INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Visualizing Big Data in Virtual Reality – Interactive Analysis of Large Scale Turbulence Simulations

Matthias Christopher Albert

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Visualizing Big Data in Virtual Reality – Interactive Analysis of Large Scale Turbulence Simulations

Matthias Christopher Albert

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Rubén Jesús García-Hernández
	Dr. Markus Rampp (Max-Planck-Gesellschaft)

Abgabetermin: 28. September 2018

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 28. September 2018

. . . .
(Unterschrift des Kandidaten)

Abstract

Direct numerical simulations of turbulent fluid simulations produce time dependent scalar fields on grids of at least 1024^3 points, along with huge amounts of trajectories of tracer particles which are advected with the fluid flow. Being able to explore the produced volumetric data in a three dimensional visualization might be a huge benefit over conventional two dimensional projections. Therefore a way to interactively explore these peta-scale data sets in virtual reality seemed to be a very appealing option for scientists researching turbulent fluid dynamics.

The goal of this work is to visualize a specific large scale turbulent fluid simulation in virtual reality. In order to do so, an already existing virtual reality application for the HTC Vive was extended. To ensure the applicability of the work flow described in this thesis and of the extended application itself, data of a real turbulent fluid simulation are used.

Due to the large size of the extracted isosurface graphic files, finding a way to visualize them, without reducing them to a quality so low that they are no longer fit for scientific visualization is the main challenge of this thesis. To accommodate for their size a multi threaded loading approach is used, where a second asynchronous thread loads new graphic files when needed. The thread communication is realized via the shared context functionality of OpenGL.

The second aim is to provide users with new controller functionalities, displayed above the controller, which enable them to explore the data efficiently and to adjust the amount of information shown without restarting the application. To identify which kind of functions are useful, a scientist tested an early version of the application and specified which functionalities were missing or needed to be changed.

The resulting application, based on NOMAD VR, provides the means to visualize large scale turbulent fluid simulations. Trajectories and various vectors of tracer particles are automatically calculated and visualized by the application, different controller functionalities allow to rescale the visualization, translate isosurfaces and to select single particles. The implemented asynchronous multi threaded loading approach works theoretically as long as the size of the few isosurface graphic files, which are loaded by the second thread, do not exceed one quarter of the graphic card memory.

Contents

1 Introduction

The goal of this thesis is to visualize a large scale turbulent fluid simulation, by using virtual reality technology. The interactive exploration of peta-scale data sets is very appealing to fluid dynamics scientists. Having volumetric data and being able to explore these in a three dimensional visualization is a huge benefit over two dimensional projections.

A two dimensional visualization lacks the perception of depth and as it is usually rendered with a fixed camera position it greatly restricts what the user is able to see. The application created for this thesis gives scientists the ability to freely move in a three dimensional, interactive visualization, without long waiting periods of rendering, which might lead to new insights [BM07]. For this purpose an existing scientific visualization tool, NOMAD VR, for molecular sciences was extended.

1.1 Motivation and Previous Work

Visualization is an important aspect of data science in general and plays a major part in the analysis and research of fluid simulation data. Turbulent fluid simulations are three dimensional direct numerical simulations producing time dependent scalar fields on grids of at least 1024^3 points, along with huge amounts of, up to 10^8, tracer particles which are advected with the fluid flow. The strength of the turbulence is represented by the scalar vorticity field and visualized as translucent isosurfaces [Aya16, Chapter 5.11.2].

Commonly used tools for these visualizations are open-source, multi-platform data analysis and visualization applications like ParaView [AJG+05] and VisIt [CBW+12]. In a previous project two dimensional videos of a turbulent fluid visualization were created with VisIt. The time dependent scalar vorticity field of the simulation data was visualized with a volume plot. This plot allows to map scalar values to customizable opacity and colour values. The strength of each vortex tube is indicated by its colouring and ranges from light blue to red. One of the tracer particles was chosen to be followed and therefore its trajectory along with various vectors were visualized. The first video used a conventional, static camera perspective giving an overview over the whole simulation. This camera perspective was also used for an overview map in second video. An example of this visualization is shown in figure 1.1 (top). A limitation of a conventional visualization is that it uses a fixed camera position placed outside of the visualized data, therefore details of the spatial correlations of the particle and the vortex tubes cannot be seen clearly.

In order to allow scientists to explore the data further, new ways of navigating the simulation volume were developed by co-moving the observer camera with a representative tracer particle inside the fluid. The user's viewpoint is set inside the simulation, following a chosen tracer particle's trajectory. To attenuate the resulting 'roller coaster' experience to a tolerable level for the observer and to avoid motion sickness a linear smoothing algorithm is used. This approach allows a closer observation of a particle's behaviour inside the fluid, especially when its close to strong turbulences. An example of this visualization is shown in figure 1.1

(bottom). However, this new approach also presents new constraints. The co-moving camera shows only part of the simulation and the selection of an interesting particle turned out to be clumsy and inefficient, as it had to be chosen by hand. An interesting particle was defined by having a sufficiently twisted trajectory and staying inside the visualized volume. After finding interesting particles a time consuming procedure was necessary before the final visualization of the dataset:

- First interesting particles had to be found and preselected.
- Afterwards a new trajectory dataset had to be created for each selected particle.
- Then a preview of the trajectories was created and the datasets were sent to the scientists involved to select which particle is followed.
- Finally, once a particle was selected, the dataset was visualized.

The tool used does not support the visualization of time dependant trajectories based on the particles spatial positions. Therefore it was necessary to create a time independent line database for each particle, by extracting its location through time. The rendering of each time step, to create a video following a particle inside the fluid, takes several days. This motivated the use of virtual reality techniques which is the main subject of this thesis. Based on an existing software framework that is used to explore molecular and chemical simulations in the field of materials science a comprehensive virtual reality application for the HTC Vive was developed in order to visualize turbulence data.

1.2 Goals

To reach the objective of this thesis, to visualize and efficiently explore turbulent fluid simulation data on a desktop computer, certain goals needed to be achieved.

The first and foremost goal was to make a three dimensional visualization of the original peta scale simulation data set, by extending an already existing virtual reality application. As explained in the previous section 1.1, there are some problems with two dimensional visualizations of three dimensional information. In order to be able to use spatial orientation properly, the image must be dynamic. In other words, the camera needs to be rotated in order to maintain the impression of a three dimensional perspective. This is needed to find and isolate important information, like the particles' corkscrewed trajectories and their spatial positioning relative to vortex tubes. Through the inherent three dimensional properties of virtual reality, it is possible to intuitively figure out where everything is, without the rotation of the data. This allow users to explore the data without the need to learn to interpret a two dimensional representation as three dimensional.

Another main goal is the efficient visualization of peta scale simulation data. The dynamic exploration of peta sized simulation data with conventional visualization tools requires the use of high performance computers. For the virtual reality application three dimensional isosurfaces and the spatial position of the particles are extracted from the whole simulation. The application then uses only the extracted information instead of the whole data set for the visualization. Through the extraction the data size is reduced to gigabyte scale, thus allowing the use of desktop computers, equipped with virtual reality technology, instead of high performance computers.

The extracted isosurfaces, even after compression, are still too big to be fully loaded by the original application. Thus the source code of the original application to load isosurfaces had to be rewritten. To accommodate for the large data size a multi threaded approach to load isosurfaces was chosen. This allows the user to explore the current data, while new data is simultaneously loaded. The main motivation for this approach was to keep the user immersed in the virtual world.

The next goal was to visualize additional information, to make the exploration easier. These are the automatic calculation and visualization of each particle's trajectory and various vectors. The trajectories help to understand the relative position of each particle to the vortex tubes, while the vectors provide information about their spatial orientation, speed and change thereof.

The last goal was to implement different interactive controller functionalities, that aid scientists in the exploration of the visualized fluid simulation data. Some are more general, to control the amount of information shown and to assist in the navigation through the volume, while others are more specialized in order to accommodate for the theoretically infinite structure of the isosurfaces.

This allows scientists to efficiently explore volumetric data sets in three dimensions with a desktop computer.

1.3 Structure

The last part of the introduction gives an overview of the content of this thesis.

Chapter 2 is about the hardware and software used to visualize the fluid simulation in virtual reality. Therefore the HTC Vive Virtual Reality Systems available are introduced and an overview of their cost as well as some technical information, like its weight and resolution, is given. The specifications of the computer used to test the virtual reality application can also be found here. This chapter also introduces the multi platform NOMAD VR virtual reality application, how it is used and details about the HTC Vive implementation. The last part explains in detail all the steps necessary to visualize a fluid simulation in the extended virtual reality application of NOMAD VR.

Chapter 3 presents the extended application and its capabilities. Therefore it introduces the reader to the functionalities of the created application. Afterwards it explains what the individual controller buttons do, their technical names and explains what a controller mode and option is. It also provides examples for each of the interactive controller functionalities implemented as part of this thesis. Therefore some important terminology, used for the rest of this thesis, is explained. Namely the difference between the scene (everything that is visualized), the simulation box (only the isosurfaces visualized) and the virtual world (the space the user moves in). In the last part of this chapter the motivation of a multi threaded approach to load isosurfaces is explained.

Chapter 4 provides technical details about the changes to the application and how they were implemented. This includes information about the textures used to visualize the controller display, how the display is changed and what changes internally, when each controller option is used. An overview of the vector visualization is given, how they are calculated and the motivation of using an adaptation of the Frenet-Serret formulas for their visualization. Finally the interaction between the main and loading thread is explained with the aid of a simplified diagram, which shows the thread interactions for loading the first 100 isosurface graphic files. A more detailed version of the diagram can be found in the appendix.

Chapter 5 begins with a summary of which goals of this thesis were reached. It compares the virtual reality application with two dimensional visualizations and weighs up its usefulness for the exploration of turbulent fluid simulations. It introduces ideas on how to generalize and improve the application and concludes with a list of known issues of the prototype.

The last chapter 6 of this thesis are acknowledgements to the scientists and supervisors that were involved in this project.

In the Appendix detailed code examples, of how to prepare the data, can be found. It also provides an example configuration file, which is needed to start the virtual reality application and a more detailed version of the multi threading diagram used in chapter 4.

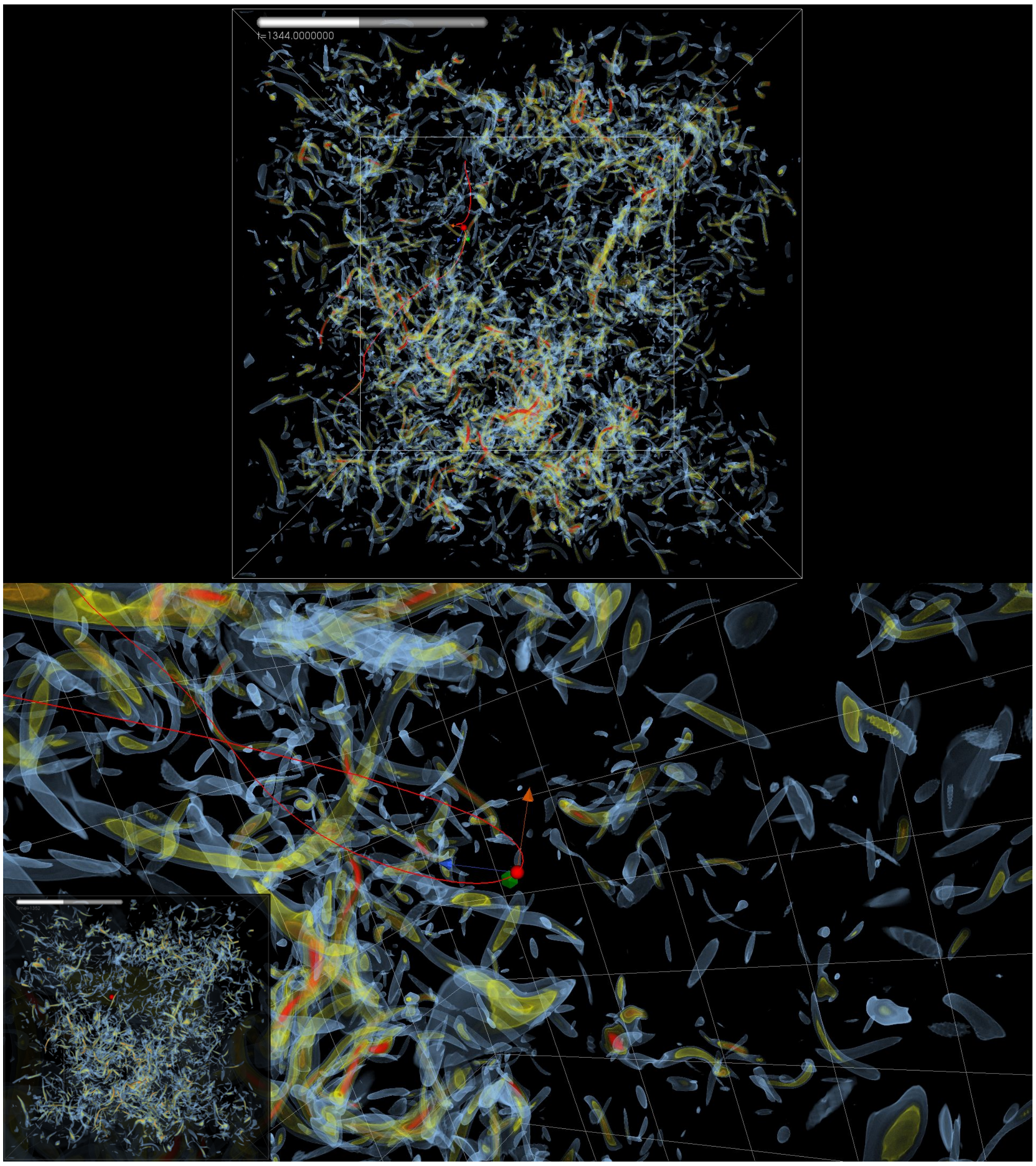


Figure 1.1: Pictures of a turbulent fluid simulation showing the same particle (red), its trajectory (red line), velocity (green arrow), acceleration (blue arrow) and their crossproduct (orange arrow). The particle is surrounded by vortex tubes, called isosurfaces. The camera is positioned outside the simulation (top) and behind the particle (bottom). [The pictures were rendered with VisIt.]

