’s built-in picker methods to facilitate interaction between the Oculus Touch controllers and the widget, but these proved to be inadequate when used with three dimensional input data. Instead the interaction was built manually for the slider widget. The widget first checks if the right hand sphere’s center is in its bounding box. If this is not the case the function returns and the abort flag will be 0, which means the `+ 4` event will be called. If it is in the bounding box the widget state is set to `sliding` and the new position for the slider is determined, so that the x coordinate of the center of the slider will be at the x coordinate of the center of the right hand sphere. To indicate that the slider is selected it is highlighted. Finally the abort flag is set to 1 and the `vtkCommand::InteractionEvent` is invoked. This event is caught in the application itself to change the contour value of the contour filter to the value that the slider widget now shows.

For the `+ 2` event, which is when the trigger is held down, a very similar process happens. It first checks if the widget state is set to `sliding`. If this is the case it once again aligns the x coordinates of the slider and the right hand sphere. It ends with invoking the same `Interaction-Event`.

When the `+ 3` event occurs, which is when the trigger is released, the widget state is set back to `start` and the widget’s highlighting is stopped.

---

If the widget is not selected when the right hand trigger is pressed the + 4 event will be invoked, which is caught in the application itself. In the application it checks if the right hand sphere’s center is in one of the boxes. If this is the case the x and y coordinates of the center of the box are set to match the x and y coordinates of the right hand sphere’s center. An integer is used to maintain which box has been selected and to determine if the + 2 (move) and + 3 (end of move) event have to be handled. These events are handled almost the same as with the slider widget, with the addition of the y component. When moving the boxes, possible arrows that are connected to the box are moved as well, so that their position relative to the box stays the same.

The A, B and X buttons both only invoke an event when they are first pressed. The A button invokes the + 5 event which is used to create arrows. It first checks if the right hand sphere is in one of the bounding boxes that could be used as output, which is every box except the one for the actor. If this is the case the start of the arrow is set to the right most side of the box and the end point follows the right hand until A is pressed again (Figure 3.6). If the right hand sphere is not in one of the boxes or in the wrong box the arrow disappears and some haptic feedback is given to the user. If it is in the right box the end of the arrow is set to the left most side of the box it is connecting to and the visualization pipeline in the back-end is updated to reflect this new connection.

![Figure 3.6: Arrow following the right hand sphere.](image)

The B button invokes the + 6 event which is used to remove arrows. If an arrow is being created it stops this process. If no arrow is currently being created it checks if the right hand sphere is in the bounding boxes of one of the arrows. If this is the case this arrow is removed and the visualization pipeline updated accordingly.

Lastly, the X button invokes the + 7 event, which is used to start the performance test. How this is done will be discussed in the next chapter.
This chapter discusses the experiments that were done on the application. It starts off by describing the idea behind the experiments and then it discusses the way the experiments were done and the different techniques that were used.

4.1 Experimenting goal

As with most graphical applications, one of the most important metrics is the number of frames per second, or FPS. According to the Oculus Best Practices guide the minimum required FPS for a pleasant user experience is 75 \[16\].

The number of frames per second is in large part dependent on the number of triangles in the rendering environment. The experiment will be to measure the FPS against the number of triangles in the result of the visualization pipeline. Note that this excludes the triangles in the rest of the graphical user interface. This is, however, not a problem, because these triangles will have no significant impact compared to the number of triangles in the result of the visualization pipeline. Using this information the number of triangles at which the average FPS is equal to the minimum required FPS of 75 will be determined. Two experiments will be done for this. The first will use large step sizes to approximate the the area in which the average FPS is 75 and the second will take smaller steps in this area to better determine the actual number of triangles for which the average FPS is 75.

