3DIS4U: Design and Implementation of a Distributed Visualization System with a Stereoscopic Display

Martin Ericsson
Abstract

3DIS4U: Design and Implementation of a Distributed Visualization System with a Stereoscopic Display

Martin Ericsson

Stereoscopic displays have been used in research as an aid for visualizations, but often they end up in a special room only to be used by a small selected audience. How should such a system be setup to make it more available to a larger group of users? We try to solve this by setting up the system in a regular lecture room, an environment already known by our users and by modifying software to make the transition from monoscopic displays to stereoscopic displays as smooth as possible. To improve the usability further, we choose to connect the stereoscopic installation to a high-performance computing (HPC) cluster. As a result, we offer our users to distribute their visualizations and by that the ability to use larger data sets.

There are two goals for this master thesis. The first goal is to setup a stereoscopic display in a regular class room environment. The second goal is to enable distributed visualization at our graphics lab and evaluate further development in this field. The first goal is accomplished by setting up the hardware and thereafter focus on making the system more usable. Three different ways will be presented, one by using the Visualization Toolkit (VTK), another by developing a small C++ library for converting existing visualizations to the stereoscopic display. And the final option is non-invasive stereoscopic visualization with the Chromium library. The second goal is realized by installing and configuring ParaView, a visualization application for distributed visualizations on a cluster connected to the stereoscopic display. Exploration of alternative ways of performing visualization on the Graphics Processing Unit (GPU) is also concluded.

The result of this master thesis work is primarily a lecture room that in a matter of a few minutes is turned into a visualization studio with a stereoscopic display for up to 30 simultaneous viewers. The result is also an extended version of VTK for our stereoscopic display, a C++ library meant to help users to port their program for stereoscopic visualization and some examples on how to use Chromium for noninvasive stereoscopic rendering. Furthermore, we have made ParaView available to HPC users by installing and configuring it on one of UPPMAX clusters.
## Contents

1 **Introduction**  
   1.1 The prerequisites ................................. 5  
   1.2 Visualization ..................................... 6  
   1.3 Distribution ..................................... 7  
   1.4 Stereoscopic display .............................. 7  
   1.5 Hardware ......................................... 8  
      1.5.1 Projectors ....................................... 8  
      1.5.2 Projector screen ............................... 9  
      1.5.3 Workstations ................................... 9  
      1.5.4 Cluster ........................................ 9  
   1.6 Test scenarios ................................... 9  
      1.6.1 Visualization of volume data .................. 9  
      1.6.2 Visualization of molecules .................... 11  

2 **Softwares** ..................................... 13  
   2.1 NVIDIAs control panel ............................ 13  
   2.2 The Visualization Toolkit .......................... 13  
   2.3 ParaView ........................................... 14  
   2.4 CMake ............................................. 14  
   2.5 Applications for comparison of GPU based visualization ....... 14  
      2.5.1 Raycasting on the GPU ....................... 15  
      2.5.2 Distributed rendering ....................... 15  

3 **Stereo projection** .............................. 17  
   3.1 Depth Cues ........................................ 17  
   3.2 Techniques for viewing stereoscopic images ............... 18  
      3.2.1 Auto stereoscopy (Spatial multiplexing) ........ 18  
      3.2.2 Stereoscopy ..................................... 19  
   3.3 Viewing analogy ................................... 23  
      3.3.1 Parallel projection ............................. 24  
      3.3.2 Dual display plane or toe-in ................... 24  
      3.3.3 Off-axis projection ............................. 25  
   3.4 Buffer techniques ................................. 27  
      3.4.1 Quad buffer ..................................... 27  
      3.4.2 Side-by-side .................................... 27  
   3.5 Artifacts ......................................... 27
Chapter 1

Introduction

The Centre for Image Analysis (CBA) at Uppsala University and the Swedish University of Agricultural Science have recently acquired equipment for building a stereo projection wall. The project has a close collaboration with Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) that will provide access to a high-performance computing (HPC) cluster for performing visualizations on. How do we set up this in such a way that it will be available to as many users as possible?

1.1 The prerequisites

To reach as many users as possible the new stereo wall was setup in a conference room already used for lectures and seminars. One important aspect with the availability of the system was that it should retain its old function as a regular seminar and lecture room, but also be able to be turned into a visualization studio in a matter of minutes. Hence a regular projector should coexist and function after installment of the stereo wall. The stereo projectors should only be used for showing stereographic content and will work as a compliment to the regular projector, not a replacement. The room can hold up to 30 people and our solution must support that many simultaneous viewers of the stereo graphical content that will be displayed on the wall. The room is 5 meters deep and 13 meters wide. The seats are set up in the rows with the middle row at 3.2 meters from the screen. Many stereo walls that are connected to a distributed system, a cluster, are made up of an array of projectors that together produce a very high resolution image not uncommonly several thousand times thousands pixels and many solutions to the problems that comes up with these kinds of systems are based around that. Our system consist of two projectors that mimics one screen instead and this makes our problems a bit different as we are working in much lower resolution than these multi projector walls. A view of the screen in the room can be seen in Figure 1.1.
1.2 Visualization

What is a visualization? Normally we think of it as a way to present results. When we have our results as numbers we want a nice picture to our report. But visualization can be more than that. If we start at looking at how we came up with our picture for our report we normally start with some phenomena or reality that we are interested in. We or someone else before us did, made a model of the phenomena that we then run a simulation of. The result from the simulation is then visualized to make it in a form that is easier to interpret and looks better. The pipeline can be seen in Figure 1.2.

![Reality ➔ Model ➔ Simulation ➔ Visualization](image)

Figure 1.2: One view of the visualization pipeline

The visualization step can itself be divided into three steps as can be seen in Figure 1.3. Getting the result from the simulation either from RAM or for larger data sets often by reading it from disk. The next step is to filter our data to a format that is more suitable for the last step, rendering. When we filter the data we also discard that part of the data set that we deem not adding anything to the visualization or enhance a special part of the data set for better visualization. Filtering in this case could also be called data preprocessing but a lot of the literature in this field use this notation so we will use that throughout this master thesis. The last step is the rendering step where we take the data and transform that into graphical data representing our filtered data set.

We have three main interests when creating visualizations.
Quality

Many scientific visualizations demand a high quality in the sense that the visualization should not add any information that is not really there. Quality also translates to performance as the more computational power that we have the larger data sets can we visualize and by that obtain higher quality.

Availability

Usability and availability is very important for the success of the stereo wall. Availability in that users should not need to be expert on stereographics to use the system nor expert programmers. The visualizations should also be runnable both on the stereo wall and on normal workstations for portability and ease of development.

Interactivity

We say that a visualization is interactive if the user can change the rendering at an interactive rate, for example, change the point of view, rotate a model or change the color. Interactivity is an important ingredient to make a visualization successful.

1.3 Distribution

Our view on performance is that it equals usability. If we can supply our users with a more performant system then they can visualize larger data sets and use our system to a larger extent i.e. it becomes more usable. To realize this we are using a cluster to do large scale visualizations. Both the filtering process and the rendering process can be distributed to the cluster if necessary. Some of the programs being set up on the cluster is also usable from other remote location so in this sense we are also increasing the availability of the system by distributing the resources in this way. We can by this option also open up new opportunities to our users by giving them the ability to visualize data sets that they where not able to do before. Some data sets are too resource demanding so that they cannot be visualized on a single workstation but must be distributed.

1.4 Stereoscopic display

Stereographical displays of various kinds have been used several times before in the research field. There are head mounted displays (HMD) where you wear a helmet like construction with two small screens, one for each eye. There exists true volumetric displays where a laser lit up certain points in a volume of smoke to render your model in mid air. The kind that we set up here is a single disp plane display. It
is meant to work as a normal projection screen but with a deeper sense of depth. Our system requires that the user wears a pair of light weight glasses, as seen in Figure 1.4, to experience the stereo effect.

![Figure 1.4: One of the three models of the glasses we use](image)

1.5 Hardware

We have used the following hardware to set up our stereo wall.

1.5.1 Projectors

The projectors used for our stereoscopic display are two InFocus projectors as seen in Figure 1.5. They have a native resolution of $1024 \times 768$ pixels. Each of them is equipped with a special spectral filter. In addition, there is a box in order to improve the quality of our stereographic display. The box can also be seen in Figure 1.5 on top of the projectors. The brightness of the projectors are listed as 3500 ANSI lumen.

![Figure 1.5: The two projectors used in our setup. Note the visible spectral difference between the filters in this picture and the “enhancing” box on top of the projectors](image)
1.5.2 Projector screen

We use a standard projector screen with dimension 256cm * 192cm which gives an aspect ratio of \( \frac{4}{3} \). One important quality of the projector screen is that it should function with a regular projector as well as the room is also used for regular lectures without the need for the stereographic display. The image is projected onto the screen from the front as most regular projectors do. Hence, this is not a back projection system where the screen is between the users and the projectors.