2 Architecture

This chapter gives an overview of HTC Vive Virtual Reality System, explains what NOMAD VR is and the software used to prepare data to visualize large scale turbulence simulations in NOMAD VR. Section 2.1 will explain what HTC Vive is and give an overview of its hardware specifications as well as its current pricing. Section 2.2 shortly explains what the virtual reality graphics tool NOMAD VR is and how it is used and section 2.3 presents the steps necessary to visualize a turbulence simulation in NOMAD VR.

2.1 HTC Vive Virtual Reality System and the Computer Used

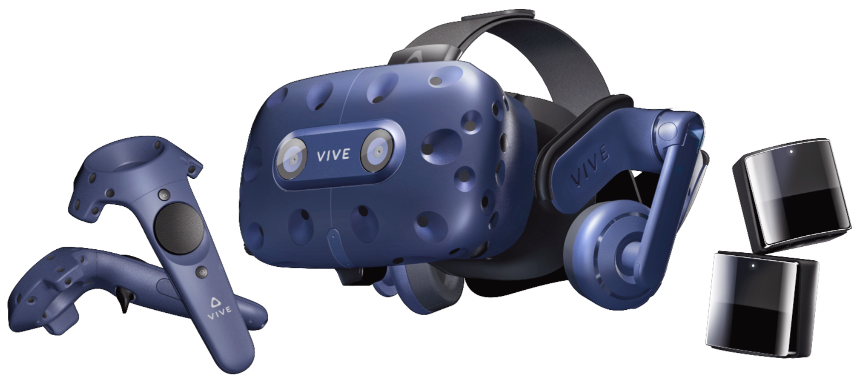


Figure 2.1: Picture of the HTC Vive Pro Virtual Reality System. Showing the head-gear (middle), controllers (left) and the infrared trackers (right). (Picture provided by the HTC Team, ©HTC `https://drive.google.com/drive/folders/1lBT58yEtshY9VxPXOYgt2QEdcTYzIZ_B`)

The HTC Vive is a Virtual Reality System consists of a headset, two controllers and two trackers enabling room scale tracking. The headset's field of view is approximately 110° through two displays with a resolution of 1080×1200 pixels per eye, that are updated at 90 Hz and weights about 555g.

The two trackers are placed diagonally to each other and can cover an area of about 4×4 meters. It is possible to place the infrared laser emitter units further apart, without visible loss of quality [NLL17]. The HTC Vive Pro has an upgraded headset with a display resolution of 1440×1600 pixels per eye, as well as new controllers and trackers that can be placed further apart (6x6m).

The full HTC Vive Virtual Reality System costs about 599 Euro, the HTC Vive Pro Virtual Reality System about 1399 Euro, both are purchasable from the official homepage `https://vive.com`, where the aforementioned specifications can also be found. (Prices from September 25, 2018)

The computer used to test the extension of the NOMAD VR application has an 'Intel® Core™ i7-4790K CPU @ 4.00GHz 4.00 GHz' processor and 32 gigabyte RAM. The disk drive installed is a 'Samsung SSD 850 PRO 512 GB ATA Device' and its graphic card is a 'GeForce GtX 980'. The operating system installed is a 64 bit version of Windows 7.

2.2 NOMAD VR

"The Novel Materials Discovery (NOMAD) Laboratory develops a Materials Encyclopedia and Big-Data Analytics and Advanced Graphics Tools for materials science and engineering." [NOM15]

NOMAD VR is part of the NOMAD project and one of the 'Advanced Graphics Tools', created to explore scientific simulations in the field of material science in virtual reality. It's a free multi platform, virtual reality, scientific visualization tool for OpenVR (HTC Vive and Oculus Rift), CAVE-like installations, Google Cardboard (Android and iOs) and GearVR. The source code is licensed under the Apache License, Version 2.0 and is available at gitlab. NOMAD VR was developed by Dr. Rubén Jesús García-Hernández. The HTC Vive application for SteamVR is compiled with Microsoft Visual Studio 2012 and is implemented in C++ and OpenGL.

The NOMAD VR application, for any platform, basically consists of two files. An executable file starting the virtual reality application and a configuration file, a plain text file created by the user containing information about the dataset visualized, its location on the hard disc drive and various visualization settings. The application is started by drag and drop of the configuration file onto the executable.

As most users had at least slight problems creating a configuration file for the first time, a fully commented example used to visualize the turbulence simulation data can be found in the appendix. The # symbol in the configuration file is used for comments. A full list of commands is available on the official NOMAD homepage `https://nomad-coe.eu/`.

The goal of this thesis is to extend the NOMAD VR for the HTC Vive Virtual Reality System, which uses the OpenVR library version 0.9.19 , in order to enable it to visualize large scale data sets of turbulence simulations for scientific exploration.

2.3 Data Preparation

This section aims to give an overview of the process, shown in figure 2.2, of extracting data from the original turbulent fluid simulation data set and then transforming them into file formats readable by NOMAD VR. The complete scripts used for the data preparation can be found in the appendix. Each step and the software used will be explained in this chapter.

The preparation is split into two main segments: the data extraction on a high performance computer node from the original data set using ParaView [AJG+05] and the data processing on a desktop computer. The processing is done via MeshLab [CCC+08] to convert and simplify the extracted files, Python [Ros95] utilizing Jupyter Notebook [KRKP+16] to finalize the particle file set and simple shell scripts to rename the initially extracted files. The software used for the data preparation is, like all other visualization software used in this thesis, open source.

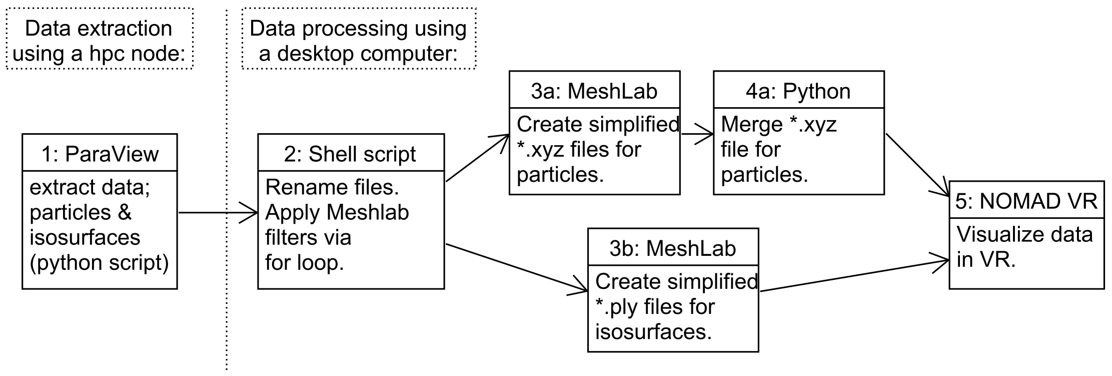

Data extraction using a hpc node:
Data processing using a desktop computer:
1: ParaView
extract data; particles & isosurfaces (python script)
2: Shell script
Rename files. Apply Meshlab filters via for loop.
3a: MeshLab
Create simplified *.xyz files for particles.
4a: Python
Merge *.xyz file for particles.
3b: MeshLab
Create simplified *.ply files for isosurfaces.
5: NOMAD VR
Visualize data in VR.

Figure 2.2: Showing the steps to prepare data for NOMAD VR. From left to right: Step 1, 2, 3a & 4a (top branch), 3b (bottom branch) and 5. (Created with UMLet 14.2 `www.umlet.com`.)

To extract isosurfaces from the original data sets vorticity field within acceptable time, a remote high performance computing node is used. This allows to extract approximately one isosurface per minute. Visualizing the whole data set requires the extraction of at least 3600 isosurfaces and the spatial positions of all particles.

The simulation data set is viewed in ParaView to find meaningful isosurface values, which can be extracted. An example script to extract isosurfaces is provided at the NOMAD VR tutorial homepage. Paraview's tracing mode [Aya16, Chapter 1.6.2, p. 18] is used to adjust this script with the correct file reader, variable names and transformations for the vorticity field and the particles. The extraction is then done for each time step, by utilizing one of ParaView's Python script interpreters. By extracting only a few isosurface values per time step from the fluid simulation, the size of the dataset aimed to visualize is already reduced from multiple petabyte to terabytes. The data is then downloaded for further processing on a desktop computer. This concludes step 1 in figure 2.2.

ParaView exports Virtual Reality Modeling Language v2 (VRML v2) files as *.vrml, MeshLab expects VRML v2 files named as *.wrl (this is also documented in the NOMAD VR Tutorial 1). As both file endings are essentially the same file format, step 2 in figure 2.2 uses shell scripts to rename the downloaded files into a format readable by MeshLab.

Converting particles in Step 3a from *.wrl to *.xyz with MeshLab removes the header and unwanted columns from the files. The result are simple files, where each line only contains three floats representing the x,y and z coordinates of a particle. A Python script, executed with Jupyter Notebook, is used in Step 4a to merge and convert them into xyz files, a common file format used to save chemical structures and readable by NOMAD VR. The particle extraction, conversion and merging takes less than an hour and the resulting file, containing positions of 1.800.000 particles, has a size of ≈ 55 megabyte.

In Step 3b the VRML v2 files containing the isosurfaces are converted into a Polygon File Format (ply), reducing their size to approximately 1/3 of their original size. Additionally a simplification filter was applied to reduce the size of each isosurface to roughly 15 megabyte. After manually applying the filter and saving it once for each isosurface value, the file conversion is automated by shell scripts.

Table 2.1 shows the time needed to convert and reduce four different isosurface values with MeshLab on a 1.7 GHz dual core laptop and a 3.4 GHz quad core desktop computer, as well as their original size, amount of vertices and the percentage reduction. The times measured in table 2.1 are approximations based on the time needed to convert and reduce isosurface graphic files for the first 200 time steps.

The results of this process are a single particle file, containing the location of all particles at all time steps and n isosurface files for each time step, where n is the amount of isosurface values extracted at each time step from the original dataset. These files can be visualized with NOMAD VR.

isosurface level	wrl size in MB	vertices in thousands	vertex reduction to	avg. time needed per file in min. 1.7 GHz	3.4 GHz
80	901-861	5503	5.5%	14	6
95	410-385	2526	11%	5	3
110	193-179	1231	25%	2	1
125	97-88	614	50%	1	0.5

Table 2.1: Original data sizes, vertices, vertex reduction and processing times with Mashlab to reduce the extracted world files to approximately 15MB and 300000 vertices and to convert them to the Polygon File Format.

3 The Virtual Reality Application to Visualize Turbulence Simulations

This section explains what the application created in this thesis is capable of, thereafter gives an overview of the controllers basic button mapping, shows their functionalities, as well as explains how they are used and why interactive controller modes were implemented in order to efficiently explore a turbulence visualization. The new controller modes and displays are explained based on different challenges presented by visualizing a turbulent fluid simulation. All technical details of the functionalities mentioned here are explained in the following chapter 4.

3.1 Core Features of the Application

The prototype extension of NOMAD VR, for the HTC Vive Virtual Reality System, to visualize turbulent fluid simulation data allows the user to explore the correlation between visualized vortex tubes and the movement of the advected tracer particles in a three dimensional environment.

The first part aiding scientists in the exploration of a data set is the creation of a customizable configuration file. This file allows to adjust general information, like the location of the data set a user wants to explore and how many time steps it consists of. It also enables the user to customize more specific aspects of the visualization. These options include the ability to change the scaling factor the application uses when it is started, or to define how many particle trajectories and which trajectories are shown. Other parameters allow the user to customize the size and colour of the particles, as well as colour and opacity of the different isosurface values.

Additionally various vectors are automatically calculated. The calculation is based on the particle file, which consists of the spacial location of all particles. These vectors include each particle's velocity, acceleration and the crossproduct of these two vectors. While the application is running, a controller option is provided to disable the rendering of some or all of these vectors.

To allow for an immersive experience various other controller functionalities were implemented. These allow users to adjust different settings during the visualization. The functions available were designed to increase the users comfort, spatial orientation and to allow them to single out and observe any single particle. Therefore different functions, some inspired by already existing applications, were implemented, one of which allows to manually change the scaling of the simulation while the application is running. Rescaling the simulation at any time, enables users to get an overview of the whole simulation and then to inspect parts of the visualization close up, by increasing its size. An optional display, showing the second controllers current coordinates, helps users to find their original position again, after they rescale the simulation.

Another option moves all visualized vortex tubes along one of the three coordinate axes. The axis is chosen based on the controller position and the distanced moved along the axis equals the size of the visualized vorticity field. This ability, to translate isosurfaces, accommodates for the theoretically infinite size of the vorticity field.

When a single particle is selected, it is marked by orange circles. Thereafter it is possible to automatically follow the particle through space while time is advanced, which can be useful if users don't want to follow it manually with the first controller, while advancing time with the second.

Other functions available while a particle is selected allow to hide the other particles, their trajectories and vectors. Additionally it is possible to show only a single, specific isosurface value, instead of all at once. These functions are intended to control the amount of information shown and to help to deal with a possible information overflow.

By utilizing OpenGL's shared context functionality the visualization of large scale data sets is realized. This allows the exploration of the data set, while new graphic files are seamlessly loaded by a second thread. Using this method enables users to explore large data sets without seeing a loading screen while new data is loaded by the application.

3.2 Button Mapping

This section explains the basic button mapping of the controllers (figure 3.1) and the names of these buttons. The specific button names used here, can be found in the official HTC Vive user manual [Cor16]. Holding down the trigger of the first controller allows the user to

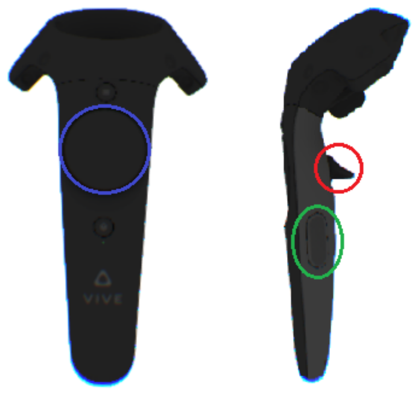


Figure 3.1: Picture of a HTC Vive controller, front (left) and side (right). Circled in different colours are the trackpad (blue), the grip button (green) and the trigger (red). (Pictures used are screenshots taken from the HTC Vive application.)

fly freely in the direction the controller is pointed, pressing its grip button advances to the next time step. Holding down the second controller's trigger advances time continuously. Its grip button changes which isosurfaces are currently shown. The second controller's trackpad is used to select and interact with the different modes and their options. By pressing or touching the trackpad in the upper or lower areas the user can change between the different modes. The left and right corners of the trackpad are used to change between the options available of a currently selected mode.