4.2 Experimenting method

The experiments were done by first reading a large dataset, pertaining to a micro CT scan of coral, containing more than 100 million triangles, when created at contour value 1 (Figure 4.1). This posed some unforeseen problems, because initially both the application and VTK (and some additional required libraries) were compiled as 32-bit programs and libraries. This was not a conscious choice, but rather the default setting. Until this moment this had not mattered, but when trying to read in the large dataset it could not generate large enough memory addresses. Therefore, both VTK and the application had to be rebuild into a 64-bit library and 64-bit program respectively. This solved the problem of reading the dataset.

It was decided to measure the FPS 500 times per different number of triangles. So after 500 frames the triangles had to be reduced to start the next FPS measurement. The original idea was to use a decimation filter, which merges and removes triangles until an approximation of a certain fraction of the original triangles remains, while trying to stay close to the original shape of the object. This did, however, not work, because the application would use to much memory, eventually causing it to crash before it finished the decimation.
So instead of using a decimation filter to remove triangles, a specified number of triangles was randomly selected from the dataset and removed. This of course does not maintain the original shape of the object, but since the only interest is in the FPS against the number of triangles this does not matter. The advantages of using this method are that it is much faster, requires much less memory and that it allows for removal of a specific number of triangles, instead of an approximation of a fraction of the triangles.

Using this method, the triangles, that were the result of applying the contour filter with contour value 1 on the dataset, were first reduced to exactly 100 million triangles for the first experiment, after which the application was started. The visualization pipeline first had to be created in the virtual environment. This was done so that the application did not have to be altered too heavily to facilitate the experiment.

Once the pipeline is created the $X$ button was pressed to start the experiment. This button causes the camera to be translated to a position where the whole result of the visualization pipeline is in the center of the view and the visualization pipeline itself is not visible. Furthermore, it changes a global Boolean indicating that the experiment is running to true.

VTK invokes the `vtkCommand::EndEvent` after the rendering of a frame completes. The `GetLastRenderTimeInSecond()` function in the `vtkRenderer` class can then be used to determine the time in seconds it took the application to render the last frame, which has an inversely proportional relationship with the FPS. The intention was to add a callback function to the aforementioned event and to use this function to determine the FPS for that frame. The function, however, does not appear to work properly, as it returns values suggesting an FPS of about 1000. This is way too high and can be determined as false by simply looking at the rendering of the scene.

To combat this problem, instead of using the renderer’s `EndEvent` an addition was made to the event loop in the `vtkOculusRenderWindowInteractor`. At the end of the loop the `Render()` function, which renders one frame, is called. By reading the time before and after this call and subtracting these, the time in seconds can be determined and with that the FPS. However, the standard C++ timers or the timer in milliseconds from windows are not precise enough and still returned the wrong results. So instead the high resolution performance counter based in the processor had to be used. By querying this performance counter before and after the `Render()` call and dividing the difference by the performance counters frequency the amount of seconds required to render the last frame is measured with a desired precision. Dividing 1.0 by this result calculates the actual FPS, which is then stored in a public variable in the renderwindow interactor class. Lastly the `vtkCommand::UserEvent + 8` is invoked to signal that a new FPS value has been calculated.
By adding a callback object that observes this event to the renderwindow interactor object in the application a function will be called after a new FPS value has been stored. The callback object contains a counter that starts from 1, and an array of 500 doubles. The function that is executed each time the + 8 event is invoked checks if the global Boolean that indicates whether the experiment is running is true. If it is it will read the FPS value from the renderwindow interactor object and add it to the array based on the current counter. This happens 500 times, after which the contents of the array are written to a .csv file and the number of triangles is reduces by a specified amount. For the first experiment the amount of triangles to be removed was 10 million and this was done for 10 steps, which means that the first experiment started at 100 million triangles and ended at 0 triangles. The FPS was counted for all eleven triangle amounts. From the resulting data, the minimum, maximum, average and standard deviation was determined and plotted in a line plot. The starting point and step size of the second experiment was based on the results from the first experiment. During the experiments the Oculus Rift was placed on a stand to ensure that there was no disruption of the experiments due to head movement.