1.5.3 Workstations

The workstation used for developing and testing the majority of the applications is a dual-dual-core AMD Opteron 64-bit processor with 4 GB of RAM. Each of the four cores are running at 2.4GHz. It has two 1Gbit/s Ethernet network adapters connected to a cluster. The graphics card installed is a NVIDIA Quadro fx4600 with dual graphics output ports with 768MB of graphics memory. This is currently considered a high end graphics card. The second workstation that was used is a dual-quad-core Intel Xeon 64-bit processor with 16GB of RAM. It is running at 2.5GHz. It is also equipped with a mid range NVIDIA graphics card, Quadro FX 370. The two computers are connected on a LAN with a speed of 1GBit/s.

1.5.4 Cluster

The cluster we used for distributing our visualization is called Isis [7] and is at the time of writing the newest of the clusters at UPPMAX. It is a 200 node cluster where each node is a dual 64-bit AMD Opteron processor with dual cores, which makes four cores per node. In total there are 800 cores in the cluster and the memory of the nodes ranges from 4GB of memory to 16GB of memory. The theoretical peak performance of the cluster is 4 Tflops/s and the cluster uses a Gigabit Ethernet as an interconnect.

1.6 Test scenarios

Two specific scenarios have been used during the work discussed in this thesis. The problems that can occur when doing visualizations are very dependent on the application you are trying to visualize. We have explored two common scenarios that captures different problem settings in visualization, which is visualization of a volumetric data set and molecule visualization.

1.6.1 Visualization of volume data

Medical data is often represented as volumes of data, an array of so called voxels. These are made of slices of 2D images that are stacked on top of each other to form a volume. The problem that can arise when visualizing these are both that they are computational heavy to render and that they can be very large memory wise, often several hundreds of Megabytes. The rendering of these volumes often requires
us to fill a lot of pixels on the screen, the so called fillrate which can turn out to be a bottleneck. The larger memory requirement also put constraints on what kind of system we can visualize these on. A volume can be visualized in many different ways, some of them are described below. We treat data as completely static in our scenario and the more dynamic part of the visualization pipeline here is how we render the actual data, the last step in the visualization pipeline.

**Raycasting**

This method works by sending out rays from the virtual eye through the view plane and then traversing our volume with discrete steps. By reading the data at each step and comparing that with a so called transfer function we get information on what color we will get as a result. If we send at least on ray per pixel this method can render us an image as in Figure 1.6. We choose to work only with this specific method in our test scenario but we will describe three more for the sake of completeness.

![Raycasting Image](image)

*Figure 1.6: A CT data set rendered with raycasting in ParaView*

**Splatting**

Another way of rendering volumetric data is by splatting. Splatting works by that we precalculate all possible projections of a voxel onto the image plane. Upon rendering our volume, do look-ups in this table and writes the projection to the framebuffer so this will form our image.

**Isosurfaces**

A method that is more in line with what a graphics card traditionally was designed to is the isosurface method. This method has a prepass where we traverse the data set and produces triangles, something graphics cards are very good at rendering. How this data preprocessing takes place is dependent on the underlying algorithm, the most common one is called Marching cubes. The result from the filtering is a shell of the data set consisting of triangles. The fact that we loose the volumetric properties when we filter the data can sometimes be seen as a disadvantage.
Texture mapping techniques

A fourth way of rendering volumes is to create planes that are aligned to view vector of the virtual camera. These planes are then textured with slices from the volumetric data. If one creates and renders sufficiently many planes this will give the illusion of a volumetric model.

1.6.2 Visualization of molecules

Molecular dynamics visualization differs a lot from the field of visualizing medical volumes. A simulation is running and the data can be dynamic. The amount of data is often much smaller than in the medical visualization case, but in a distributed environment it might need to be sent more times over the network due to its dynamic nature. The atoms and how they are connected can be represented in several different ways. We have chosen what is called the ball and sticks method as in Figure 1.7. The atoms are represented by a sphere and the connections between them are represented as cylinders.

![Molecule Structure](image)

Figure 1.7: Example of a molecule structure rendered with ParaView
Chapter 2

Softwares

The operating systems that are installed on our two workstations are dual boot with Windows XP and Red Hat Linux Enterprise (RHLE) on the first machine and single boot RHLE on the second machine.

2.1 NVIDIA's control panel

To set the desired display for our projector setup we used the control panel bundled with the drivers for the NVIDIA graphics card installed in the workstations. The control panel software was used on the Windows setup both as the actual graphical application and in our own library for stereoscopic rendering where we load the dynamically linked library (DLL).

2.2 The Visualization Toolkit

![VTK Logotype](www.vtk.org)

Figure 2.1: VTK Logotype www.vtk.org [3]

The Visualization Toolkit (VTK) (Figure 2.1) is a large open-source framework for visualization developed by Kitware. It covers the whole visualization pipeline from acquiring the data to preprocessing and filtering and finally render it. The pipeline is implemented to be independent of traversal direction, one should always have the choice to go both forwards and backwards in the pipeline. VTK supports a lot of different data formats, and filtering algorithms. Some of the filter algorithms that we have used during this thesis work are histograms, marching cubes, clipping etc. The current version consists of several hundreds of classes made up of in total several hundreds of thousands lines of code. The source code is written in C++ and
there are bindings for Java, TCL and Python. Python is our binding of choice for VTK.

2.3 ParaView

![ParaView Logotype](www.paraview.org)

Figure 2.2: ParaView logotype www.paraview.org [6]

The Parallel Visualization Application or ParaView (Figure 2.2) is a visualization software also developed by Kitware. ParaView is built upon VTK with focus on visualization with larger scale than VTK is capable of. The structure is divided into a client/server mode. The ParaView client have a default easy to use Graphical User Interface (GUI) where one can create visualizations. The server is divided into two parts, one render server and one data server. The render server takes care of all rendering while the data server loads models and computes filtering of data.

2.4 CMake

To make the development process a bit easier we used the CMake application [2] which is directly supported by Kitware, makers of VTK and ParaView. CMake is an open source make system that is also cross platform so we used this both for Linux and Windows. CMake uses a platform independent and compiler independent script to produce native and compiler specific make files. In our case, we generated make files for Visual Studio for Windows and GCC for Linux. CMake was also developed by Kitware and has included the necessary scripts for both VTK and ParaView to shorten the time needed to setup the development environment for both these projects.

2.5 Applications for comparison of GPU based visualization

We also wrote some software to compare the render speed between VTK and custom build software that handle the rendering only with the GPU. VTK cannot render all types of data with the GPU and does in these cases resort to CPU-based rendering. The reason why we are interested in this comparison is that during the last decade the computational power of GPUs have exceeded the scaling of the CPUs. Many GPUs today are a lot faster than the common CPU, a modern GPU can have up to ten times the computational power, and by conforming our rendering algorithm
to reside on the GPU we hope to gain a lot of performance, which could instead be used on, for example, sophisticated interactivity in our real-time renderings.

2.5.1 Raycasting on the GPU

It has been shown before that the GPU can be programmed to perform raycasting at a remarkable speed. Our implementation is based from the description in [26] with influences from [25]. The GPU was programmed with the OpenGL API together with the OpenGL Shading Language (GLSL) and OpenGL Utility Toolkit (GLUT).

2.5.2 Distributed rendering

One down side to using GPUs for more general purpose computation, is in comparison to a CPU, that the amount of memory available can be rather small. A consumer based GPU have today in the range of 128MB to 1GB of memory and the non-consumer based models can be bought with up to at least 2GB of memory. This is in comparison to a normal PC that has memory that ranges from 1GB to at least 16GB. To get around the problem of relative low memory, we decided to use several GPUs connected in a network. We did not have any local resources to use for this, so our software was developed and tested in a public computer lab intended for under-graduate teaching equipped with consumer based PCs.
Chapter 3
Stereo projection

When we normally view three-dimensional (3D) graphics on a display we look at a two-dimensional (2D) surface, a computer screen, but we still get a notion of a third dimension, namely that is a sense of depth. What differs on a stereoscopic display? Why do we perceive much more depth on one of these devices?

3.1 Depth Cues

To start explaining the phenomena of depth perception we first look at why we perceive a flat standard computer screen as being able to display 3D models. Perception psychology defines [23] seven depth cues that gives us feedback on distance to object as well as inter-object relations. The cues have been known for a long time and the depth cues have been used in many different media to make us perceive objects with depth. The seven depth cues can briefly be described as

1. Relative size is one of the depth cues and the effect of this is that objects near us in the virtual world gets bigger and objects further away gets smaller which gives us a sense of depth.

2. An object that lies in the line of sight to another object obscure each other called occlusion also gives an inter object relation.

3. Visualizations with a built in lightning model that shades objects and cast shadows also gives a sense of curvature and inter object relations.

4. Difference in detail also tells us how far an objects is placed in our virtual world, objects with greater detail and texture gives us the cue that it is closer to our virtual camera than an object with less detail and texture.

5. Moving objects appear closer to us if they move at a higher speed than slower objects.

6. Another depth cue is perspective, two parallel lines convergence at the horizon.

7. And finally the height over the horizontal plane. Objects closer to the horizontal plane appears closer to us than objects further from the horizontal plane.
Three examples of this can be seen in Figure 3.1. All of these effects that help us perceive depth on a flat surface are called the secondary depth cues.