The application recognizes a controller once a mapped button is used. Therefore the first and second controller are defined by the order in which the user chooses to interact with them after the application is started.

3.3 Using the Controller Modes

The interactive controller modes allow a user to change different parameters during the visualization, which would otherwise require restarting the application with changed parameters in the configuration file, or require the user to create new datasets.

To efficiently explain them, the basic terminology used in this section needs to be explained. The space the user moves in will be called the world or virtual world. The vorticity field visualized has a cubic structure. The isosurfaces are visualized inside this cube, which will be called the simulation box. The scene includes isosurfaces, particles, their trajectories and vectors and consists of the data visualized. Touching the trackpad in the upper or lower area changes the current controller mode, touching the left or right corners of the trackpad changes the current mode's options.

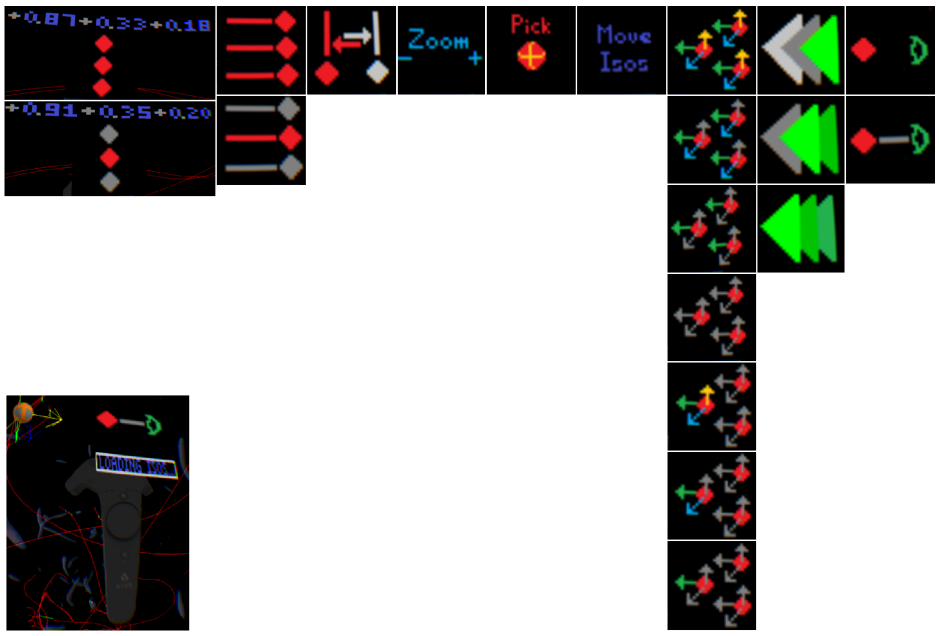


Figure 3.2: Picture showing the nine available controller modes (left to right). Mode 1 (left) additionally shows the controller coordinates. In the bottom left the controller with the loading bar is shown. (Pictures used are screenshots taken from the HTC Vive application.)

An overview of the nine default controller modes is shown in figure 3.2. A short summary of the controller modes is provided here, before a detailed explanation.

1. Toggle Particles: Shows either all particles or a single one. Also displays the controller's current coordinates.

2. Toggle Trajectories: Shows either all or a single trajectory.

3. Next/Previous Trajectory: Shows next or previous single trajectory.
4. Zoom: Changes scaling of the scene.
5. Pick Particle: Marks a particle with orange circles. Also used for modes 1, 2, 7 and 9.
6. Move Isos: Translates isos, short for isosurfaces, in the axis direction the controller is facing.
7. Vectors: Shows different vectors of all or a single particle.
8. Speed: Changes the speed with which the user progresses through time when holding down the trigger.
9. Camera: Offers a free mode or moves the user parallel to a selected particle as time is advanced.

The scaling option is available before starting the application, as a configuration file option and also as a controller option while it is running. The scaling parameter rescales the entire scene by a fixed number. The controller option reduces or increases this number by 0.2 each time the trackpad is pressed. The decrement cannot become smaller than 0.2.

Changing the scaling of the scene does not change the origin of the world or the user's current position, thus rescaling the scene changes the users relative position to the scene (figure 3.3). To make spatial orientation easier an extended display on the second controller shows its relative coordinates, while the first mode is selected. This way users can find their relative position in the scene again, after they changed its scaling, or continue their exploration from the same location after restarting the application. During the turbulence simulation most particles leave the visualized simulation box. To be able to see the particles movement inside the vorticity field again the user can translate the position of the isosurfaces in the axis direction the controller is currently pointing, by touching the trackpad on the right side, or reset their position back to default, by touching the left side (figure 3.4 (bottom)). In Figure 3.4 (top left) the observer is floating inside the simulation box. Seeing all trajectories at once might give a good impression of the particle flow, but obscures the trajectory of individual particles. Different modes were designed to reduce the information overflow, allowing users to choose which and how much information can be seen at once. Using the pick option allows to manually select a particle with the controller, which is then highlighted by three orange circles surrounding its surface. The controller's haptic feedback is utilized to indicate if a particle is selectable. After a particle is picked, it is possible to disable the other trajectories and only a single trajectory is shown. The user can then toggle between the trajectory modes at any time to compare it to surrounding trajectories (figure 3.4 (top)). As all trajectories are internally saved in a list, controller mode 3 allows the user to go through this list one by one. While only a single trajectory is shown, using this mode can show each particles trajectory, one after the other, without the need of manually picking each particle.

By default each particle's scaled velocity is shown, its acceleration and the crossproduct of the latter is also shown after the first time step. While the velocity uses a representative scaling, those of the acceleration and crossproduct currently use artificial scaling values. The vector visualization intends to give information about the particle's spacial orientation and current speed. Controller mode 7 provides the ability to reduce the amount of vectors shown, allowing to render only the velocity and acceleration, the velocity alone or no vectors

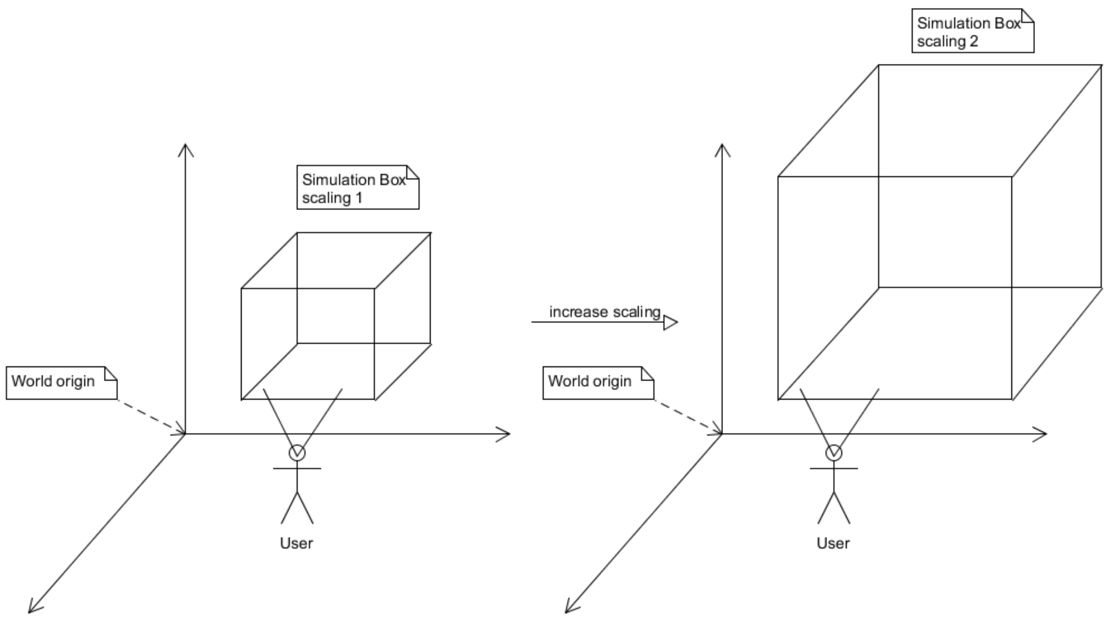


Figure 3.3: Rescaling the scene does not move the user, thus changing what the user sees. (Created with UMLet 14.2 `www.umlet.com`.)

at all. These options are also available for a single selected particle vectors. In case of a selected particle being obscured by others, it is possible to hide all other particles with the first controller mode. Combining the different options makes it possible to reduce the information shown to a single particle, its trajectory and a single isosurface.

The eighth mode alters the video speed, allowing a user to double or triple the speed with which the visualization is advanced through time while the trigger button is held. The ninth and final mode allows to use an alternate camera option, which tethers the user to a picked particle and automatically moves the user along with the particle movement. This removes the need to manually follow a particle while advancing through time.

3.4 Data Pre-Loading

Smaller parts of the visualized data, like the particle data set, can be loaded completely to the graphic card memory, when the application is started. The files containing the isosurface structures are too big to be completely loaded while the application is started and reducing them to a size where this would be possible removes too many details from the isosurface graphic files, rendering them useless for scientific visualization and exploration.

Consequently they need to be partially loaded while the application is running. Loading the data in the main thread would stop it from rendering while it is loaded, thus breaking the user's immersion. Therefore a second thread to pre load the next chunk of data when needed was implemented, allowing the user to simultaneously explore and advance through time of already loaded parts of the visualization while new data is loading in the second thread. The loading progress of the isosurfaces graphic files is indicated by a small loading

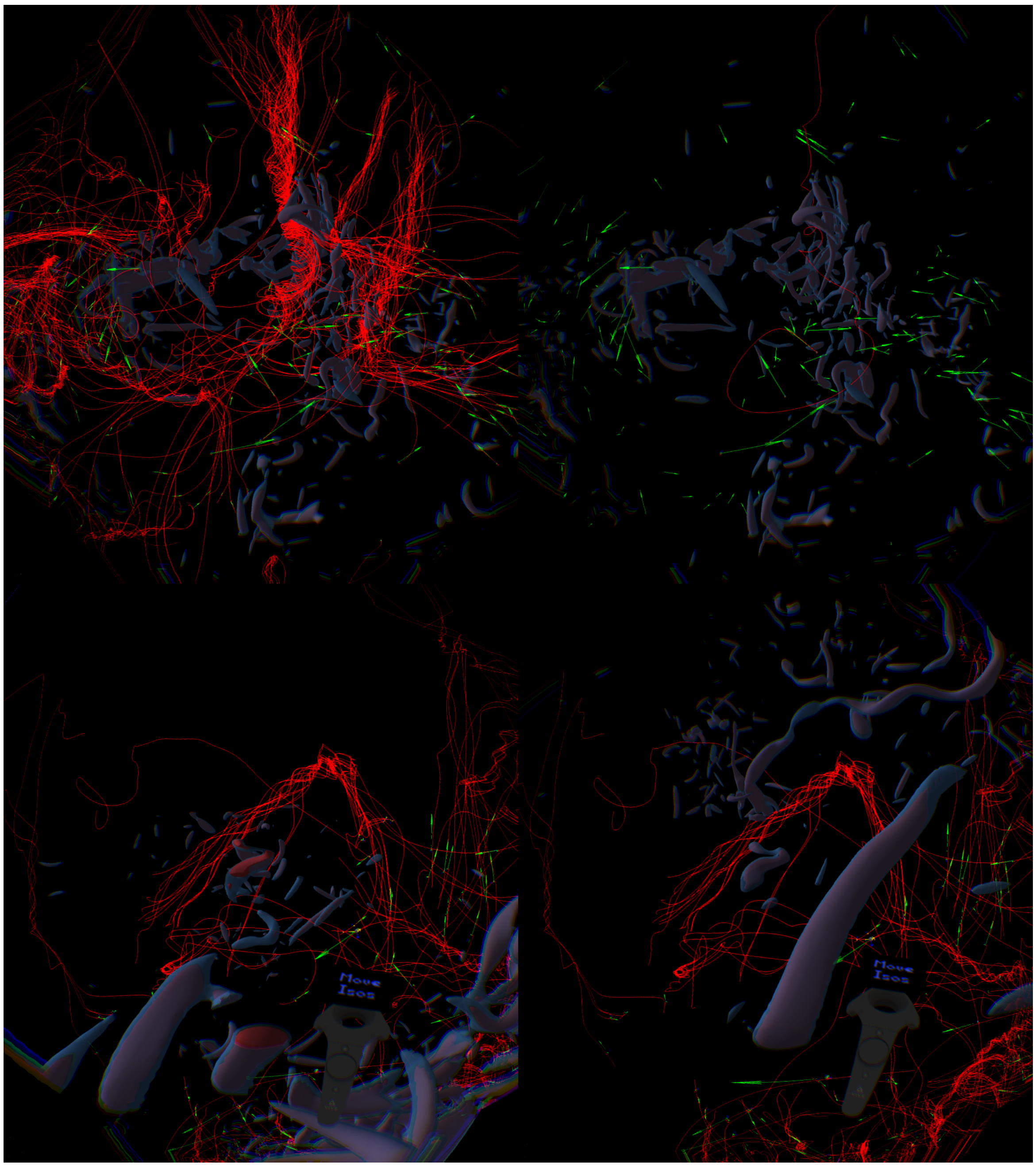


Figure 3.4: Picture showing all trajectories (top left), a single picked particle's trajetory (top right) and the translation of isosurfacres (bottom). (Pictures used are screenshots taken from the HTC Vive application.)

bar below the display shrinking over time, shown in figure 3.2 (bottom left). Technical details are explained in section 4.4.

4 Implementation on top of NOMAD VR

In order to visualize and explore the turbulent fluid simulation data efficiently, additional functionalities for the HTC Vive application of NOMAD VR were implemented. This section explains their implementation and provides technical details.

4.1 Controller Modes

This section introduces the textures used and their specifications, how the different controller display modes and options are checked and how they were realised. The textures for all three displays are loaded when the application is started and consist of three independent files. The texture files need to be in the same folder as he data set visualized. These three files are the selectable modes, the controller position and the loading bar textures. Each of them are simplistic, hand drawn pixel graphics.

The digits to display the controller's current location and the loading bar texture files each consist of 64×7 pixels. A single digit consists of 4×7 pixels, a letter of up to $5x7$ pixels. The controller modes use a 672×32 pixel texture, using 32×32 pixels for each option. This was done to see how OpenGL would render textures consisting of very few pixels.