Table 4.1 shows the specifications of the machine on which the experiments were run.

<table>
<thead>
<tr>
<th>Component</th>
<th>Used System</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7-5930K - 6 cores - 3.50 GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>MSI NVIDIA GTX Titan Black - 6GB GDDR5 memory - 2.880 cores - 889MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB - DDR4 - 2800 MHz</td>
</tr>
<tr>
<td>HDD</td>
<td>WD Black 2.0TB - 7200 rpm - 164 Mb/s Read &amp; Write</td>
</tr>
<tr>
<td>OS</td>
<td>Windows 10 Pro</td>
</tr>
</tbody>
</table>

Table 4.1: The specifications of the machine that ran the experiments.
This chapter shows the results of the two experiments that were done.

As can be seen in figure 5.1, the transition from FPS values above 75 to FPS values below 75 happens between 0 and 10 million triangles. The second experiment will therefore be run from 10 million triangles to 0 triangles, reducing the amount of triangles by 500000 each step. This means 20 steps will have to be taken, which will result in 21 measuring points (Figure 5.2).
Figure 5.2: The results of the second experiment. The error bars show the standard deviation.
CHAPTER 6

Conclusions

The goal of this project was to create an application that allowed its user to create, edit and view a visualization pipeline all from the same virtual reality environment. To achieve this there were three obstacle to overcome: the various classes of VTK and their hierarchical structure had to be introspectable from within the application; A graphical user interface had to be created that allows for the creation and alteration of visualization pipelines using VTK objects; and support had to be integrated into VTK for the Oculus Touch controllers.

The first of these three obstacles proved to be too big, because while programming the application using the python wrappers from VTK would allow it to have the desirable introspection, it did not have the any wrappers for the classes that were required for rendering to an Oculus Rift. Therefore, it would either be an application in which everything in VTK could be used and every possible pipeline created, but on a normal screen, or the application would be limited to certain classes of VTK, but it would be a virtual reality application.

The second option has been executed and while this means that the application is not as extensive as was originally desired, it still serves as a nice starting point for future research. The graphical user interface works as intended and allows the user to create and edit the visualization pipeline. This is achieved by the new support that has been build into VTK for the Oculus Touch controllers. This thesis therefore partially answers its research question, because it does demonstrate a way in which a virtual reality application can be build in which visualization pipelines can be created, edited and viewed, but it does not describe an application which can perform arbitrary visualizations.

Aside from the fact that the application lacks introspection into VTK, there are some other features that are not present. Right now the only editing possible is a single contour value. It is, however, possible for a contour filter to have multiple contour values, allowing for more than one surface to be calculated and rendered. Furthermore, there are some other parameters that are static at the moment, but should rather be dynamic, like the input file’s name for instance. Right now this is passed as a command line argument, but it would be better if a file browser could be build into the graphical user interface that allows the user to change the file used as input from within the virtual reality environment.

The Oculus Touch controller support is sufficient to use the entire application, but could certainly be improved upon. Right now the user can only translate the camera, but has to change its yaw, pitch and roll by actual moving their head (or more precise the headset on their head). This interaction method could be build into the current setup of the Oculus Touch support without too much effort. Furthermore, the translation of the camera happens with hand movements, but this does not allow the user to “fly” through the virtual environment by specifying a direction and a speed, for example by using one of the analog sticks. This is also an improvement for which the groundwork exists, but that has not been specifically implemented.
In general only the controls that were required for this application have been implemented into VTK with events. However, should general support for the Oculus Touch controllers be desired events would have to be added for every possible action for each button or gesture. It would be best if these were given proper events, instead of \texttt{vtkCommand::UserEvent + n} events.

The results of the first experiment are as expected. The FPS gradually decreases as the number of triangles increases. This experiment places the 75 FPS point somewhere between 0 and 10 million triangles, supposedly around 5 million triangles.