But what is the difference then to view a photo from actually looking at the world with your own eyes? A lot of the depth information lost in the transformation from a 3D world to a 2D surface is the fact that we have two eyes and the camera only have one lens. In addition to the secondary depth cues, we also define the primary depth cues, accommodation, convergence and retinal disparity. Accommodation is the amount of pressure that we apply to deform the eye lens so it refracts the light to focus on an object. This cue gets feedback regardless if you have one or two eyes. The other two primary depth cues requires that you have two functional eyes, namely convergence and retinal disparity. When we gaze at an object we need to rotate our eyes so the lines of sight cross at the point where we focus. The amount of rotation is also a depth cue for us as we can by this measure how far away we are looking. This cue is not biologically connected to the accommodation cue, but as we leave childhood we have learned to link these to behaviors together. This trained behavior has some implication when viewing stereoscopic displays. Having two eyes gives us two slightly different views of the world when we fuse these two views in our mind we get an additional depth cue, retinal disparity. The two primary depth cues that we are trying to mimic here to give a greater sense of depth is retinal disparity and convergence.

3.2 Techniques for viewing stereoscopic images

There are several techniques to limit the view intended for the left to the right eye and vice versa. This is crucial to obtain an appealing stereo effect. A few of these techniques can be combined to either enhance the stereopsis or to create support for several independent view points.

3.2.1 Auto stereoscopy (Spatial multiplexing)

The auto stereoscopy technique is a actually several different techniques based around the same idea. To spatially occlude a part of a display so that each eye gets a unique view of the display. This can for example be done with a setup of slits in front of a screen. The slits will occlude in this case each odd line for the left eye and each even line for the right eye if the viewer is located at the correct position. This particular setup limits the field of view and the horizontal resolution.
for the viewer. With more slits the field of view get wider with the trade-off is that resolution gets lower. As the field of view gets larger more viewer can use the display at the same time. Common for all the auto stereoscopy techniques is that they do not require any device, e.g., glasses to perceive the stereoptic effect. Another way of achieving the same effect as with the slits is to cover the display with small lenticular lenses. These lenses are cut in such a way that they refract the light differently depending on the angle that you view. When the user now changes position the lenses will refract the light in a different path so the user will see another part of the screen and get a greater sense of depth. Auto-stereoscopy is also called spatial multiplexing.

3.2.2 Stereoscopy

The following techniques are all based on that the viewer uses a pair of glasses to separate the left image from the right image.

![Figure 3.2: A snapshot of temporal multiplexing. The right eye is occluded and only the left eye can see the image.](image)

**Temporal multiplexing**

Temporal multiplexing is working in the time domain. You show the left eye an image on the display for a period of time while you cover the right eye, see Figure 3.2. Then you switch and cover the left eye while you display another image for the right eye, see Figure 3.3. If this is done rapidly enough then our mind will interpret this as a stream of simultaneous inputs to both eyes and will fuse these into one image, the stereopsis effect. Drawbacks of this is that it will cut the amount of updates in half as you are spending half of the time getting no input to one eye. Beneficial is that you sustain the complete resolution from the rendered image in comparison to the auto-stereoscopy techniques where you lower the resolution but can keep the framerate. The most common glasses for temporal multiplexing is a pair of active
shutter glasses where each lens is an LCD display which can block the light for one eye at a time. The glasses need to be synchronized with the display so one eye is blocked during the time when the image for the other eye is shown. This is an extra factor for this kind of setup to consider, because if the connection breaks, then the illusion of extra depth will break.

![Figure 3.3](image)

Figure 3.3: The left eye is occluded and only the right eye can see the image. Compared to Figure 3.2 there is a slight shift in the image rendered to simulate the interocular distance.

![Figure 3.4](image)

Figure 3.4: Two lenses with different polarization. Only light with matching polarization will pass through the lens.

**Polarized light**

Another way of multiplexing is by having two sources for your images and superimpose them. Then the two images are separated by shifting the polarization a bit differently for each source, see Figure 3.4. The image bounces off a screen back to the user that uses a pair of glasses with polarized glass, each one matching the polarization of the filter for each projector. If correctly calibrated this will occlude the left image from the right eye and vice versa, but the brightness of the image might be lower as a side effect from the polarization. The display in this case must have some
special qualities, it must preserve polarization, otherwise the images will leak over to the other eye and the stereo effect will be gone. The display is usually referred to as a silver screen and is a special treated surface that preserves polarization.

![Figure 3.5: Two images with different view points are split up into the primary color channels red, green and blue. The combined image is created by the blue and green channel from the top image and the red channel from the bottom. The output image will be seen as stereo if the viewer wears a pair of anaglyph glasses](image)

Anaglyph

The anaglyph technique works by chromatic multiplexing, i.e., color shifting. By wearing glasses that filter out non-overlapping colors, the stereo effect can be achieved. The left eye gets a color not visible to the right eye and vice versa. Most commonly the filters are red and green or red and cyan. The filtered image from the left and right source is then merged and displayed to the user. Here, only one image needs to be presented to the user as both views are encoded in the same image. Example can be seen in Figure 3.5. The merging of the image is done when created, e.g., rendered by the rendering software and then separated by the glasses. This
makes this technique usable in printing as well as on computer screens and regular television sets. The downside to this technique is that the color representation can be poor, due to that a lot of colors are filtered out.

Interference filter technology

The technique we chose for our stereoscopic visualization is the Interference filter technology (INFITEC)[1]. This technique is the newest among the ones described in this thesis work. The idea is based on spectral multiplexing where you block parts of the spectra for each eye to get the multiplexing. In Figure 3.6 we see a transmission graph of unfiltered light. It is an extension of the color anaglyph by not only dividing the light into cyan and red but into three different wavelengths for each eye. Figure 3.7 and Figure 3.8 shows this effect for the left and the right eye respectively. This gives the benefit that there is very little crosstalk between left and right eye and much better color representation. In comparison with the polarized light solution the INFITEC solution has one great benefit, you can tilt your head without loosing the depth perception. Doing this while using a polarized based system you will change the calibration of the glasses as they are built upon the assumption that the user has his head straight up. Tilting the head changes the polarization and there will be crosstalk between the left and right eyes. We have our workstation connected to an INFITEC box in turn connected to the two projectors. The process of filtering light occurs at three places in this system. The INFITEC box (see Figure 1.5) conforms the light toward the respective spectra that we need for the stereoscopic effect. Then there is a lens mounted on each of the projectors that filter the lights as described above. And the last step of the filter process is the glasses that the users wear. This is a passive system with no need for active shutters in the glasses.

![Figure 3.6: Sketch of unfiltered light from a example transmission.](image)
3.3 Viewing analogy

The most commonly used viewing analogy in computer graphics is that the viewer is located at the origin looking along the $z$-axis either in positive or negative direction depending on which graphics API you use. In multiviewer systems we need to support several viewing positions at the same time to get a completely accurate representation of the world. Our system is currently limited to one viewing position and we have focused on getting a good viewing analogy for that scenario. If we were to support multiple viewing positions then we need to be able to track the positions of our users and this does not fit inside our setup. In fact, we have not given multiple viewing positions any consideration as we feel that it would not be feasible with the current technology that we are using and with the large number of...
simultaneous users we have for our system.

### 3.3.1 Parallel projection

Parallel projection is when both the view point and the camera are separated by the virtual interocular distance. This will give us two unique views of our virtual world at the position of our virtual eyes but there exists some parts without overlap for example, where objects are visible to only one eye where they in reality would be visible to both. See Figure 3.9. The formula for setting the eye positions are

\[
\text{eye} = \text{eye} + \text{right} \times d
\]

\[
\text{at} = \text{at} + \text{right} \times d
\]

where \(\text{eye}\) is the camera position, \(\text{right}\) is the right vector in the local coordinate system for the camera and \(\text{at}\) is the direction vector in this coordinate system. The variable \(d\) is half of the interocular distance with a sign depending on whether we are setting up the left or the right eye.

![Figure 3.9: Parallel projection, two frustum with parallel line of sight separated by a short distance to render two slightly different views for stereo projection.](image)

### 3.3.2 Dual display plane or toe-in

Another way of creating two separate viewing frustum is by keeping the look at point fixed and to separate the virtual cameras. This will give us a projection where we do not have a single plane to project our image onto but one for each eye. This
can in some cases produce artifacts in systems like our where we only have one display plane. Similar to the case with parallel projections we here displace the virtual camera along the right vector in the local camera coordinate system by a factor of half of the interocular distance with the sign depending on which eye we are currently setting up and keeping the at point fixed, see Figure 3.10

\[
\begin{align*}
\text{eye} &= \text{eye} + \text{right} \times d \\
\text{at} &= \text{at}
\end{align*}
\]

Figure 3.10: Dual display plane also called toe-in projection. Similar to parallel projection (See figure 3.9) both eyes are separated by a small distance but here also rotated so both line of sights cross at the look-at point $P$.