To check which controller mode texture is shown, a one dimensional integer array and a function, checking if the current value is viable and has a texture, are used. Changing the current mode changes the current mode array's index, changing an option changes the array's value, then the texture rendered is updated (section 3.3 explains the difference between mode and option). A simplified example of how the current mode is changed is provided here.

Listing 4.1: Changing the current mode:

```
if(trackpad: Up/down){
  if( 0 >= currentIndex < currentMode.length() )
    currentIndex--/++;
                  currentTexture = currentMode[currentIndex];
}
//In rendering loop:
//glDrawElements of currentTexture above controller
```

Each mode has a corresponding variable, changing different parameters of the visualization. Interacting with a controller mode has usually two steps. Changing the current texture, then updating all variables.

Listing 4.2: Interacting with a mode:

```
if(trackpad: Left/Right){
  //change current texture
  currentMode[currentIndex]++/--;
  currentMode[currentIndex] =
    checkIfViable(currentIndex,currentMode[currentIndex]);
  currentTexture = currentMode[currentIndex];
  //update variables
  toggleParticles = currentMode[0] - startingValues[0];
  toggleTrajectories = currentMode[1] - startingValues[1];
  if(currentTrajectoryIndex in range && currentMode==2)
    currentTrajectoryIndex--/++;
  if(currentMode == 3)
    scaling+/-=0.2;
  if(scaling < 0.2)
    scaling = 0.2;
  //Pick Particle (currentMode[4]) --> HapticFeedback
  //Move Isos, reset to origin
  if(currentMode == 5)
    moveIsos = get Min/Max of controller axis orientation;
  toggleVectors = currentMode[6] - startingValues[6];
  videoSpeedModifier = (currentMode[7] + 1) - startingValues[7];
  cameraMode = currentMode[8] - startingValues[8];
}
//In rendering loop:
//glDrawElements of currentTexture above controller;
```

The rest of this section gives an overview of each controller mode, which function is called and which parameters are changed.

The toggleParticles uses a value of 0 to show all particles or 1 to show only a single one. While the default option uses an array of all particle locations, the other option can only show those particles, which trajectories were loaded during the start up. The application's original mode of operation selects the trajectories of the particles which are to be shown from the configuration file, at dataset load time.

Listing 4.3: Controller Mode 1, toggle between all & single particles rendered:

```
if(toggleParticles==1){
  //draw for a single particle at time step 0
  ...
  glDrawArrays(GL_PATCHES,
    particletrajectories[currentParticletrajectoryIndex]);
}else{
  //default draw at time step 0
  ...
  glDrawArrays(GL_PATCHES, 0, numParticles[0]);}
```

As mentioned in section 3.3 an additional display showing the controllers relative spatial position is rendered while the first mode is selected. The coordinates displayed range from $(-9.99, -9.99, -9.99)$ to $(+9.99, +9.99, +9.99)$ and will turn green if the controller leaves this area. The particles maximum position is $(6.10, 1.52, 9.98)$, thus satisfying the minimum requirements for this visualization. A single digit consists of 4×7 pixels and it was found that GL_LINEAR and GL_LINEAR_MIPMAP_LINEAR lead to unwanted border effects, if the texture is too small. GL_NEAREST and GL_NEAREST_MIPMAP_NEAREST are used to avoid these effects.

Satisfying only the minimal requirements and using textures with as few pixels as possible was chosen out of performance issue concerns, as the coordinate display requires to render 16 additional textures per frame. As no performance issues arose, the next step for this display is to generalize it for other datasets and change where the coordinates are shown.

The second controller mode, toggling trajectories, works in a similar way to the first one. Visualizing trajectories only works correctly if each time step has the same amount of particles and each particle has the same position in the data file. For example, the tenth particle needs to be the tenth particle at each time step in the data file and each time step needs to consist of the same amount of particles.

The third mode increase or decreases the value of currentParticletrajectoryIndex, thereby changing the particle and trajectory rendered.

The option to select a particle with the controller was added to the HapticFeedback function. Haptic feedback makes the controller vibrate while it is held inside a particle. Selecting a particle sets the selectedParticle variable, marking a particle with three orange circles. The intensity with which the controller vibrates changes, while the trackpad is used to indicate that the selection was successful. Then the currentParticletrajectoryIndex is updated, thereby changing which single particle, trajectory and vector arrows are rendered. The mark uses a different variable to allow switching to new trajectories with the third mode while keeping the original particle marked.

Listing 4.4: Controller Mode 4, selecting a particle:

```
if(controller inside particle i){
  if( (trackpad_is_used) && (currentMode == 4) ){
    selectedParticle = i;
    m_pHMD->TriggerHapticPulse(device, 0, 1000);
    currentParticletrajectoryIndex = selectedParticle;
  }else{
    m_pHMD->TriggerHapticPulse(device, 0, 3000);
  }
}
```

The toggleVectors variable ranges from 0 to 6 and is split into three main rendering options. If it is smaller than 3, vector arrows for all particles are rendered; if it is bigger than 3, vector arrows for a single particle are rendered and if it equals 3 no arrows are rendered. Inside these rendering options, is a second layer checking which vector arrows are rendered.

To render the vector arrows correctly the particle data set needs to fulfil the same requirements as for the trajectory visualization. How the different vectors are calculated is explained in section 4.2.

Listing 4.5: Controller Mode 6, selecting which vector arrows are rendered:

```
if(toggleVectors < 3){
  //render velocity of all particles
  if(toggleVectors < 2)
    //render acceleration of all
  if(toggleVectors < 1)
    //render crossproduct of all
} //if(toggleVectors == 3) render no arrows
if(toggleVectors > 3){
  //render velocity of a single particle
  if(toggleVectors < 6 )
    //render acceleration of a single particle
  if(toggleVectors < 5)
    //render crossproduct of a single particle
}
```

To correctly render the scene, the particles and isosurfaces each use a transformation matrix, containing information about the rendered textures orientation, spatial positioning and scaling. The zoom option allows the user to modify the scaling of these transformation matrices. By changing the particles scaling matrix, their trajectories and vector arrows automatically rescale, as they are both dependant on the spatial position of the particles.

'Move Isos' first option calls a function, that takes the controller's directional vector, calculates the maximum of its absolute axis values and returns a one dimensional axis translation vector, with a length equal to the edge length of the simulation box. The returned vector is then used to translate the isosurfaces' transformation matrix accordingly.

Using this option multiple times adds up the resulting translation vectors, allowing users to move the isosurfaces where needed. The second option of this mode sets the translation vector back to (0,0,0), resetting the rendering position of the isosurfaces back to their initial location.

The video speed option changes a variable, which is multiplied with the animation speed, which determines how fast time is advanced while the trigger is held.

While the camera is tethered to a particle, its unscaled velocity is added to the user position whenever time is advanced.

4.2 Vector Visualization

> "In classical differential geometry, the Frenet-Serret formulas describe the kinematic properties of a particle moving along a continuous, differentiable curve in three-dimensional Euclidean space $\Re^3$, or the geometric properties of the curve itself irrespective of any motion. More specifically, the formulas describe the derivatives of the so-called tangent, normal, and binormal unit vectors in terms of each other. The tangent, normal, and binormal unit vectors, often called T, N, and B, or collectively the Frenet-Serret frame, together form an orthonormal basis spanning $\Re^3$ and are defined as follows: T is the unit vector tangent to the curve, pointing in the direction of motion. N is the normal unit vector, the derivative of T with respect to the arclength parameter of the curve, divided by its length. B is the binormal unit vector, the cross product of T and N." [Yan16]

The Frenet-Serret formula was adapted to visualize more information, about the particles spatial orientation and movement. Figure 4.1 illustrates the process. The tangent $\hat{T}$, normal $\hat{N}$, and binormal $\hat{B}$ orthonormal vectors (figure 4.1 (bottom)) were, as a first step, replaced by the particles velocity $\vec{v}$, acceleration $\vec{a}$ and their crossproduct $\vec{c}$. These vectors were extremely small and either barely visible during the visualization, or even ended up inside a particle.

The second step was to prevent the vectors ending up inside a particle. This was done by adding twice the diameter of a particle to each vector (figure 4.1 (middle)).

As a last step a representative scaling factor for the velocity was introduced and used instead of an arbitrary length (figure 4.1 (top)). This factor was calculated by one of the creators of the turbulent fluid simulation.

This was done to give scientists more information about the dataset visualized. While the normalized acceleration and crossproduct vectors visualize the spatial orientation of all particles, the scaled velocity also reveals scientific information about the speed and its change, while moving through the vorticity field. This is especially interesting when a particle moves close to a strong vortex tube, as their spatial orientation and speed can change rapidly. All vectors are calculated by utilizing the spatial positions of the particles through time, when the application is started.

$$velocity\ \vec{v}'_{t_i} = (\vec{x}_{t_{i+1}} - \vec{x}_{t_i}) \times scaling\_factor$$

$$acceleration\ \vec{a}_{t_i} = (\vec{v}_{t_i} - \vec{v}_{t_{i-1}}) + |\vec{v}_{t_i} - \vec{v}_{t_{i-1}}| \times 2 \times particle_{diameter}$$

$$crossproduct\ \vec{c}_{t_i} = (\vec{v}_{t_i} \times \vec{a}_{t_i}) + |\vec{v}_{t_i} \times \vec{a}_{t_i}| \times 2 \times particle_{diameter}$$

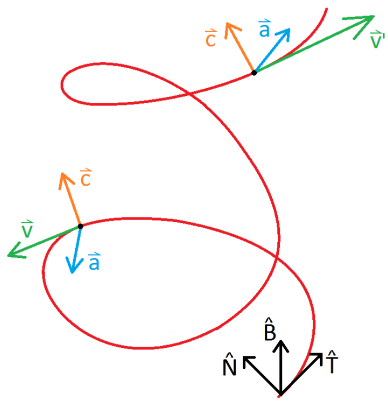


Figure 4.1: Adaptation of the Frenet-Serret formulas showing the orthonormal vectors $\hat{T}$,$\hat{N}$,$\hat{B}$ (bottom). The arbitrarily scaled velocity $\vec{v}$, acceleration $\vec{a}$ and crossproduct $\vec{c}$ (middle) and the final version, using scaled $\vec{v}'$, $\vec{a}$ and $\vec{c}$ vectors (top). (Graphic adapted from the wikipedia entry of the Frenet-Serret formula. `https://en.wikipedia.org/wiki/Frenet-Serret_formulas` last visited: 2018, August, 22)

4.3 Camera Mode

The two dimensional rendered video made heavy use of a moving camera perspective, which chased a particle. The camera orientation depended on the particle's velocity and up-vector, a vector perpendicular to its velocity and acceleration. By co-moving the user through the fluid simulation, following a particle closely, its behaviour, especially near strong vortex tubes, was easier to observe.

Therefore it was tried to recreate this experience in virtual reality, by moving the user slightly behind and above a chosen particle and aligning their orientation to the particle's spatial orientation.

After presenting an early version of the application, which included different camera rotations, to one of the creators of the turbulent fluid simulation visualized in this thesis, a major issue with camera rotations was found. Each rotation lead to a loss of all reference points the user had, leading to a loss of orientation after each time step and thereby made it impossible to efficiently analyse the visualized data.

It was concluded that, even if camera smoothing algorithms were implemented, a static spatial framework is preferred for scientific visualizations in virtual reality. Any information gained by following a particle in the two dimensional video, can easily be obtained through the user's freedom movement inside virtual reality.

The only camera mode accepted was one not using any rotations, simply co-moving the user with a selected particle. This allows them to advance through time and follow a selected particle, without the need to simultaneously navigate manually through the visualization.

This was implemented by calculating the distance a particle moves between two time steps and adding the resulting vector to the user's current position, whenever time is advanced.

4.4 Changes to the Data Loading

To ensure a lag free and immersive virtual reality experience NOMAD VR loads the whole data set to the graphic card memory, when the application is started. This approach works excellent for data sets that are not bigger than the graphic card memory.

Even after reducing the isosurface files to the minimal resolution acceptable for the scientists involved, a single file has about 15 MB and two isosurfaces are shown per time step. The total data size for all 1800 time steps would amount to approximately 162 GB, thus the data loading process needed to be changed. Any approximation in this section is based on the first 200 time steps of the simulation, which were used to test the application.

To accommodate for the amount of data that needs to be loaded for the visualization and to keep the program from breaking immersion while new data is loaded, a multi threaded approach was chosen. This enables the application to render already loaded graphic files in the main thread while new isosurface files are loaded by a second, loading thread.

The thread communication is realized through the shared context functionality, provided by OpenGL [MS17, Chapter 5]. Therefore two different contexts are created, when the application is started and the shared context attribute is set. Once the second thread is initialized, its active context is set to the one not used by the main thread. The two threads can now share various different data.

One characteristic of the shared context is, that only some graphic objects are shared between the contexts. Container objects, such as the vertex buffer, cannot be shared and need to be bound manually in each context. As long as the shared graphic objects are attached to a container object, they can be attached to another container object in another context. Additionally they cannot be deleted, even if explicitly deleted by commands like glDelete, as long as they are attached to any container object in any context. Memory is automatically reclaimed by the implementation, when all objects are deleted from all contexts and are no longer attached to a container object [MS17, Chapter 5.1]. These attributes are used to load isosurface graphic files in the loading thread, then make them available to the main thread for rendering and later to free the allocated memory from the graphic card, once they are no longer used by the main thread.

A simplified version of the thread interaction is illustrated as diagrams in figure 4.2 and figure 4.3 and a more detailed illustration can be found in the appendix. The diagrams are read top to bottom. On the left side activities in the main thread are shown, on the right side those of the second thread. In the middle different boxes are shown. The one named 'int buffer' illustrates, which array index of the created objects is used to load and render the isosurface graphic files. The 'GPU' box shows which graphics are available to which context. The first number indicates which array index is used in the main thread, if the number is in brackets, it is not used by any thread. This number is followed by either 'currentBuffer' indicating that the data is rendered in the main thread, or 'nextBuffer' indicating that data is loaded in the second thread. The next line shows which isosurface graphic files are visualized or loaded, followed by which context the data is attached to.