Therefore, the second experiment zoomed in on this area to better determine the number of triangles for which the average FPS is 75. This experiment, however, revealed some interesting behaviour. In the interval from 7.5 million to 10 million triangles the FPS is fairly steady around 45. Then from 5 million to 7.5 million triangles the average FPS increases from 45 to 90, but with a very large standard deviation. When looking at the individual frames for these points the reason for this standard deviation becomes clear. The FPS constantly alternates between two values. For example, for 7 million triangles these two values are \( \sim 50 \) and \( \sim 80 \) FPS. The reason for this behaviour is unclear. There was no explanation found in any of the documentation that would explain this behaviour. The suspicion is that it has to do with one of two things.

The first possible explanation is that one of the FPS values is for the right eye and the other FPS value is for the left eye. This would imply that the eye with the higher FPS value could make better use of caching or another, similar, performance increasing method. This would, however, probably be very apparent to the user wearing the Oculus Rift, making it an undesirable effect, which makes this explanation less likely.

The second possible explanation has to do with the two other parts of the graph. As was mentioned from 7.5 million to 10 million the FPS is relatively stable around 45. From 0 to 5 million the FPS is relatively stable around 90. This suggests that the FPS is limited to 90 by either VTK or by the Oculus SDK. The alternating FPS values from 5 million triangles to 7 million triangles seem to lean towards these values also, which could imply that there is a built-in mechanism that caps the FPS at 45 until at least half of the frames can reach close to 90 FPS. However, the reason why this would be desirable over a gradually increasing FPS is unclear.

These two explanation are speculations at best. There is no way to prove or disprove them, without diving into the source code of VTK and the Oculus SDK. It is definitely possible that the actual explanation has nothing to do with either of the aforementioned explanations.

Determining the number of triangles at which the average FPS exceeds 75 is not as straightforward as expected. Technically the answer lies at about 6 million triangles. However, this experiment was done in relationship to the user experience. So even though the average FPS exceeds 75 at 6 million triangles, it isn’t until 5 million triangles that the FPS stabilizes around 90 FPS. Therefore, when considering the user experience, the triangle limit should probably be placed at 5 million triangles.

While this application has limited practical uses it still serves as a good foundation upon which to build an actual application that gives the user access to the entirety of VTK. The graphical user interface has been build in a way that allows for relatively easy extensions into multiple classes and a varying number of stages. The interaction with these classes can be made independent on how many there are and how they can connect with each other.

All together this application provides a basis upon which further research can be based to eventually realize the desire of a virtual reality environment in which every possible visualization pipeline can be created, edited and viewed.
6.1 Future Work

6.1.1 Oculus Rift rendering in python

As mentioned, the biggest obstacle during the execution of this project was the unavailability of the appropriate python wrappers for the Oculus rendering classes. An interesting topic for future research would be to find out exactly why these classes are not wrapped in python and to circumvent these issues to enable developers to build an application that allows for VTK introspection and Oculus Rift rendering.

6.1.2 Full Oculus touch controller support

It was already mentioned that the current Oculus Touch support is aimed towards this project. An useful subject for development would be to generalize this interaction to a standard approach that allows the users of VTK to write their own interpretation of each input. This would mean removing the camera translation and object rotation from the \texttt{vtkOculusRenderWindowInteractor} class and instead invoking events for those particular situations as well, allowing the user to decide how to handle camera movement and object interaction.

6.1.3 General widget interaction

Another interesting topic for future development, related to the previous subsection, would be the designing of a function that uses the Oculus Touch interaction to interact with the various widgets available in VTK. Right now only the slider widget can be used with the Oculus Touch controllers, but there are many more widgets which can be very useful in virtual reality visualization applications. Having a general interaction method that can be used in arbitrary widgets would allow the application programmers to use them more freely and allow the VTK developers to create new widgets without having to worry about the various interaction possibilities.