### 3.3.3 Off-axis projection

The dual display plane viewing analogy does not transfer that well to the single plane display models i.e. the screen that we setup. The screen in this case does not have two viewing planes, only one so we need the line of sight from the left eye to be parallel with the line of sight with the right eye. But by just putting them in parallel we get section where we do not get stereopsis because the viewing frustums do not intersect everywhere. To make this completely correct we need a true off-axis projection. What we do now instead is forming two parallel view frustums that we skew so they both cover the whole screen, see Figure 3.11. To get this we need to change the projection matrix of the graphics pipeline as this normally is not
implemented in the basic APIs. We base our derivation here on the OpenGL [14] pipeline but this should be generally applied to all graphics pipelines. The standard OpenGL projection matrix $R$ is defined as

$$
R = \begin{bmatrix}
2n & 0 & \frac{r+l}{f+n} & 0 \\
0 & \frac{2n}{l-b} & \frac{r+l}{f+n} & 0 \\
0 & 0 & -\frac{2fn}{f-n} & -1 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
$$

where $n$ and $f$ are the distances to the near and far clipping planes and $t$ and $b$ are the top and bottom and finally $l$ and $r$ are the left and right. To go from this to an off-axis projection, we need to shift the eye position and the left and right variables according to the inter ocular distance and our focal point. We now introduce $f'$ that is defined to be $0.5 \times \frac{n}{\text{focalDistance}}$ which is then added or subtracted from $l$ and $r$ depending on which eye we currently render. This results in the matrix $R'$

$$
R' = \begin{bmatrix}
\frac{2n}{r-l+2f'} & 0 & \frac{r+l+2f'}{f+n} & 0 \\
0 & \frac{2n}{l-b} & \frac{r+l+2f'}{f+n} & 0 \\
0 & 0 & -\frac{2fn}{f-n} & -1 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
$$

Figure 3.11: Off-axis projection keeps both lines of sight parallel but skew the view frustum so they cover the same view plane.
3.4 Buffer techniques

There are two major buffer techniques used when rendering stereo which are also tightly connected to which kind of stereo display technique used. If you are using active stereo, time multiplexing then you normally use quad buffer and if you are using passive stereo e.g. polarized light or spectral multiplexing then you use the side-by-side buffer technique. Techniques like anaglyph does not need to use any special modifications as it can be rendered with the standard double buffered technique.

3.4.1 Quad buffer

The quad buffer techniques works with four buffers at the same time. You have two front buffer, one for the left view and one for the right view and then same for the two back buffers. When rendering one of the front buffers are displayed while the opposite back buffer is being rendered. When after the buffers are swapped the rendering begin at the other back buffer and the opposite from before front buffer is being displayed. The quad buffer technique needs some way of knowing the the switch is going to be made and this needs to be in synchronization with the active device that are going to display the content of the buffers. Most consumer based graphics card does not support the quad buffer technique and you need to buy the more expensive professional based card to get support for this.

3.4.2 Side-by-side

For passive stereo it is sufficient to have images of both point of views in the same buffer, called side-by-side view. The buffers is a regular double buffer with one front and one back buffer but they are twice as wide as the desired resolution. If these images are being sent to a graphics card with dual graphics port the graphics driver will take care of splitting the image and sending the left part to one projector and the right part to the other projector. All that is needed here is that you have a graphics card with two graphics ports and that you can allocate a twice as wide framebuffer. This is commonly supported on modern consumer based hardware which in most cases are cheaper than the professional based models. One issue with this technique is that you can get artifacts due to the fact that you are simulating two buffers but you only have one. Left side of the screen will be seen by left eye and right side will be seen by the right eye. The rendering in Figure 3.16 will be perceive as one foot to a user of a stereoscopic display and not two as seen in print in this thesis.

3.5 Artifacts

There are a few artifact that can occur when you are using a stereoscopic display. All of these causes strain on the user and they should be as minimal as possible if the system is intended to be used over a longer period of time. One major issue is ghosting. This is when the information meant for the right eye leaks into the left eye. This can be due to for example overlapping color space in anaglyph
method, temporal overlap in temporal multiplexing. Flicker is related to this when the user get input in between the different frame in a time multiplexed system. Accurate synchronization when the image should flip is needed to avert this problem. Another source of strain on the user is accommodation cues. If the physical surface and the projected surface is different then the user will for example be looking at a object 5 meters away in the simulation but the screen is 3 meters away. The eye then accommodate to a distance of 3 meters but we view the visualization as being 5 meters away causing strain of the ocular system. The behavior of linking accommodation and convergence also comes to play here. As mentioned earlier these two depth cues are not linked biologically but most of us learn to link them as a child. We know how much we are suppose to accommodate when we converge a certain amount and vice versa. This then becomes a problem when trying to view stereo optical content on a single display plane as we have as described earlier artifact between the accommodation and convergence depth cue. Another problem that can occur in stereoscopic visualizations is low frame rate. Techniques that shows information to one eye at a time cuts the frame rate in half. At low frame rate we will start to put strain on the user due to jerky pictures. In a similar way some auto stereoscopic displays have lower resolution than a normal display which can also cause strain in the long run. Tied to this is also low brightness of these displays. Last there is also the fact that most applications assumes that the viewer is at a fixed position in front of the display. If we view the same display a bit from the side we will get a skewed image. This error in projection will also cause strain to the user but can be solved by using another type of projection, off-axis projection, in conjunction with a positioning tracker.
3.6 View spaces

Objects in the virtual space can be positioned in three different ways related to the screen, at the same position, in front of the screen and behind the screen. The scenario when we have the object at the same position as the screen we say that is at zero parallax and poses no problem to us. That is how objects normally are rendered. Both projections of the object will be the same on both the left and the right eye in this case (Figure 3.13). Objects that are behind the screen relative to the viewer are said to be at positive parallax. A picture of this can be seen in Figure 3.14 and here we see that the projection of the objects at left and right eye ends up a bit shifted in comparison to each other. The shift gives the convergence cue to us and we perceive the object as being expanded in air inside the screen. At the far plane the amount of shift should be at max so great that our sight lines are parallel to each other, that the biological limit for rotation of the human eye, we cannot diverge with our eyes. The last scenario where the object is in front of the screen is the most troublesome one but also viewed as the most spectacular. When objects really pops out of a screen and when you get the feeling that you need to try to grab the virtual object, that is the definition of stereo graphics for a lot of people. As can be seen in Figure 3.15 the projection on the left and right image have switched position if compared to the case when the object was behind the screen. As the objects gets closer and closer to the viewer the user must skew more and more with his eyes. When the object gets too close the user see two images that is too much difference in them and the retinal disparity, fusing of the two images to
one, fails and the illusion breaks down. Having objects at negative parallax as this is called, can give very good feedback but can also put strain on the user if used for a longer period of time.

3.7 VTK and stereoscopic displays

VTK have native support for stereoscopic display like crystal view stereo, anaglyph and a few more but not side-by-side rendering and this was something that we needed for our project. Another thing that we missed was the native support for off-axis projection. During the development phase of these features we also discovered the actual annoyance of working with a stereoscopic side-by-side display when working with the underlaying operating system. It is hard to navigate due to the displacement of the mouse pointer. Before you have established on which window your mouse cursor is we need to move it to the edge of the screen to see if it wraps or not. The same thing occurs when you start a new application. There is no cue to whether the application is on the left or right side of the desktop. To get around this we tried to use NVIDIA’s API for controlling the graphics driver directly. So when we enter stereographic mode in VTK, we automatically switch to side-by-side view in VTK. Then when we leave it we restore clone mode that the computer was running before we entered stereographic display mode. Clone mode have a one to one correspondence when using two projectors so there cannot be any confusion on where the mouse pointer actually are in this case. This modification
of the automatic switch of display modes will hopefully make the workstation a bit more usable when browsing folders and file when you are not viewing any stereographic visualizations. One benefit with side by side view is that it is very easy to keep your application in hardware accelerated mode where all rendering occurs on the GPU if possible. Looking at the source code for VTK and how specifically at the other stereographic modes are handled this seems not always the case. Some of them copies the framebuffer to RAM and do the calculation with the CPU instead. When the calculations is complete the framebuffer is once again uploaded to the GPU for displaying. If one would use one of the other stereo optic modes in VTK we recommend that you check the source code to see the solution that is there and if one can modify it to stay on the GPU for better performance.