Normal black arrows are labelled and show which array index is used by which thread, when a thread is notified and which isosurfaces are currently rendered. A grey unlabelled arrow means that these isosurfaces could still be rendered by the main thread. A filled, black or green arrow is used, if all objects needed to render the corresponding isosurfaces are available to one context. A blue, unfilled arrow is used when shared object pointers are copied to the main thread and to indicate to which array index they are copied to. A black or grey dashed line arrow means that some, but not all, objects attached to these graphics were deleted. A grey dotted line is used to indicate, that the second thread is still in the same loop.

When the application is started, the shared context attribute is set and two contexts are created. The main thread then creates arrays of objects needed to visualize isosurfaces and loads the first n isosurfaces to the first array index of the objects created, where n is the amount of graphic files needed to visualize 20 time steps and the array index ranges from 0 to 2. Thereafter the second thread is initialized and the unused context is set to be its current context. To ensure that the second thread wont terminate, the loading process is inside a while loop.

At the beginning of each loop the objects needed to load the next n isosurfaces are created. The graphic files are then loaded and the shared pointers are copied to the corresponding objects in the main thread. To prevent unintentional side effects, while data is copied, a mutually exclusive flag is used. Thereafter some of the objects are deleted and the thread waits until the data loading to the graphic card is completed. A boolean flag 'threadReady' is then set for the main thread and then the second thread waits until notified.

In the main thread's rendering loop it is checked, if the second thread's ready flag is true and if another boolean flag, 'needsUpdate', used by the main thread to signal that new graphic files need to be loaded, is true. If this is the case, a container object is attached to the loaded graphic files and both flags are set to false.

When time is advanced, it is also checked if new isosurface graphic files need to be loaded. If this is the case, the 'needsUpdate' bool is set true and the second thread is notified. This bool also prevents the user from advancing too far in time, before new graphic files are loaded. Upon notification, the second thread deletes the last of its objects, thereby removing any attachments it had to the previously loaded graphic files and starts its loading loop again.

By keeping a container object attached to the loaded graphics in one context, another context can attach a container object to the same graphics. Deleting all objects created in the second thread, allows to create new ones at the beginning of each loop, without corrupting memory. Rebinding all objects, used to visualize isosurfaces, in the main thread frees unused memory [MS17, Chapter 5.1].

Loading graphics during runtime in a second thread increases the application's loading time by approximately 10%, which is to be expected when using multi threading in OpenGL.

The times shown in table 4.1 were measured with a stop watch. To average the loading time 5 of the 7 measurements were taken, the best and worst times were removed.

NOMAD VR's single thread implementation averaged 12.81 seconds, the multi thread implementation had an average loading time of 14.27 seconds for 60 time steps. Loading times for the first 20 time steps were also taken and averaged to 4.25 seconds for the single thread implementation, which suggests a linear correlation between loading time and the data loaded. Assuming linear loading time a single time step would need approximately 0.21 seconds.

The size of the isosurface files and the resulting high loading time make a multi threaded approach useful, even though the program loads slower than it would otherwise do, as it allows users to continue to explore the currently loaded data set while new data is loaded, instead of switching to a loading screen, breaking immersion.

Single Thread	Multi Threading			
	1-20	21-40	41-60	Sum
13.02	4.52	5.44	4.04	14.00
12.96	4.46	5.64	4.13	14.23
12.91	4.57	5.39	4.13	14.09
12.43	4.66	5.64	4.19	14.49
12.58	4.47	5.67	4.43	14.57
12.59	4.46	5.54	4.12	14.12
13.69	4.68	5.50	4.24	14.42

Table 4.1: Time in seconds needed to load the first 60 time steps of the visualization from a SSD with NOMAD VR and the multi threaded version of the application. The multi threaded approach loads isosurface files for 20 time steps, their individual loading times and their sum are listed.

4.5 Known Issues

To conclude this chapter, known issues of the prototype implementation are mentioned. These issues are mostly minor and don't affect the application, but should be mentioned nonetheless.

The extension was implemented under the assumption, that the trajectories of all particles are calculated at the beginning. If the user chooses not to calculate all trajectories and then selects a particle, which trajectory was not pre-calculated, an error occurs. This might need fixing, if the application is generalized for other visualizations.

When visualizing trajectories, it is recommended that the data set has the same amount of entries for each time step and that the same order, in which they are listed, is kept. If not, unwanted effects, like showing the wrong trajectories, can occur. This cannot be fixed, without changing how trajectories are calculated.

When the isosurfaces were interactively translated and the visualization was rescaled afterwards, the isosurfaces were no longer shown in the correct place. This can easily be fixed by resetting their position with the controller, at any time, and then translating them again, after changing the scaling.

After an update of SteamVR, the haptic feedback for particles worked incorrectly. After the visualization was rotated, the haptic feedback still activated at the original positions of the particles, instead of activating where they were visualized. This lead to the controller vibrating, where no particles were visualized.

Terminating the application, while isosurface graphic files are still loaded by the second thread, lead to a windows error message.

When the application is started for the first time and SteamVR was not started beforehand, a colourless, greyish graphic model of the headgear is rendered. This is fixed by restarting the application. In more current versions of NOMAD VR this bug is already fixed.

While testing the application the HTC Vive Pro system was installed and slight lags occurred during the last second, when new isosurfaces were loaded by the second thread. Reducing the amount of isosurfaces loaded by the second thread might solve this problem.

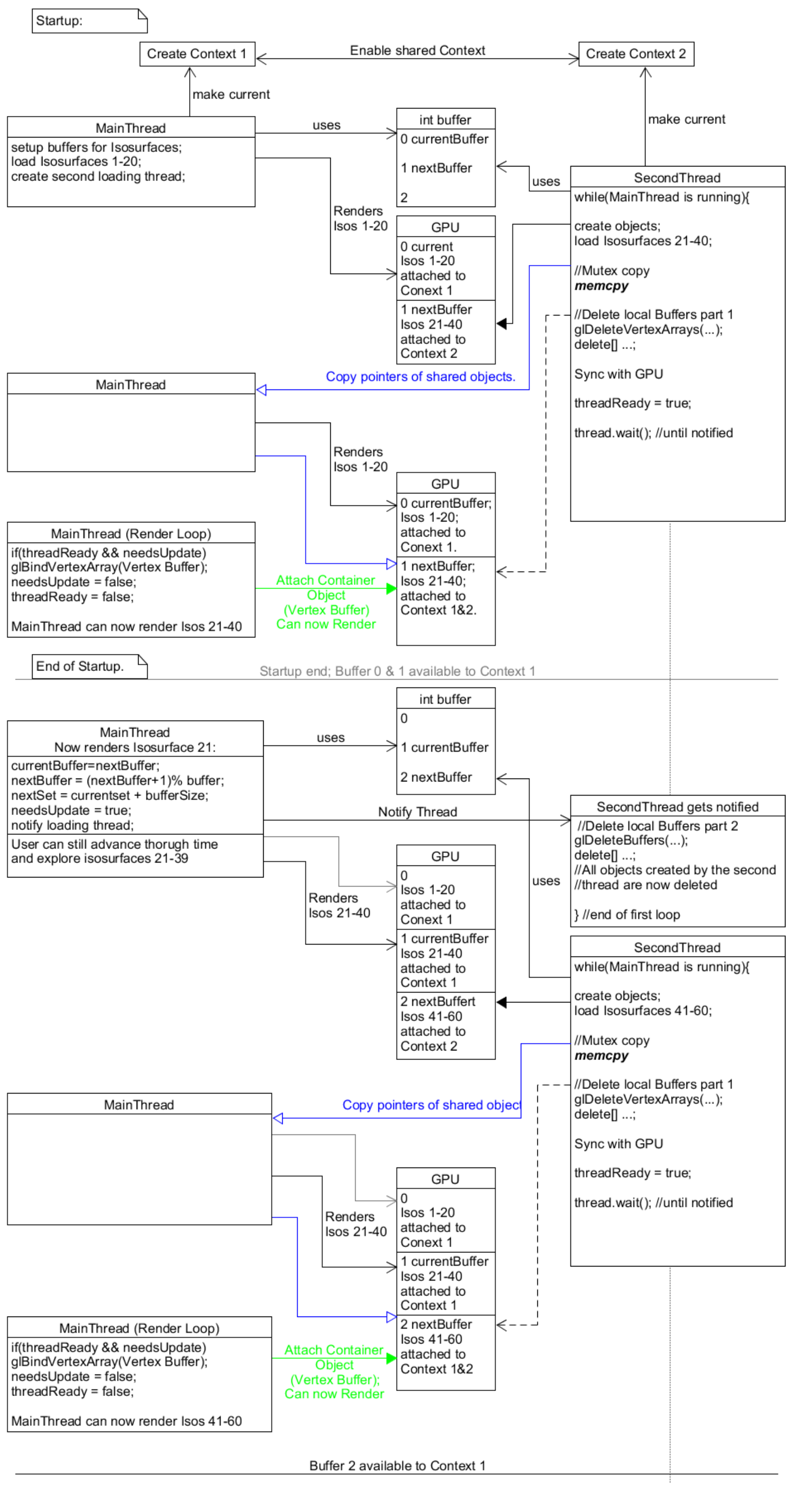

Startup:
Create Context 1
Enable shared Context
Create Context 2
make current
make current
MainThread
setup buffers for Isosurfaces;
load Isosurfaces 1-20;
create second loading thread;
uses
int buffer
0 currentBuffer
1 nextBuffer
2
uses
SecondThread
while(MainThread is running){
create objects;
load Isosurfaces 21-40;
//Mutex copy
memcpy
//Delete local Buffers part 1
glDeleteVertexArrays(...);
delete[] ...;
Sync with GPU
threadReady = true;
thread.wait(); //until notified
Renders Isos 1-20
GPU
0 current Isos 1-20 attached to Conext 1
1 nextBuffer Isos 21-40 attached to Context 2
Copy pointers of shared objects.
MainThread
Renders Isos 1-20
GPU
0 currentBuffer; Isos 1-20; attached to Context 1.
1 nextBuffer; Isos 21-40; attached to Context 1&2.
MainThread (Render Loop)
if(threadReady && needsUpdate)
glBindVertexArray(Vertex Buffer);
needsUpdate = false;
threadReady = false;
MainThread can now render Isos 21-40
Attach Container Object (Vertex Buffer) Can now Render
End of Startup.
Startup end; Buffer 0 & 1 available to Context 1
int buffer
0
1 currentBuffer
2 nextBuffer
MainThread
Now renders Isosurface 21:
currentBuffer=nextBuffer;
nextBuffer = (nextBuffer+1)% buffer;
nextSet = currentset + bufferSize;
needsUpdate = true;
notify loading thread;
User can still advance thorugh time and explore isosurfaces 21-39
uses
Notify Thread
SecondThread gets notified
//Delete local Buffers part 2
glDeleteBuffers(...);
delete[] ...;
//All objects created by the second
//thread are now deleted
} //end of first loop
uses
GPU
0 Isos 1-20 attached to Conext 1
1 currentBuffer Isos 21-40 attached to Context 1
2 nextBuffert Isos 41-60 attached to Context 2
Renders Isos 21-40
SecondThread
while(MainThread is running){
create objects;
load Isosurfaces 41-60;
//Mutex copy
memcpy
//Delete local Buffers part 1
glDeleteVertexArrays(...);
delete[] ...;
Sync with GPU
threadReady = true;
thread.wait(); //until notified
MainThread
Copy pointers of shared object
Renders Isos 21-40
GPU
0 Isos 1-20 attached to Conext 1
1 currentBuffer Isos 21-40 attached to Context 1
2 nextBuffer Isos 41-60 attached to Context 1&2
MainThread (Render Loop)
if(threadReady && needsUpdate)
glBindVertexArray(Vertex Buffer);
needsUpdate = false;
threadReady = false;
MainThread can now render Isos 41-60
Attach Container Object (Vertex Buffer); Can now Render
Buffer 2 available to Context 1

Figure 4.2: Diagram (part 1 of 2) illustrating the interaction between the loading and the main thread. Loading isosurface files for the first 60 time steps. (Created with UMLet 14.2 `www.umlet.com`.)

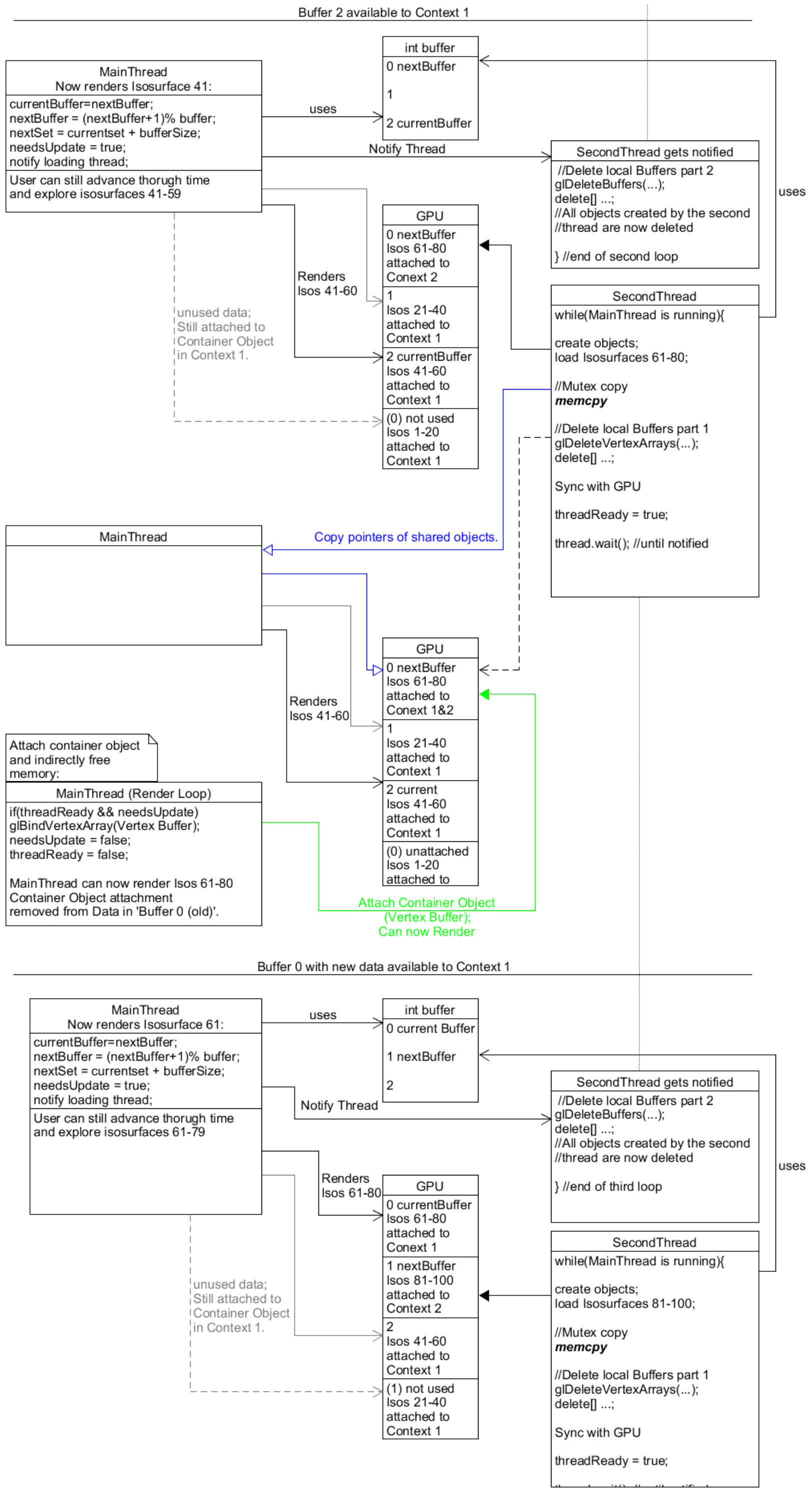


Figure 4.3: Diagram (part 2 of 2) illustrating the interaction between the loading and the main thread. Loading isosurface files for time steps 61 to 100. (Created with UMLet 14.2 `www.umlet.com`.)

5 Conclusion and Future Work

The prototype application, based on NOMAD VR, implemented in this thesis provides the means to visualize large scale turbulent fluid simulations. Trajectories and various vectors of each tracer particle are automatically calculated and visualized by the application. Different controller functionalities, displayed on top of the controller, allow to rescale the visualization, translate isosurfaces and to select single particles. After a particle is selected the amount of information shown can easily be adjusted, without restarting the application. To make spatial orientation easier for users, the spatial coordinates of a controller can be displayed. The way graphic files are loaded was also adjusted, to accommodate for the large size of the isosurface graphic files.

Together this gives scientists the ability to freely move and explore a three dimensional, interactive scientific turbulent fluid visualization, without long waiting periods while new data is loaded, which might lead to new insights. The applicability of the work flow described in this thesis and of the extended application itself was also confirmed, as data of a real turbulent fluid simulation was used.

The size of the full dataset no longer defines if it can be visualized by the application. The implemented asynchronous multi threaded loading approach works theoretically as long as the size of the few isosurface graphic files, which are loaded by the second thread, do not exceed one quarter of the graphic card memory. This allows the researcher to view large scale data sets in virtual reality without breaking immersion, thereby fulfilling the main goal of this thesis.

The time consuming procedure to visualize a particle's trajectory and various vectors, described in section 1.1, was already automatized in NOMAD VR. Ways to switch the observation between different single trajectories, without restarting the application and breaking immersion, were added. This allows for an easy exploration of the interaction of tracer particles with the vortex tubes in the simulation.

To aid scientists in the exploration, different interactive controller functionalities, indicated on a display hovering above the second controller, were added. Showing thousands of particles, vectors and trajectories at once is sometimes confusing. In order to directly control the amount of information shown during the visualization, additional controller options were implemented. Enabling users to change the scaling, or to translate isosurfaces with the controller, while the application is running, also helps to uphold immersion. To make spatial orientation easier an additional alpha version display was added, showing the controller's current coordinates. This additional display only shows when the first controller mode is selected.

The emphasis on and assumption, that greater immersion will help with a better understanding of the visualized data is based on [BM07] as,

> "It should not be surprising, then, that a higher level of immersion can lead to greater spatial understanding, which can result in greater effectiveness for many applications such as scientific visualization, design review, and virtual prototyping."

All display textures use graphic files with very few pixels. This was done to test if low quality graphics are a viable option to use for a display. It was found that the performance saving, low pixel digits and letters are very well readable, when rendered in the HTC Vive.

The large size of the isosurface graphic files motivated to try out loading them in a separate second thread. This enables the application to render already loaded data, while new data is simultaneously loaded. By doing so a slight increase of the loading time occurs, as was expected. The time increase is approximately 10%, the expected increase was between 10% and 30%.

The work's starting point was to reproduce the roller coaster experience of the two dimensional visualization. Through a moving camera perspective, closely following a selected particle, it was possible to see its interaction with the surrounding vortex tubes. During an early visit of one of the scientists it was found, that the inherent three dimensionality of virtual reality and the ability to move freely in the visualization make camera rotations not only obsolete, but a hindrance to the exploration. The two dimensional visualization showed four different isosurface values per time step. This was also tried out in the virtual reality application and it was found that the visualization of more than two isosurface values per time step lead to confusion and interfered with the exploration.

When comparing the images of the two dimensional visualization (figure 1.1) with those of the virtual reality application (figure 3.4), it might seem as if the latter has a much lower quality. This is only true for static images, where a rendering time of more than 20 minutes can be acceptable. When exploring the dataset dynamically in ParaView or VisIt, users need to use the same visualization plot as is used in the virtual reality application.

The application's advantages over a two dimensional visualization were already explained, but if the effort to prepare data for the virtual reality application is too great, it wont be used.

The process to visualize a few hundred particle trajectories in VisIt or ParaView takes multiple days of work, while creating a database for the virtual reality application, capable of showing thousands of trajectories, takes only a few hours and adjusting the scripts used to extract and convert the isosurface files can also be done within a few hours. Even if the process for the trajectory visualization was optimized, the virtual reality application still has the advantage of being three dimensional. Therefore it can be concluded that the data preparation for the virtual reality application is at least as fast as creating a two dimensional video.

The main goal of this thesis was to test if a turbulent fluid simulation can be visualized in virtual reality, which changes need to be made to the existing application and which additional functionalities are useful or needed when doing so.

Because of that different functionalities currently use fixed values, instead of being adjustable in the configuration file, before the application is started. To increase the general applicability of the application in the future the following options can be added as parameters changeable in the configuration file:

- Translation distance. A parameter that changes how far the isosurfaces are moved, when they are shifted with the controller.

- Velocity scaling. An option to adjust the scaling factor, by which the visualized velocity vector is multiplied with.

- Isosurfaces loaded. Changing how many isosurface are loaded by the second thread and when the application is started, giving the user more control over the maximum size of the isosurface graphic files.

To make the display, hovering above the second controller, visually more appealing, a rotation according to the controller's spacial orientation can be added, as well as how, where and when its coordinates are shown.

While the application can load new data and simultaneously be interacted with, the relative high loading time for the isosurfaces seems far from optimized. Three different ideas to optimize this came to mind. The first and most obvious is to try out a better graphic card. If the loading time isn't reduced, the hard drive might cause the high loading time. Therefore the second approach, to create a RAM drive and copy the data files onto it, might speed up the loading process. The third option is to create an additional loading thread and see what happens.

With the development of new virtual reality technology with higher resolutions, like the HTC Vive Pro, performance issues might arise. As each eye renders a slightly different picture, the application can utilize a second graphic card. By doing so one graphic card would render the left eye's screen and the other one the screen of the right eye.

6 Acknowledgements

At this point I'd like to thank everyone involved in this project.

First and foremost I'd like thank my first supervisor, Dr. Rubén Jesús García-Hernández (LRZ, BadW) and the LRZ, for granting me access to the NOMAD VR project and the HTC Vive Virtual Reality System. His expertise and supervision during this thesis made this project possible.

My gratitude also goes to Dr. Cristian Lalescu, Dr. Michael Wilczek (Max Planck Institute for Dynamics and Self-Organization), the creators of the turbulent fluid simulation used in this thesis, for granting me access to the dataset, helping me to define which controller options are useful for them. I also thank Dr. Cristian Lalescu for taking the time to visit us at the LRZ and test an early version of the extension.

My thanks also goes to my second supervisor, Dr. Markus Rampp, who also granted me access to the dataset and provided me with all the resources needed for the data extraction.

Appendix

Listing 1: Example configuration file used to start the extended version of NOMAD VR

```
#General Options
timesteps 200 #number of timesteps
userpos 0 0 0 #starting position
background 0 0 0 #black; r, g, b
scaling 1 #initial world scaling
hapticfeedback #controllers vibrate if held inside a particle
showcontrollers #remove this line to hide controllers
animationspeed 0.45 #changes the maximum speed, the user can
  #advance to the next timestep.

path "C:\path\to\data\"
  #This directory contains all isosurface files and the particle file

#Isosurfaces
isos 2 #number of isosurfaces shown per timestep
transparencyquality 3 #amount of transparency layers for isosurfaces.
values "110n" "125n" #name of *.ply files in 'path'.
  # "110n" if your isosurfaces are named: 1-110n.ply, 2-110n.ply ...
colours 0.35 0.63 0.95 0.6      0.9 0.2 0.2 0.8 #r g b opacity
  #each isosurface has its own colour and opacity setting.

#particles
newatom C 0.5 0.5 0.5 0.006865
  #creates a new or overwrites an existing atom's colour and radius

xyzfile "fullParticles1000.xyz" #name of particle file in 'path'
atomscaling 0.5 #changes the scaling of all particles.
showtrajectory 0 #int <= 0 shows trajectories of all particles
```

Listing 2: Python script to export isosurface graphic files with ParaView.

```
#### import the simple module from the paraview
from paraview.simple import *

BaseInputFilename='/path/to/files/input/filename_%04d.xmf'
BaseOutputFilename='path/to/files/output/'


timesteps= 1800
isovalues=[80.0,95.0,110.0,125.0]
isovaluesNames=["80n", "95n", "110n","125n"]


```

```
for iter in range (len(isovalues)):

  myi=isovalues[iter]
  myir=isovaluesNames[iter]


  for time in range(0,timesteps+1):

    #### disable automatic camera reset on 'Show'
    paraview.simple._DisableFirstRenderCameraReset()

    myfile=BaseInputFilename % time

    #create a new VisItBovReader
    #a1total_densitycube = OpenDataFile(myfile)
    a1total_densitycube = XDMFReader(FileNames=[myfile])
    #a1total_densitycube = Xdmf3ReaderS(FileName=[myfile])

    #set Mesh and Voritcity

    #a1total_densitycube.MeshStatus = ['Eulerian grid']
    a1total_densitycube.CellArrayStatus = ['vorticity_magnitude']
    a1total_densitycube.GridStatus = ['Eulerian grid']

    # set active source
    SetActiveSource(a1total_densitycube)

    # get active view
    renderView1 = GetActiveViewOrCreate('RenderView')
    # uncomment following to set a specific view size
    # renderView1.ViewSize = [1229, 934]


    # show data in view
    a1total_densitycubeDisplay = Show(a1total_densitycube, renderView1)

    # reset view to fit data
    renderView1.ResetCamera()
    renderView1.Update()


    #cellDataToPoint
    cellDatatoPointData1 =
              CellDatatoPointData(Input=a1total_densitycube)
    SetActiveSource(cellDatatoPointData1)

    # create a new 'Contour'
    contour1 = Contour(Input=cellDatatoPointData1)
    contour1.ContourBy = ['POINTS', 'vorticity_magnitude']
    contour1.Isosurfaces = [myi]
    contour1.PointMergeMethod = 'Uniform Binning'

    # show data in view
    contour1Display = Show(contour1, renderView1)
    renderView1.Update()

    # reset view to fit data
    renderView1.ResetCamera()
    print ("time ", time)
    i = time +1
    #print ("i ", i)
    myoutputfile=BaseOutputFilename+ str(i)+"-"+myir+".vrml"

    # export view
    ExportView(myoutputfile, view=renderView1)

    Delete(contour1)
    del contour1
    Delete(contour1Display)
    del contour1Display

    Delete(cellDatatoPointData1)
    del cellDatatoPointData1

    Delete(renderView1)
    del renderView1
    del myoutputfile
    Delete(a1total_densitycubeDisplay)
    del a1total_densitycubeDisplay
    Delete(a1total_densitycube)
    del a1total_densitycube
```

Listing 3: Python script to export particle graphic files with ParaView.

```
ExportParticles_nbdps.py
#### import the simple module from the paraview
from paraview.simple import *

# \\ instead of / when using a windows operting system.
BaseInputFilename='C:\\path\\to\\file\\input\\particles-step_%04d.xmf'
BaseOutputFilename='C:\\path\\to\\file\\output\\'


timesteps=1801
#isovalue = 50
isovaluesName = "particles"


#myi=isovalue
myir=isovaluesName

for time in range(0,timesteps,1):

  #### disable automatic camera reset on 'Show'
  paraview.simple._DisableFirstRenderCameraReset()

```

```

myfile=BaseInputFilename % time

# create a new VisItBovReader
a1total_densitycube = XDMFReader(FileNames=[myfile])

# set Array and Grid Status
a1total_densitycube.CellArrayStatus = []
a1total_densitycube.GridStatus = ['particles']

# set active source
SetActiveSource(a1total_densitycube)

# get active view
renderView1 = GetActiveViewOrCreate('RenderView')
# uncomment following to set a specific view size
# renderView1.ViewSize = [1229, 934]


# show data in view
a1total_densitycubeDisplay = Show(a1total_densitycube, renderView1)

# reset view to fit data
renderView1.ResetCamera()
renderView1.Update()


i = time + 1
print ("i ", i)

myoutputfile=BaseOutputFilename+ str(i)+"-"+myir+".vrml"

# export view
ExportView(myoutputfile, view=renderView1)


Delete(renderView1)
del renderView1
del myoutputfile
Delete(a1total_densitycubeDisplay)
del a1total_densitycubeDisplay
Delete(a1total_densitycube)
del a1total_densitycube
```

Listing 4: Shell script for windows used to process isosurfaces and to apply a filter with MeshLab

```
for /l %%x in (start, step, end) do (
   echo %%x
   "C:\Program Files\VCG\MeshLab\meshlabserver"
      -i %%x-80n.wrl -o %%x-80n.ply -m vn -s filter.mlx
)
```

Listing 5: Shell script for windows used to convert the particle files from *.wrl to *.xyz with MeshLab

```
for /l %%x in (1, 1, 1801) do (
    echo %%x
    "C:\Program Files\VCG\MeshLab\meshlabserver"
       -i %%x-particles.wrl -o %%x-particles.xyz
)
```