3.7.1 Pointer problem

To render the standard mouse pointer with a stereoscopic display can with some techniques cause artifacts. The mouse pointer is rendered in the image plane, as it is a 2D image with zero parallax. In the case of side-by-side rendering we also have the problem that the pointer is only rendered once on the screen which transfers to that we only see it with one eye. This gives it a transparent look and it is perceived by some people as very bothersome. When we move the pointer over an object with positive parallax it is rendered over it as the depth impression tells us. But in the other case where we have an object with negative parallax then the pointer will be rendered over this object even though the depth cues tells us that it should be behind
the object and this brakes the illusion of depth in the image. One solution to this problem is to turn off the rendering of the pointer. That works well in applications where you do not care where you click e.g. where you rotate the scene by click and drag. But in other cases where you need a pointer as a more precise input device then this becomes rather bothersome as you often in these cases need to click on a 3D object i.e. the scenario where the artifact is noticeable. A solution to this is to not render the standard window mouse pointer and instead render a 3D widget in VTK. This would solve the problem with the ghosting due to it only being rendered once and now it would have a depth value and be more consistent with the depth cues from the stereographic projection. We did not aim for a true 3D interaction extension for VTK, this is beyond the scope of this thesis. Some good ideas on this subject can be found in [27] “A multimodal Virtual Reality Interface for 3D Interaction with VTK” by Kok and van Liere. There they have among other things, developed a 3D interaction extension for VTK. What we did was just placing our VTK rendered pointer in the image i.e. at the same position that the regular window would have been. We also implemented the same solution in our framework for porting regular applications to our stereo wall for test purposes. A screen shot from the experiment can be seen in Figure 3.16. Side by side mode have similar problems with GUI toolkits that uses the regular windows system to render its widgets. The problem stems from that the window system does not know that this is suppose to be two windows here so a more complete solution to this would be implement a library that makes two copies of all GUI widget and translates them so they end up on the same position for both eyes. This was something that we considered but never implemented. This solution would also open up to window mode solution with side by side rendering which is not possible with out this kind of special solution.

![Side-by-side rendering](image)

Figure 3.16: Side-by-side rendering. Left and right view have different color for clarity. The pointer from the experiment can be seen as a pink sphere in the picture. Viewed on the stereo wall the pointer would be perceived as being placed in the image plane and would interact with the virtual scene as expected.
3.8 Non-invasive stereographic rendering

Another way of making an application displaying stereographic imaging without the need of modifying the source code is to use a non-invasive method. This is suitable for example when you do not have access to the source code. Non-invasive modification of a graphical application can be seen as an extra layer between the application and the graphics hardware. The software library intercepts calls to the graphics card and manipulates them before sending them to the graphics card. No change to the application in question should be needed, we only manipulate the output from the original program.

3.8.1 Chromium usage

We have used the Chromium library developed at Stanford University [5] as a non-invasive method. It is a second generation open source library that is built upon a previous project called WireGL. In our case, Chromium is a layer between VTK, or any other application that we need a stereo output from, and the OpenGL API. Our script takes the OpenGL stream of commands and we create a new full screen window where we do side-by-side rendering. We do this by creating two viewports also called tiles in Chromium and we perform the displacement of the virtual cameras that are needed to achieve the stereoptic effect. Chromium has three main parts that we need to use to get our program to stereo. First which one closest to our application is the Chromium application faker, tricks the application to send the graphics stream to Chromium instead of the standard OpenGL library. The application faker talks to the Chromium server for manipulation of the graphics stream and the third part, the Chromium mothership works as a bootstrap and controller for the Chromium session. To run an application three simple steps needs to be done.

1. Run the Chromium mothership with your script as input and the application that you are interested in running on the stereoscopic display. The Chromium mothership is a python program and is as such started with a python interpreter e.g.
   >python stereoscopic.conf a.out &

2. Next, we need to run one or more Chromium servers, the number depending on what our Chromium script is suppose to do. It is a standalone application simple run by the command
   >crserver &

3. And finally we need to run the Chromium application faker, which will start our program and display the result.
   >crappfaker

This will create two windows, one empty that the application originally created and one Chromium window which renders the modified application. If this is being run with the test script provided in the appendix there will be a full screen window that render the application in side-by-side mode.
3.8.2 Chromium pitfalls

There are a few pitfalls when working with graphic library interceptors. The data stream is built only by the actual data sent to the graphics card. A lot of high performance software uses software algorithms to cull information that never will be used by the graphics card. The culling is based on the position of the viewer and this can be a problem when we are trying to intercept the data stream and change the point of view. When we shift the camera position to mimic two eyes viewing the world instead, we might need data that the application discarded earlier in the pipeline and never sent to the graphics card. This will yield holes in our virtual world. Another problem also related to how the original application behaves is what kind of transformations it performs. The interceptor need to be able to parse the intended camera position from the stream of graphics commands. Some applications use transformations in such a way that it is impossible for an interceptor to calculate where the original point of view resides and these applications cannot be modified to render stereographical images with a non-invasive technique. The non-invasive technique will also incur a performance penalty. The CPU needs to put clock cycles into keeping track of the graphical data stream and to analyze it at, preferably, an interactive rate. One might be tricked into believing that there is no development time to modify an application with a non-invasive method, but that is not really true. One needs to per application setup a script or modify the interceptor source code so that it manipulates the data stream in a desirable way which can take up a varying amount of time.
Cluster based visualization

When we want to visualize larger data sets that are more computational expensive than we have local resources for we have two choices. Either we down-sample our data set and by that loose information and quality, or we can distribute the computation and use a remote and more powerful computational resource. What we first need to decide is what should we distribute? The filtering or the rendering or maybe both? We also need to keep track of where the data is stored and where we need to send it. Other questions are how should we render the data, by software or by hardware, and how should we distribute this? If we are running a visualization of a magnitude that it needs to be distributed, then how do we render it when we have done the filtering? If we need to distribute the computation for filtering then we can assume that the data set is so large that it cannot be rendered on one node. How do you divide the work that needs to be done? There are two major ways of doing this, sort the data before you render or sort the data after you render.

4.1 General programming of clusters

The most important thing to gain performance when working with a multiprocessor or multicore system is of course to parallelize your program. A measurement for an upper bound on the amount of performance gain that one can get by parallelization is called Amdahl’s law [28]. It says that the upper bound of the performance factor gained from running a parallelized program over \( N \) processors is dependent on the amount \( P \), of the program that can be parallelized, i.e.,

\[
\frac{1}{1 - P + \frac{P}{N}}
\]

so it is very important to get \( P \) as large as possible. For example if half of our application is parallelizable (\( P = 0.5 \)) then the limit for performance gain is two no matter how many processors we distribute the problem to.

4.1.1 Parallel programming

To do parallelization of applications when need some means of running several tasks at the same time. Depending on what kind of architecture your program is meant
to run on this is often done a bit different. Below we describe briefly two ways of doing it, one used by VTK and the other used by ParaView.

**Threads**

To distribute computations over several processor on a multiprocessor machine with VTK threading is used. Threading creates several small execution units that can communicate between each other by sharing memory space.

**Message Passing Interface (MPI)**

Another way of distributing an application is by creating independent processes that communicates by sending messages to each other. This fit well in a cluster environment as the communication between processes works naturally between nodes. A common standard for this is the MPI library [8] that exists in several implementation as MPICH, MSMPI or OpenMPI. ParaView uses the MPI library for distributing computations and it is up to the user of the software to choose MPI implementation at compile time.

### 4.1.2 Network programming

When distributing computation we can also use traditional network programming. The server client model is very common and for example ParaView uses that as well. One client is running on a single workstations sending commands to a distributed server on a cluster. There are two major protocols used when sending data between two computer, TCP and UDP. TCP have a mechanism that reassures that packages arrives at the same order that they where sent and that they are resent if it seems that it got lost on the way. UDP on the other hand is a more bare bone protocol that does not have any measure to see if packages arrives at all or that they arrive in the same order as they are being sent out. Both these protocols complement each other, TCP should be used for traffic that are critical and UDP for traffic that needs faster communication but not 100% correctness. TCP like features can be added to an application communicating with UDP on the application level but introduces an additional overhead both in performace and development time.

### 4.2 Rendering

To render efficiently on several resources we need to sort the data somehow as the assumption is that the data set we are rendering are not suited to render on one node.

#### 4.2.1 Sort first

The sort first method sorts the geometry data spatially and distributes it to the nodes that are suppose to render that particular geometry. That reduces the bandwidth requirements but imposes new problems as the spatial sort needs to be parallel for
the method to scale well. As can be seen in Figure 4.1 we divide the framebuffer into at least N areas also called buckets. Each of the buckets contains the geometry that needs to be projected to that part of the framebuffer. This is the sorting step. Each node now only needs to render the geometry for their part of the framebuffer and then sends it back. All parts of the framebuffer are then trivially merged, as there is no overlap between the individual frames. Depending on how exact the sorting stage is, there might be some data that are rendered several times, but that does not pose a problem for correctness of the final output. Such data would be clipped on the node that is rendering it. OpenGL interceptors, e.g., Chromium, often use this technique.

4.2.2 Sort last

With the sort last algorithm, we do not do any sorting before distributing the data, we simply divide the data set into chunks and send them off to the nodes. Each node then renders the geometry that has been given to a framebuffer as big as the final output framebuffer. So for N nodes we render N number of framebuffers with the same size as the final output. All of these framebuffers are then accumulated, and now comes the sorting part. Two framebuffers are compared pixel by pixel to...
Figure 4.2: Two nodes send back full sized framebuffer containing both color and depth values. The framebuffers are then merged on the server by comparing depth values pixel per pixel discarding occluded areas.

see which one of them are closest to the viewer by comparing depth values, see Figure 4.2. The one closest is kept in the output buffer and the other is discarded. Sort last is the most commonly used render composition technique today. It scales well with geometry and the composition pass is cheap. This is the method used by Paraview.