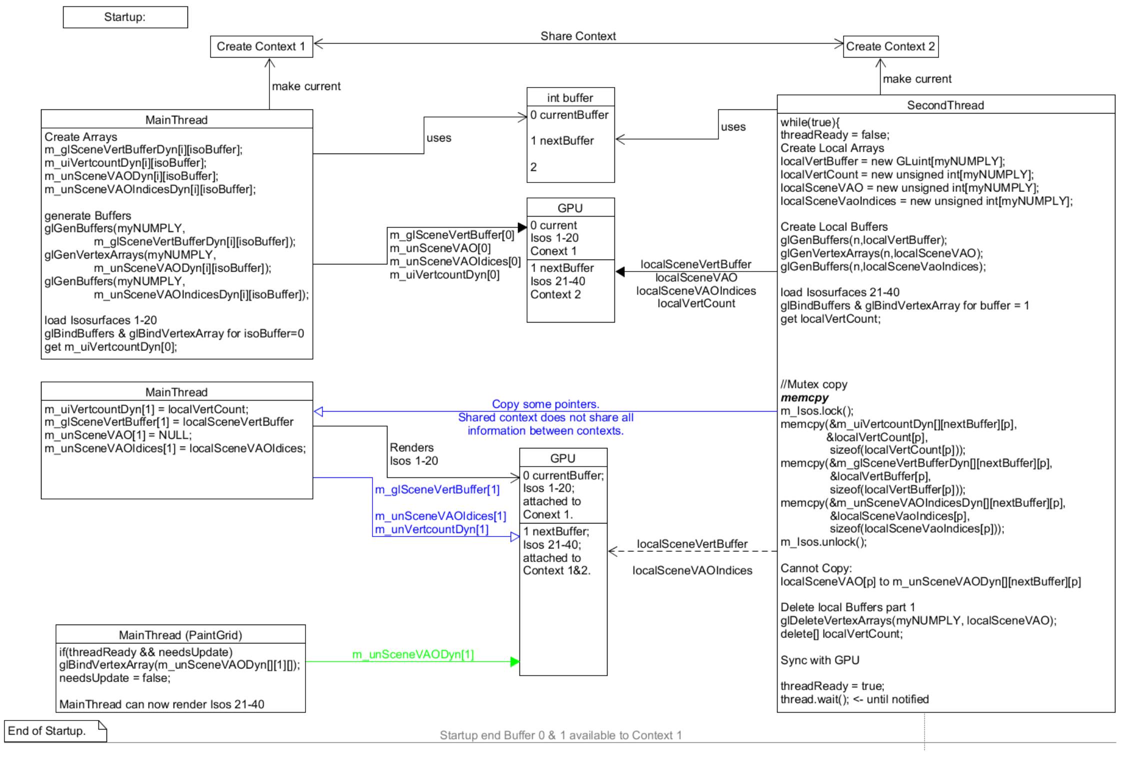
Listing 6: Python script to merge and convert particle graphic files.

```
#create new file
     fi = open(\"fullParticles.xyz\",\"w+\")
     print(\"Starting\")
     #Loop for all subsets
     for n in range(1,1801):
          filename = \"%d-particles.xyz\" %n
          fo = open(filename,\"r\")

          #Amount of particles \\n  timestep \\n
          # total 1000
          writeIntoFi = \"1000\\ni = %d\\n\" %(n-1)
          fi.write(writeIntoFi)

          print(\"Filename \",fo.name)
          #read i'th line from fo
          for i, line in enumerate(fo):
                    writeIntoFi = \"C \"+line
                    #write to fi
                    fi.write(writeIntoFi)
          #close old file
          fo.close()

     fi.close()
     print(\"done\")
```



MainThread

```
Create Arrays
m_glSceneVertBufferDyn[i][isoBuffer];
m_uiVertcountDyn[i][isoBuffer];
m_unSceneVAODyn[i][isoBuffer];
m_unSceneVAOIndicesDyn[i][isoBuffer];

generate Buffers
glGenBuffers(myNUMPLY,
            m_glSceneVertBufferDyn[i][isoBuffer]);
glGenVertexArrays(myNUMPLY,
            m_unSceneVAODyn[i][isoBuffer]);
glGenBuffers(myNUMPLY,
            m_unSceneVAOIndicesDyn[i][isoBuffer]);

load Isosurfaces 1-20
glBindBuffers & glBindVertexArray for isoBuffer=0
get m_uiVertcountDyn[0];
```

SecondThread

```
while(true){
threadReady = false;
Create Local Arrays
localVertBuffer = new GLuint[myNUMPLY];
localVertCount = new unsigned int[myNUMPLY];
localSceneVAO = new unsigned int[myNUMPLY];
localSceneVaoIndices = new unsigned int[myNUMPLY];

Create Local Buffers
glGenBuffers(n,localVertBuffer);
glGenVertexArrays(n,localSceneVAO);
glGenBuffers(n,localSceneVaoIndices);

load Isosurfaces 21-40
glBindBuffers & glBindVertexArray for buffer = 1
get localVertCount;

//Mutex copy
memcpy
m_Isos.lock();
memcpy(&m_uiVertcountDyn[][nextBuffer][p],
            &localVertCount[p],
             sizeof(localVertCount[p]));
memcpy(&m_glSceneVertBufferDyn[][nextBuffer][p],
             &localVertBuffer[p],
             sizeof(localVertBuffer[p]));
memcpy(&m_unSceneVAOIndicesDyn[][nextBuffer][p],
             &localSceneVaoIndices[p],
             sizeof(localSceneVaoIndices[p]));
m_Isos.unlock();

Cannot Copy:
localSceneVAO[p] to m_unSceneVAODyn[][nextBuffer][p]

Delete local Buffers part 1
glDeleteVertexArrays(myNUMPLY, localSceneVAO);
delete[] localVertCount;

Sync with GPU

threadReady = true;
thread.wait(); <- until notified
```

MainThread

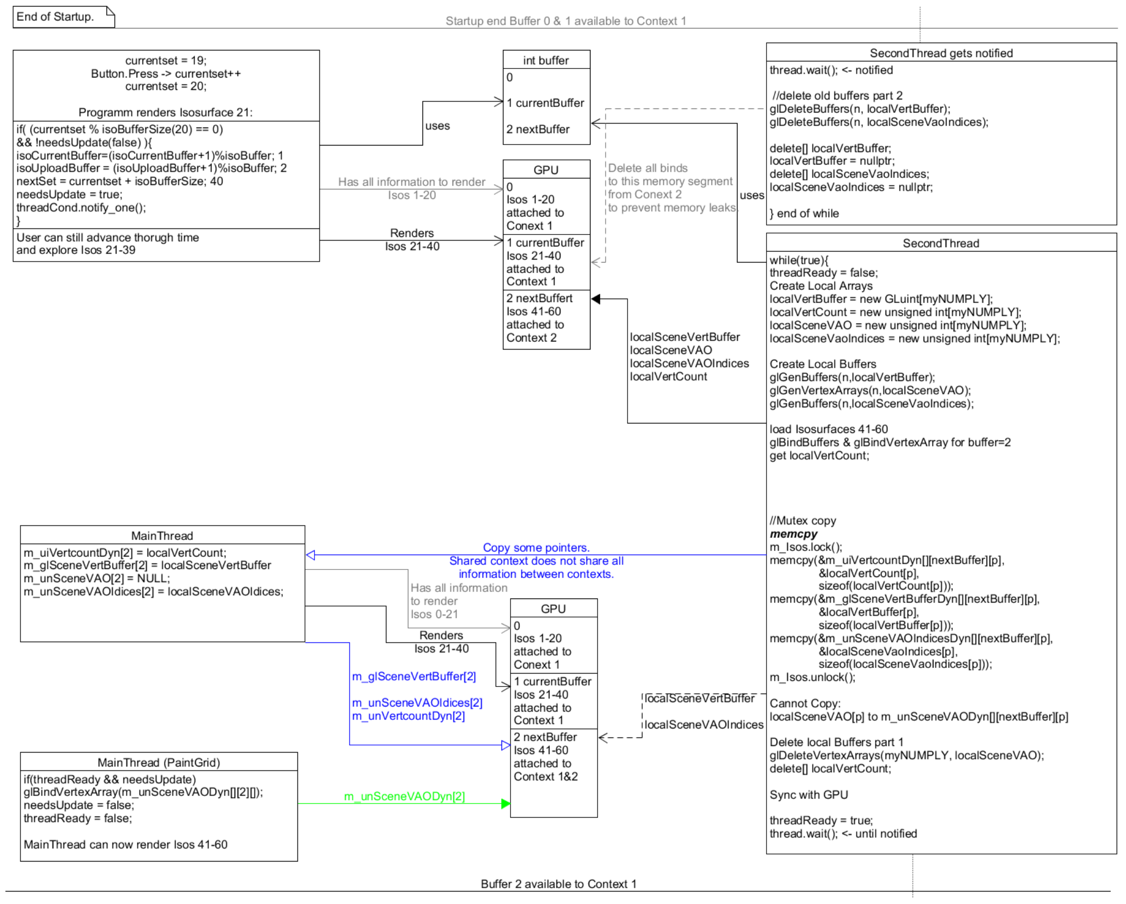
```
m_uiVertcountDyn[1] = localVertCount;
m_glSceneVertBuffer[1] = localSceneVertBuffer
m_unSceneVAO[1] = NULL;
m_unSceneVAOIdices[1] = localSceneVAOIdices;
```

MainThread (PaintGrid)

```
if(threadReady && needsUpdate)
glBindVertexArray(m_unSceneVAODyn[][1][]);
needsUpdate = false;

MainThread can now render Isos 21-40
```

Figure 1: More detailed diagram showing the thread interaction (part 1 of 4). (Created with UMLet 14.2 `www.umlet.com`.)



```
currentset = 19;
Button.Press -> currentset++
currentset = 20;

Programm renders Isosurface 21:
```

```
if( (currentset % isoBufferSize(20) == 0)
&& !needsUpdate(false) ){
isoCurrentBuffer=(isoCurrentBuffer+1)%isoBuffer; 1
isoUploadBuffer = (isoUploadBuffer+1)%isoBuffer; 2
nextSet = currentset + isoBufferSize; 40
needsUpdate = true;
threadCond.notify_one();
}
```

User can still advance thorugh time
and explore Isos 21-39

SecondThread gets notified

```
thread.wait(); <- notified

 //delete old buffers part 2
glDeleteBuffers(n, localVertBuffer);
glDeleteBuffers(n, localSceneVaoIndices);

delete[] localVertBuffer;
localVertBuffer = nullptr;
delete[] localSceneVaoIndices;
localSceneVaoIndices = nullptr;

} end of while
```

SecondThread

```
while(true){
threadReady = false;
Create Local Arrays
localVertBuffer = new GLuint[myNUMPLY];
localVertCount = new unsigned int[myNUMPLY];
localSceneVAO = new unsigned int[myNUMPLY];
localSceneVaoIndices = new unsigned int[myNUMPLY];

Create Local Buffers
glGenBuffers(n,localVertBuffer);
glGenVertexArrays(n,localSceneVAO);
glGenBuffers(n,localSceneVaoIndices);

load Isosurfaces 41-60
glBindBuffers & glBindVertexArray for buffer=2
get localVertCount;



//Mutex copy
memcpy
m_Isos.lock();
memcpy(&m_uiVertcountDyn[][nextBuffer][p],
            &localVertCount[p],
            sizeof(localVertCount[p]));
memcpy(&m_glSceneVertBufferDyn[][nextBuffer][p],
            &localVertBuffer[p],
            sizeof(localVertBuffer[p]));
memcpy(&m_unSceneVAOIndicesDyn[][nextBuffer][p],
            &localSceneVaoIndices[p],
            sizeof(localSceneVaoIndices[p]));
m_Isos.unlock();

Cannot Copy:
localSceneVAO[p] to m_unSceneVAODyn[][nextBuffer][p]

Delete local Buffers part 1
glDeleteVertexArrays(myNUMPLY, localSceneVAO);
delete[] localVertCount;

Sync with GPU

threadReady = true;
thread.wait(); <- until notified
```

MainThread

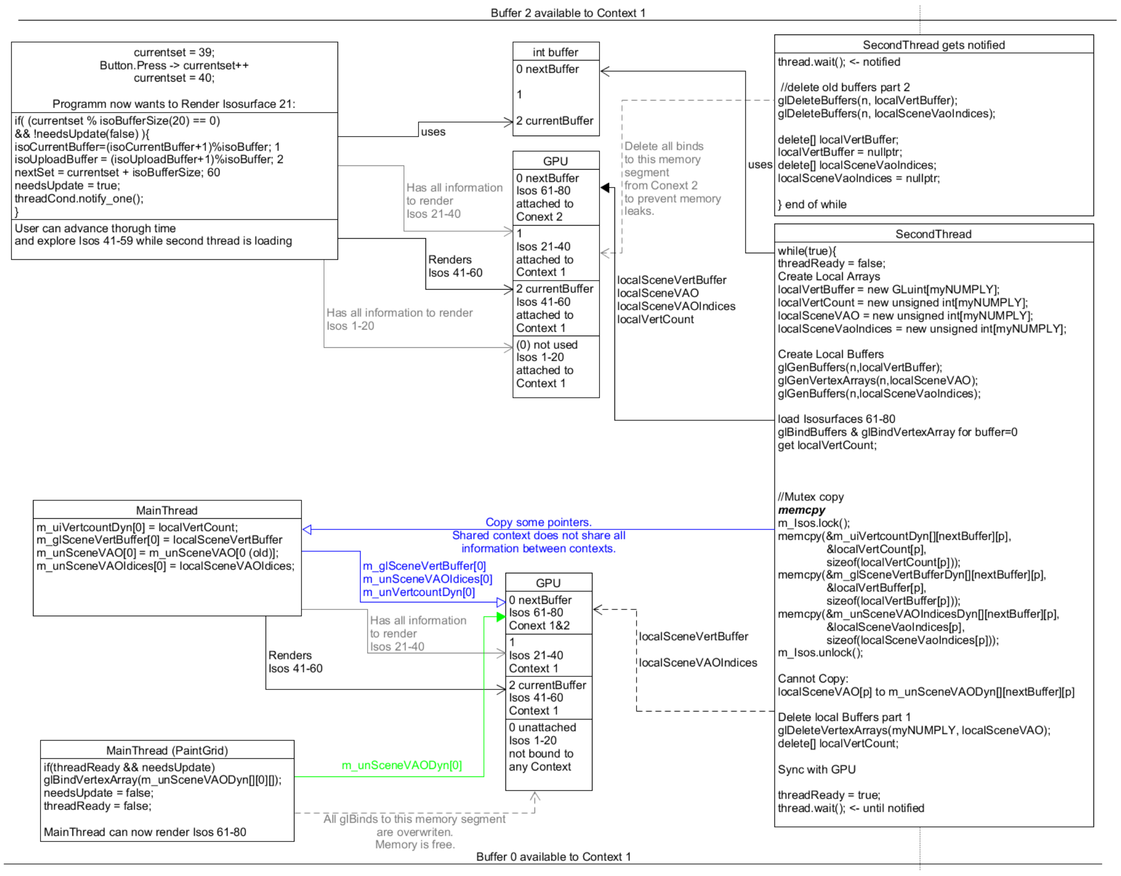
```
m_uiVertcountDyn[2] = localVertCount;
m_glSceneVertBuffer[2] = localSceneVertBuffer
m_unSceneVAO[2] = NULL;
m_unSceneVAOIdices[2] = localSceneVAOIdices;
```

MainThread (PaintGrid)

```
if(threadReady && needsUpdate)
glBindVertexArray(m_unSceneVAODyn[][2][]);
needsUpdate = false;
threadReady = false;

MainThread can now render Isos 41-60
```

Figure 2: More detailed diagram showing the thread interaction (part 2 of 4). (Created with UMLet 14.2 `www.umlet.com`.)



```
currentset = 39;
Button.Press -> currentset++
currentset = 40;

Programm now wants to Render Isosurface 21:
if( (currentset % isoBufferSize(20) == 0)
&& !needsUpdate(false) ){
isoCurrentBuffer=(isoCurrentBuffer+1)%isoBuffer; 1
isoUploadBuffer = (isoUploadBuffer+1)%isoBuffer; 2
nextSet = currentset + isoBufferSize; 60
needsUpdate = true;
threadCond.notify_one();
}
User can advance thorugh time
and explore Isos 41-59 while second thread is loading
```

```
SecondThread gets notified
thread.wait(); <- notified

 //delete old buffers part 2
glDeleteBuffers(n, localVertBuffer);
glDeleteBuffers(n, localSceneVaoIndices);

delete[] localVertBuffer;
localVertBuffer = nullptr;
delete[] localSceneVaoIndices;
localSceneVaoIndices = nullptr;

} end of while
```

```
SecondThread
while(true){
threadReady = false;
Create Local Arrays
localVertBuffer = new GLuint[myNUMPLY];
localVertCount = new unsigned int[myNUMPLY];
localSceneVAO = new unsigned int[myNUMPLY];
localSceneVaoIndices = new unsigned int[myNUMPLY];

Create Local Buffers
glGenBuffers(n,localVertBuffer);
glGenVertexArrays(n,localSceneVAO);
glGenBuffers(n,localSceneVaoIndices);

load Isosurfaces 61-80
glBindBuffers & glBindVertexArray for buffer=0
get localVertCount;


//Mutex copy
memcpy
m_Isos.lock();
memcpy(&m_uiVertcountDyn[][nextBuffer][p],
        &localVertCount[p],
        sizeof(localVertCount[p]));
memcpy(&m_glSceneVertBufferDyn[][nextBuffer][p],
        &localVertBuffer[p],
        sizeof(localVertBuffer[p]));
memcpy(&m_unSceneVAOIndicesDyn[][nextBuffer][p],
        &localSceneVaoIndices[p],
        sizeof(localSceneVaoIndices[p]));
m_Isos.unlock();

Cannot Copy:
localSceneVAO[p] to m_unSceneVAODyn[][nextBuffer][p]

Delete local Buffers part 1
glDeleteVertexArrays(myNUMPLY, localSceneVAO);
delete[] localVertCount;

Sync with GPU

threadReady = true;
thread.wait(); <- until notified
```

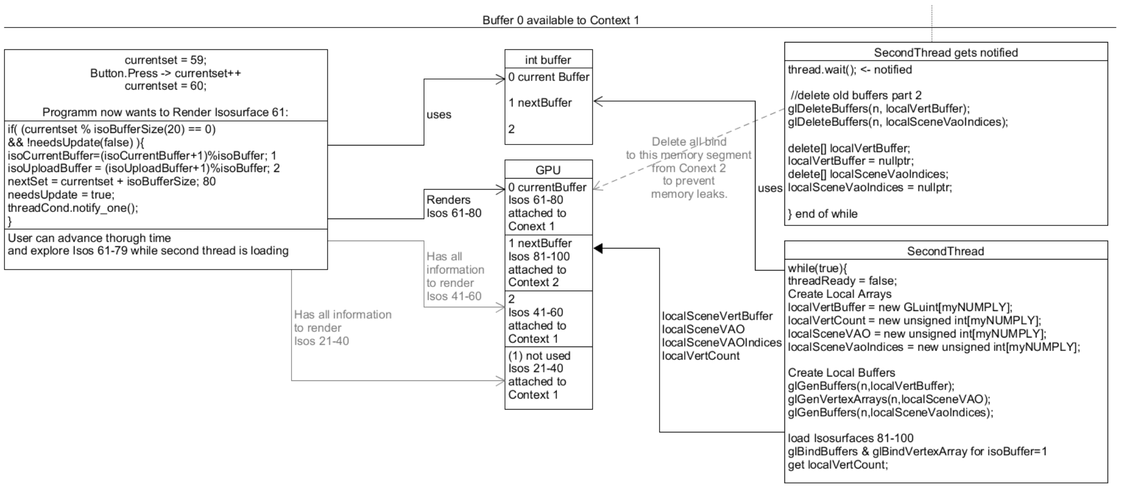
```
MainThread
m_uiVertcountDyn[0] = localVertCount;
m_glSceneVertBuffer[0] = localSceneVertBuffer
m_unSceneVAO[0] = m_unSceneVAO[0 (old)];
m_unSceneVAOIdices[0] = localSceneVAOIdices;
```

```
MainThread (PaintGrid)
if(threadReady && needsUpdate)
glBindVertexArray(m_unSceneVAODyn[][0][]);
needsUpdate = false;
threadReady = false;

MainThread can now render Isos 61-80
```

Buffer 2 available to Context 1
int buffer
0 nextBuffer
1
2 currentBuffer
uses
GPU
0 nextBuffer Isos 61-80 attached to Context 2
1 Isos 21-40 attached to Context 1
2 currentBuffer Isos 41-60 attached to Context 1
(0) not used Isos 1-20 attached to Context 1
Has all information to render Isos 21-40
Renders Isos 41-60
Has all information to render Isos 1-20
Delete all binds to this memory segment from Context 2 to prevent memory leaks.
uses
localSceneVertBuffer
localSceneVAO
localSceneVAOIndices
localVertCount
Copy some pointers.
Shared context does not share all information between contexts.
m_glSceneVertBuffer[0]
m_unSceneVAOIdices[0]
m_unVertcountDyn[0]
GPU
0 nextBuffer Isos 61-80 Context 1&2
1 Isos 21-40 Context 1
2 currentBuffer Isos 41-60 Context 1
0 unattached Isos 1-20 not bound to any Context
Has all information to render Isos 21-40
Renders Isos 41-60
localSceneVertBuffer
localSceneVAOIndices
m_unSceneVAODyn[0]
All glBinds to this memory segment are overwriten. Memory is free.
Buffer 0 available to Context 1

Figure 3: More detailed diagram showing the thread interaction (part 3 of 4). (Created with UMLet 14.2 www.umlet.com.)



```
currentset = 59;
Button.Press -> currentset++
currentset = 60;

Programm now wants to Render Isosurface 61:
if( (currentset % isoBufferSize(20) == 0)
&& !needsUpdate(false) ){
isoCurrentBuffer=(isoCurrentBuffer+1)%isoBuffer; 1
isoUploadBuffer = (isoUploadBuffer+1)%isoBuffer; 2
nextSet = currentset + isoBufferSize; 80
needsUpdate = true;
threadCond.notify_one();
}
User can advance thorugh time
and explore Isos 61-79 while second thread is loading
```

```
SecondThread gets notified
thread.wait(); <- notified

 //delete old buffers part 2
glDeleteBuffers(n, localVertBuffer);
glDeleteBuffers(n, localSceneVaoIndices);

delete[] localVertBuffer;
localVertBuffer = nullptr;
delete[] localSceneVaoIndices;
localSceneVaoIndices = nullptr;

} end of while
```

```
SecondThread
while(true){
threadReady = false;
Create Local Arrays
localVertBuffer = new GLuint[myNUMPLY];
localVertCount = new unsigned int[myNUMPLY];
localSceneVAO = new unsigned int[myNUMPLY];
localSceneVaoIndices = new unsigned int[myNUMPLY];

Create Local Buffers
glGenBuffers(n,localVertBuffer);
glGenVertexArrays(n,localSceneVAO);
glGenBuffers(n,localSceneVaoIndices);

load Isosurfaces 81-100
glBindBuffers & glBindVertexArray for isoBuffer=1
get localVertCount;
```

Buffer 0 available to Context 1
int buffer
0 current Buffer
1 nextBuffer
2
uses
GPU
0 currentBuffer Isos 61-80 attached to Context 1
1 nextBuffer Isos 81-100 attached to Context 2
2 Isos 41-60 attached to Context 1
(1) not used Isos 21-40 attached to Context 1
Renders Isos 61-80
Has all information to render Isos 41-60
Has all information to render Isos 21-40
Delete all bind to this memory segment from Context 2 to prevent memory leaks.
uses
localSceneVertBuffer
localSceneVAO
localSceneVAOIndices
localVertCount

Figure 4: More detailed diagram showing the thread interaction (part 4 of 4). (Created with UMLet 14.2 www.umlet.com.)

Figure 5: Dr. Cristian Lalescu (left), one of the creators of the turbulent fluid simulation used in this thesis, testing an early version of the virtual reality application at the LRZ.

List of Figures

Bibliography

[AJG+05] Ahrens ; James ; Geveci ; Berk ; Law ; Charles: *ParaView: An End-User Tool for Large Data Visualization.* Elsevier, 2005. – ISBN ISBN–13: 978–0123875822

[Aya16] Ayachit, Utkarsh ; Avila, Lisa (Hrsg.) ; Osterdahl, Katie (Hrsg.) ; McKenzie, Sandy (Hrsg.) ; Jordan, Steve (Hrsg.): *The ParaView Guide.* Kitware Inc., 2016

[BM07] Bowman, Doug A. ; McMahan, Ryan P.: *Virtual Reality: How Much Immersion Is Enough?* July 2007. – `http://www.cs.rug.nl/~roe/courses/OriInf/Bowman-Virtual-Reality.pdf` (Accessed 2018-08-04)

[CBW+12] Childs, Hank ; Brugger, Eric ; Whitlock, Brad ; Meredith, Jeremy ; Ahern, Sean ; Pugmire, David ; Biagas, Kathleen ; Miller, Mark ; Harrison, Cyrus ; Weber, Gunther H. ; Krishnan, Hari ; Fogal, Thomas ; Sanderson, Allen ; Garth, Christoph ; Bethel, E. W. ; Camp, David ; Rübel, Oliver ; Durant, Marc ; Favre, Jean M. ; Navrátil, Paul: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In: *High Performance Visualization–Enabling Extreme-Scale Scientific Insight.* 2012, S. 357–372

[CCC+08] Cignoni, Paolo ; Callieri, Marco ; Corsini, Massimiliano ; Dellepiane, Matteo ; Ganovelli, Fabio ; Ranzuglia, Guido: MeshLab: an Open-Source Mesh Processing Tool. In: Scarano, Vittorio (Hrsg.) ; Chiara, Rosario D. (Hrsg.) ; Erra, Ugo (Hrsg.): *Eurographics Italian Chapter Conference*, The Eurographics Association, January 2008. – ISBN 978–3–905673–68–5

[Cor16] Corporation, Valve: *Vive PRE User Guide.* 2016. – `https://www.htc.com/managed-assets/shared/desktop/vive/Vive_PRE_User_Guide.pdf`

[KRKP+16] Kluyver, Thomas ; Ragan-Kelley, Benjamin ; Pérez, Fernando ; Granger, Brian ; Bussonnier, Matthias ; Frederic, Jonathan ; Kelley, Kyle ; Hamrick, Jessica ; Grout, Jason ; Corlay, Sylvain ; Ivanov, Paul ; Avila, Damián ; Abdalla, Safia ; Willing, Carol: Jupyter Notebooks – a publishing format for reproducible computational workflows. In: Loizides, F. (Hrsg.) ; Schmidt, B. (Hrsg.) ; IOS Press (Veranst.): *Positioning and Power in Academic Publishing: Players, Agents and Agendas* IOS Press, 2016, S. 87 – 90

[MS17] Mark Segal, Kurt A.: *The OpenGL Graphics System: A Specification.* The Khronos Group Inc., 2017. – `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.compatibility.pdf` (Accessed 2018-08-22)

[NLL17] NIEHORSTER, Diederick C. ; LI, Li ; LAPPE, Markus: The Accuracy and Precision of Position and Orientation Tracking in the HTC Vive Virtual Reality System for Scientific Research. In: *i-PERECPTION* (2017). – http://journals.sagepub.com.emedien.ub.uni-muenchen.de/doi/full/10.1177/2041669517708205

[NOM15] NOMAD, H2020: *"The NOMAD Laboratory A European Centre of Excellence, homepage"*. 2015. – https://nomad-coe.eu/externals/european-centres-of-excellence (Accessed 2018-08-30)

[Ros95] ROSSUM, G. van: Python tutorial / Centrum voor Wiskunde en Informatica (CWI). Amsterdam, May 1995 (CS-R9526). – Forschungsbericht. – https://docs.python.org/3/faq/general.html#are-there-copyright-restrictions-on-the-use-of-python (Accessed 2018-08-30)

[Yan16] YANG, Yun: *The Frenet - Serret formulas of a discrete centroaffine curve.* January 2016. – https://www.researchgate.net/publication/301874077_The_Frenet-Serret_formulas_of_a_discrete_centroaffine_curve (Accessed 2018-09-25)