4.3 Filtering

The filtering process also needs to be distributed if the visualization should be able to scale. A common problem for all parallelization is concurrency. Several different instances of your program can read and write the same data, which is something that needs to be taken care of so there does not exist any invalid copies of that particular data. Algorithms that are completely independent are those who are the easiest to parallelize and this is something that we look for when turning a serialized program into a parallel one.

4.3.1 Marching cubes

An example of a commonly used filter algorithm is the marching cubes algorithm [24]. It is used to produce isosurfaces from volume data sets. The algorithm traverses the dataset and at each point collects eight voxels and creates a cube where each voxel represents a corner of this cube. For each combination of voxels being inside or outside the surface there’s a precalculated look up table with how the corresponding voxels would be represented as triangles. The result from this is then added to the list of output triangles. When the whole volume is traversed, we have a list of triangles representing an isosurface of the volume. The parallelizable parts here is the actual traversing of the voxels, this is a read operation and we never change the values of the volume data set. Here, we can divide the volume and distribute it to the nodes we have and get back a list of triangles. These triangles must then sorted in some way before rendering as described above. The marching cubes algorithm is
an example of an algorithm that is highly parallelizable due to all the independent
operations. All we need is the synchronization in the end when all nodes are done
with the calculation. Details on marching cubes can be found in [21] for example.

4.4 Tests

We started our test sequence by running ParaView locally on one of the workstations
running on one core without using mpi. This was to get familiar with the program
and to have some kind of notion of the basic performance of ParaView. The next
step was to distribute the visualization by having the secondary workstation act as
both data and render server and the primary one as a client. After that the logical
step was to install MPI on the eight core machine and run the server on all eight
cores. Finally we setup the server on different configurations on the cluster and used
the workstation as client. The data that was rendered was volume data of different
sizes and molecule structures of different sizes. We also tried several different filters
to see if we could stress the architecture even further.
Chapter 5

GPU based visualization

In similarity with ParaView’s render server, we can deploy a part of the rendering on our workstation with far superior render power than a single cluster node.

5.1 General programming of GPUs

During the last several years the computational power of the GPU have surpassed that of the more general x86 based processors. If one can fit the problem into a format that fits on the GPU a lot of performance can be gained.

5.2 Rendering

We concentrated on comparing the render performance of a specific implementation if volume rendering on the GPU with volume rendering in VTK and ParaView. Note that the comparison is shifted toward our experiment due to we only implement a small part of a renderer and VTK is a toolkit with a complete visualization pipeline. The reason why we did not implement our test into VTK is due to time constraints and the result of this is just meant as a measurement on what kind of performance would be expected if this is incorporated into VTK and if it is worth the investment in hardware and development time.

5.2.1 Deferred rendering

One idea that we had was to render triangle data at the cluster but not doing any shading but instead sending back buffers with geometry information such as normals, material values etc. This could then be used to do the shading on the workstation with the GPU instead. This was an idea that we did not implement but we still feel that it is worth mentioning and that it would be interesting to see if there is any benefits to this solution.

5.2.2 Volume visualization

We chose direct rendering of a volume as we felt that it is was good test with just rendering and no filtering involved. It is also simpler to max out the testing as
it is a rather heavy way of rendering. The raycasting method that we chose to implement works by rendering the front and back faces of a bounding box to the volume. By saving the depth values from these we have for each pixel a vector which we discretely traverse the volume along. And this whole computation is done purely on the GPU.

5.3 Filtering

There has been several research projects (for example [21]) on filtering on the GPU instead of the CPU. The general benefit of doing filtering on the GPU is that the data resides on the GPU and does not need to be transferred when the rendering occur. A draw back of this is if the data is needed for an algorithm that will run on the CPU.

5.4 Implementation

We setup a network of four computers with a newer model of budget range graphics cards (Nvidia 8500GT). One node acts as both server and client and we deal with concurrency by passing a token to the other clients. When the client gets the token it renders one frame, compresses it and sends it back to the server. We use a Gnu Public License (GPL) library for the lossless compression that we are performing. The library is the well known bzip2 library [20]. We use lossless compression as we try to keep the rendering as correct as possible and we do not perform any other optimization techniques as subsampling, quantization of colors or other lossy compression of data. We use TCP for communication as we also want reliable transfers between the client and the server with minimal packet loss. The client/server node renders a frame after it sent out tokens to the other clients. When all data has been gathered and sent to the graphics card the composition of the different framebuffers is performed on the graphics card.

5.4.1 Rendering types

We implemented two transfer functions that have different implication on performance. The first one is Maximum Intensity Projection (MIP), where we traverse the whole length of the ray and return the maximum value encountered. This method makes the rendering look a bit like an x-ray picture where we can see the maximum density in the current pixel. At the same time we loose all depth information in the picture. This method is useful for visualizations and is often used in for example the field of medical visualizations. One implementation benefit and the reason why we included this method is that the result of the raycasting return very little data. As mentioned before there is no depth value here and there is only an intensity value returned. So the total size of the framebuffer that needs to be sent to the host is

\[ \text{totalSize} = \text{intensityPrecision} \times \text{framebufferWidth} \times \text{framebufferHeight} \]

and to have a lower bound of the amount of data that need to be sent we choose to have 256 intensity levels i.e. 8 bits precision. The other transfer function can be
described as finding the surface of the volume. We set a threshold value defining at what density value we deem that the surface is. We then traverse our rays until we sample a position inside the volume that have equal or greater density value than our threshold. When found the depth of our ray surface intersection we approximate the normal at the intersection point by taking the centered differences

\[
Normal = \begin{bmatrix}
(x + \epsilon, y, z) - f(x - \epsilon, y, z) \\
f(x, y + \epsilon, z) - f(x, y - \epsilon, z) \\
f(x, y, z + \epsilon) - f(x, y, z - \epsilon)
\end{bmatrix}
\]

where \( \epsilon \) is a suitable value close enough to the point of interest. The normal is then used to calculate Lambertian shading \( Intensity = \max(0, N \cdot L) \), that is, the intensity is equal to the dot product of the normal and the vector pointing towards the light source. If the intensity value is below zero we set it to zero. We now have three values that we are sending back to the host, the depth value found at the intersection, the density value where we stopped traversing the ray and finally the intensity value calculated with Lambertian shading see Figure 5.1 for result. This gives us the following

\[
totalSize = (depth + intensity + shading) \times framebufferWidth \times framebufferHeight
\]

where we choose to represent depth, intensity and shading values with a 32 bit floating point number. Both of these transfer functions also differ on the host where different ways of combining the values are needed. The MIP function needs to be performed once more at the host. For each pixel in the framebuffer we compare which one of the four targets that have the greatest intensity and that is what we present to the user. The other transfer function needs to include occlusion when combining the different framebuffers. For each pixel, we perform a test to see which one is closer to the near plane, the standard depth test normally made when rendering with a z-buffer. The intensity value of the closest point in the current pixel is then used to sample a color value from a color lookup table and then finally multiplied with the shading value sent back from the client.

### 5.5 Bottlenecks

There are some obvious bottlenecks present in this test project, one being network speed. The testing was performed on a 100Mbit/s network which equals 12.5MB/s of transfer speed. If we would set a resolution to \( 512 \times 512 \) in and render with MIP then we would have a frame size of 0.25MB which would give us a theoretical maximum render speed at 50 frames per second. In the other scenario where we shade the volume and send more data we would have a cap at 4 frames per second even at this rather low resolution and this can not be considered interactive any more. By compressing the framebuffer we see an increase in performance but the lossless compression that we choose work quite fast at this resolution but not to as effective. In general we achieve a compression ratio of 2.5:1 which would bring up the theoretical max of this scenario to around 10 frames per second at least which we consider to be interactive at least. At this lower resolution we do not see any
Figure 5.1: An example of the output from the raycasting on the GPU with shading

other bottlenecks as can come up at higher resolution such as transfer speed from RAM to GPU memory or render performance. The render performance on a single node is very good and is mostly effected by the resolution of the framebuffer.
Chapter 6

Results

The system was evaluated on performance, usability and quality of the display conditions.

6.1 Quality of our stereographic display

The most common feedback from our users except from the comments on the actual stereo effect, is that the stereo display is very bright and colorful. The majority of our users have either used anaglyph stereo technique or polarized light technique before and they are then comparing the new experience with that. As the room is intended to work as a lecture room, both with regular lectures and seminars being held there, it is important to see what kind of viewing conditions that effect the stereographic display. During some of these activities people might need to take and read notes and people will be spread out all over the room. We set up a test to under what light conditions we perceive the stereographic display as usable and a second one where we ran several different visualizations and viewed it at different positions in the room.

6.1.1 Brightness

For the brightness test we ran three stereoscopic visualizations with different content. One had a very bright background, one darker almost black background and finally one with mid level brightness background. With the lights turned off and the windows covered the stereo screen works at an optimum. But the the light from the screen in all three test cases are as expected to faint to read and write comfortable in for a longer period of time. Removing the curtains from the window during daytime, with no sun shining directly into the room we performed the test again. This time we feel that you can comfortably read and write and both the mid bright background scene and the bright background scene is clearly viewable, the brightest one being a bit better. But the dark background scene is close to unusable, very little stereo effect can be seen due to the screen gets to bright from the daylight.

The final test was meant to be made in two stages by turning on the strip light in the ceiling at lower intensity and at full intensity. But as soon as we turned on the light we figured out that you cannot use the glasses in connection with having the
fluorescent lights turned on. The color difference between left and right eye becomes
to great and if you for example look at a paper it seems to flicker a lot between red
and green. But still if you view the screen with the light on you still see the stereo
effect bright and clear in the case where the light are the lower intensity. At max
intensity the stereo screen becomes to bright and you loose to much details.

6.1.2 Perspective

For the second test we ran the same visualizations and positioned ourselves at the
the edges of the room to get a feel for the visualization quality at the in our case
most extreme angles. There is a perceivable difference when sitting in the front row
at the along the side of the wall then when you are sitting in the middle of the room.
But even though there are a perceivable difference there is still a rather good stereo
effect even at a large offset from the optimal viewing position which is in the center
of the room.

6.1.3 Lights

So our conclusion to this is that the stereo screen gives an sufficient stereographic
view but the current light condition is not satisfactory. So what can be done to
improve this? A simple solution would be just to recommend the users to remove
their glasses when they are not looking directly at the screen. But to investigate
further we took some samples of the strip light with a spectrometer (see Figure 6.1).
We have chosen to only show the part of the spectra that are near the visible range of
humans, 300nm to 700nm. In the graph we see two sharp spikes as an artifact of how
phosphoric light work. When we then filter the phosphoric light through the glasses
we get a graph like in Figure 6.2 where we can see a red graph for the left filter and a
blue graph for the blue filter. The reason for the flickering is that one filter removes
one spike while the other filter removes the other creating unbalance in energy levels.
To find another solution we tried to sample a Light-emitting diode (LED) lamp to
see if it would be more fitting for our project. The result of the unfiltered light
can be seen in Figure 6.3 which does not have the sharp spikes that the phosphoric
light have. But a simple visual test with having a piece of paper and trying to read
it with the INFITEC glasses on also causes flickering. The filtered light have the
following form (see Figure 6.4) and here we can see an unbalance among the three
different filtered ranges. This unbalance causes a flickering appearance, a lot less
than the phosphoric light but we still do not deem it good enough to use for a longer
period of time. The last light that we tried was a standard 60W light bulb. The
spectra from that looks like in Figure 6.5 a more evenly distributed graph with more
increasing values as we get closer to more reddish colors. A reading test with the
glasses in this light is much more pleasant and we deem this to be actually usable.
When viewing the filtered light graph (see Figure 6.6) we see a much more even
distribution among the pairs of filtered spectral ranges as expected from the visual
result.
6.1.4 Projection modes

As mentioned earlier we have two projection modes, one actually for dual display plane and one with off axis projection. The dual display plane projection is theoretically wrong here but we have used it in several test and got good results. Yet again you cannot really tell that it is something wrong with the projection when viewing those visualization but this might just be a lucky streak. It might just work in our test cases and as we have both types implemented we would suggest that off axis projection is used if possible.

6.1.5 Calibration

Our setup is meant to always be fixed, at the same position and never be moved. So theoretically we would only need to put the projectors in place and calibrate them once and for all. The reality for this is a bit different. During the setup of this we needed recalibrate the system a few times. The actual calibration procedure is not technically advanced but it can be rather time consuming, the quality of it is mostly proportional to the time spent doing it. The projectors, as could be seen in Figure 1.5, are mounted above each other in a rack. The rack is not fixed so if we need to move the projectors we can move the whole rack. The projectors
themselves are each mounted in the rack with four mechanical arms. We have setup each projector with keystone and lens shift that we see fit and we try not to change those values when we recalibrate the projectors. Instead we move the mechanical arms that the projectors are mounted in by tightening and loosening the screws that control the arms. With this we can get a both tilting and rotation effect that is need to co-align the two projector images. To check the alignment of the two projectors we setup the computer in clone mode, both projectors shows the same image. The calibration pattern used for this is the same as in Figure 6.8, it is just picture that we created to try to capture the effect of tilting and rotation. With our black and white we can see the difference between the projector if we view the screen without glasses. The spectral shifting will produce one greenish pattern and one pinkish pattern and by this can we differentiate between the two projected images. The procedure is now to rotate and tilt the projectors until they somewhat align. To get a pixel perfect alignment is normally a very time consuming procedure and we noticed that it is not really needed for a good stereoscopic effect. Even when the projectors are rather ill calibrated there is a not a really noticeable difference when viewing a stereographic visualization. Most of our experiments have been with non immersive visualizations where most of the detail is in the middle of the screen. It is also the middle of the screen that is easiest to get well calibrated, but we have run
Figure 6.3: A spectral sample from a LED lamp made in the visualization studio as a comparison to the current strobe light

an example with text at the border of the screen on an ill calibrated screen and still feel that it is readable. The human mind seems rather insensitive to this kind of discrepancy. There is one big downside with working with an ill calibrated system though. In the majority of the cases we use clone mode when we are not viewing stereoscopic content on the screen. An ill calibrated system is very noticeable in 2D when navigating a normal windows system without the glasses on due to all the blurriness of the misaligned text.

6.1.6 Glasses

We have also evaluated three different pair of glasses for our setup seen in Figure 6.7. These three glasses varies in price, comfort, field of view and viewing quality. Price ranges from cheapest to the left in the image, mid range price in the middle and the most expensive pair to the right. The cheapest pair is made out of rigid plastic and does not really fit when the user is wearing his own glasses beneath but the other two models have lots of space underneath so that it they are comfortable to wear them together with a normal pair of glasses. The most expensive pair has by far the greatest field of view but at the same time some difference in color composition
depending on where on the glasses you look through. All of the three models have this discrepancy but it is most noticeable with the most expensive pair probably due to that they have curved glass. We see that the mid range pair fit our need the best as we don’t need the larger field of view that the more expensive version offer and they are at least as comfortable with out the more noticeable color discrepancy that can be noticed when you turn your head.

6.1.7 Conclusion

All in all we are very satisfied with the quality of the stereo screen, even the rather low resolution of the projectors turned out to work nicely. The resolution of 1024 × 768 cannot be considered to be high resolutions with most desktop workstations running at higher resolutions but we have never felt that we were limited by the resolution of the screen so far.
Figure 6.5: A spectral sample of a regular 60 Watt light bulb taken for comparison with the current strobe light from the visualization studio

6.2 VTK

We are using both Linux (Red Hat Enterprise 5) and Windows XP on our workstations. We expect users from all disciplines of the university where both of these operating systems are used. There are some difference between these both setups in our case. The Windows XP version that we are running is a 32-bit Windows Professional edition. The 32 bit version have a limit on how memory a process can allocate and this limits us on how big visualizations we can perform on this system. The Windows system have better graphics driver with better support for stereoscopic displays and better performance. If the performance difference is to the actual driver of the different implementation of VTK is still remain unknown to us. The Linux systems had far fewer crashes with VTK but at the same time we developed more code and did more testing on the Windows system. This was our first time that we got acquainted with the VTK source code and even though the massive size of the code we felt that is was really well structured and easy to read.
Figure 6.6: Spectral sample of a regular 60 Watt light bulb filtered by the spectral filters mounted on the stereoscopic projectors taken for comparison with the current strobe light.

Figure 6.7: The price ranges of the glasses we tested, from lowest to the left and highest to the right.

6.2.1 Usability

The result for our modification to VTK is that our users can bring their VTK script to the lab, add three or four lines of code and then run their visualization in stereo with the operating system of their choice. We can of course not guarantee that it can be viewed in a satisfactory fashion without tweaking view and projection matrices.
but this is a problem that is unique for each visualization and is unsolvable in the general case.

### 6.2.2 Chromium

We also then used the Chromium library to achieve stereo optic rendering. We tested this on some of our own applications to see what kind of effort that is needed to make a script and what kind of performance you can get. The test was performed on Linux only. This works fine in most cases and we get a good stereo optic effect. We had some problems with some features in our programs most commonly more modern features as programmable shaders. From our understanding shaders is not yet completely supported by Chromium but it is still an active project with more features added for every release and we are sure that this will soon be completely implemented in an upcoming release.

### 6.3 ParaView

As we did with VTK we also tested ParaView on both Windows and on Linux. Linux was used both for standalone application and server where Windows was only used as client or standalone application. The limits that we saw with the Windows system when working with VTK was not present when working with ParaView in client-server mode where the client resides on on workstation running Linux and the client running on the windows XP machine. To compile ParaView we made makefiles with the CMake files that are included in the distribution of ParaView. We used the latest source distribution (3.3) of ParaView that we downloaded of the subversion (SVN) server. We downloaded and compiled all the necessary libraries including OpenMPI where we used the latest stable version, 1.2.6. The cluster does not have any kind of GPU and we just need to render directly to RAM so the compilation process had to be changed to conform to this.
6.3.1 Performance

The render speed that we achieved in the end is satisfactory. We have not done any benchmarking we we actually measured render time per frame and compared the scaling but instead we felt that the visual result was enough. Memory consumption and not rendering speed is the bottle neck for large volumes. The volume sizes we have worked with all fits into memory but the problem arrives when we start manipulating the volumes through different filters. VKT keeps all information from all stages of the pipeline as default, the design is that you should be able to traverse the pipeline forwards and backwards at any point. This behavior ramps up the memory consumption but it is possible to flush the pipeline if necessary. Still we saw that memory consumption was by far our largest bottle neck, not network bandwidth or fillrate/render performance. This pattern might change later on if we would change the resolution of the display which would tax the network bandwidth more and at the same time fill more pixels. The models we used for molecular structures have been faster to render and those not fill up the memory with the visualization techniques as we used during the project.

Distribution

At first we where rather complexed about the render performance of ParaView but after some tweaking we felt more at ease. For smaller datasets we see as expected a great speed boost by using display list on the machine with the more powerful graphics card and we can fit rather large models in the graphics RAM of 768 MB installed on the card. The render performance is noticeable slower on the other machine as one would expect. For volume rendering where ParaView with its VTK base work in software we see a lot of difference when we distribute our computations onto for example the eight core machine. In our scenario we see that jobs that requires less or equal to eight cores works best if they are kept locally on the workstations and not to send it to the cluster. This method is both superior in performance and that we do not need to take up slots on the actual cluster that other users might need.

Scaling

ParaView scales well in all our test. The interactivity goes up for each processor that you add to the server but it is also important to see if the problem needs to be distributed at all. A lot of problems actually fits on one of the workstations and then the performance is at the its best.

Usability

ParaViews interface was a pleasant surprise. With out prior knowledge we felt that it was usable and easy and fast to get visualizations up and running. Prior knowledge with VTK helped a lot and further extensions to ParaView with its scriptable GUI seems also easy.
6.4 Framework for grid/single workstation

Our own code base for porting code to stereoscopic displays and our test for distributed rendering on GPUs worked as expected. There are still many ways that one could develop both these projects but we feel that they work in a satisfactory way for what we aimed for in this project.

6.4.1 Library for porting application to stereo

The library for porting stereo have some high level functions for faster porting an application to stereo by just changing the main render loop a bit see the example below. The library was developed by porting a GLUT program but at the same time we tried to keep it as general as possible and also as readable as possible. Both side by side buffering and quad buffering is supported. All three viewing paradigms are also supported, parallel projection, toe in projection and off axis projection. There is also support for switching display mode from single pane display mode that is the normal mode, to clone mode used with quad buffering or horizontal display mode used with side by side buffering. The pointer experiment described in an earlier chapter is also included for completeness. The source code for the library is included in the appendix.

// Original render loop

setupCamera()
render()
swapBuffer()

// Modified render loop

setupCamera()
update()
render()
update()
render()
swapBuffer()

6.4.2 Distributed rendering on GPUs

The result from the experiment with the distributed GPU rendering confirms that what can be seen as common sense... Rendering with special hardware is better or at least faster. The stand alone test application was rather easy to develop and together with the good code structure of VTK this method would probably be a moderate task to port to VTK for more testing. Kitware already have documentation [30] on how to use programmable shaders with VTK, the same that we used to implement our experiment. This would take care of rendering the volume locally on
one workstation. For distributing the rendering onto several GPUs then larger ef-
fort and more restructuring needs to be done. Another interesting alternative would
be to use the Chromium library used for the non invasive part of the experiment.
Chromium also support a client server structure where one can send OpenGL com-
mands directly over a network. This could be used to distribute the rendering of
the volume over several nodes but still keep the structure of VTK.

6.5 Conclusions of the results

6.5.1 Stereoptic display

We have tested three ways for our users to use the stereoptic display as smoothly
as possible, by using the modified version of VTK, by using non-invasive method
i.e. Chromium library or by using the rendering library developed especially for
the stereoptic display. Which method one would use does completely depend on
what you are working with. If you are already working with VTK the most usable
solution is to just modify your script to get stereo optic effect. Similar if you plan to
create a new visualization we recommend VTK with all its built in features and then
to convert it to stereoscopic display as described earlier. We do not recommend to
use Chromium for this as the end result would be the same but require a bit more
effort. If you have an application that you have the source code to you could take
on of two routes, either make a Chromium setup if you do not have the time or the
knowledge to modify the graphics engine of the application. The other route would
then be to modify the program to use the functions provided from our library to
modify the application to render in stereo optic instead. If you know the source
code to the application already e.g. if you wrote some of the code yourself this will
probably be faster. Finally if you have an application that are closed source and
does not support stereo optic output natively the Chromium route is the way to go.
An example for a simple Chromium script that we used can be found the appendix.

6.5.2 ParaView

ParaView works as expected both on the cluster and on the local workstation. It
can be used for small scale visualizations and bigger ones as can be seen in Fig-
ure 6.9. The performance on the two local workstation working to distribute the
visualizations is a great improvement compared to using a similar solution on a local
workstation and for jobs that can benefit up to eight cores we recommend this for
visualization with our with out the stereoscopic display. See the appendix for some
further notes on practical details on using ParaView.
Figure 6.9: A dataset of a lion captured at CMIV, Linköping by Anders Persson. The size of the data set is $512 \times 512 \times 3865$ voxels, requires 2GB of storage space and is rendered interactively by using the two workstations assigned to this project.
Chapter 7

Discussion and further results

7.1 Performance

We have in thesis work viewed performance as usability and we are now at a schism where computer architecture are evolving to a new state and with this the performance metrics. Today we are moving toward multcore architectures where operations that where very expensive on a multi processor system are much less costly on a multcore processor. One of these is communication between processors. With multiple processor system communication is expensive and the majority of algorithm that are written for this kind of architecture have been designed with this in mind. Now that this is changed the visualization field might need to do what is already being done in the field of Computational Science and High Performance Computing(HPC), revisit the old algorithms and update to fit the multcore revolution. We also see the difference between High Performance Visualization(HPV) and HPC development cycles. HPV specific hardware, mostly GPUs, have a much shorter lifespan than the architectures used for HPC. HPC code is written to be used for more than a decade and we can see the same pattern in HPV. To have a truly HPV system we need conform even more to the hardware trends are put a lot of effort in writing modular code that are easy to expand and adapt for each hardware generation. The only industry that is following the fast development of GPUs today is the game industry and maybe the HPV community can learn something from there as they have shorter support cycles that are more bound to the generations of GPUs. Another way of looking at performance that is interesting is for example our experiences with molecular rendering in ParaView. Our results showed that today with the models and the hardware that we use that we were not limited by rendering performance in the terms of triangle count. Bottlenecks as memory bandwidth have not been tested here but we feel that molecular visualization have other exiting ways of being enhanced. Compare Figure 1.7 to Figure 7.1 where they have implemented methods to enhance the depth cues based on [19]. They have added shadows and edge enhancement and we see this as a really interesting alternative to put resources on.
7.2 Distributed visualization

The word distributed has been used in the context of “distributing your computations onto several resources”. An alternative view of this is to view distributed visualization as “distributing the resources to the users”. ParaView is made for a single client interacting with a distributed server. The client can be at a remote position but here can still only be one client at a time. We see a possible future direction to take this project defined by three keywords.

- Distributed - Parallelize computations, on a multicore processor or a cluster resource, to offer better performance and more usability
- Remote - Users should be able to access the resources from anywhere with any kind of computer
- Collaborative - Several users should be able to view and interact with the same visualization at the same time

An approach would be to build upon and extend ParaView’s distributed functionality with parallel computations. Then to add more remote and collaborative capabilities by making it possible to be several simultaneous clients on different machines. In [15] we found one description on how to modify ParaView for a more collaborative approach. Other ideas is to distribute the resources by VirtualGL [17] for example. VirtualGL would give the users the ability to run more demanding GPU-based
programs from resources which lacks this in the normal case, e.g., a standard laptop. It works by distributing OpenGL calls to a remote GPU render them there and then send the result back. The collaborative idea could be implemented by writing a Verse plug in for ParaView. Verse is a collaborative network protocol develop at the Royal Technical Institute of Sweden by Brink et. al. [16]. A verse plug in to ParaView would have the benefit more of the visualization pipeline could be made interactive. By setting up a verse-server on the cluster and then let the server communication with the simulation program as the simulation is being computed we could present results during the actual computation. Users could roam the data and start to analyze it at an early stage for faster feedback and exploration of their visualization. We provide a small sketch of a possible design for our proposed approach in Figure 7.2. The proposed idea also have similarities with a solution made by IBM called Deep Computational Visualization (DCV) [12].

![Figure 7.2: A proposal on how to continue to extend the current base for the distributed visualization system to accommodate for larger dataset, several collaborative users and a more interactive pipeline.](image)

### 7.3 Stereoscopic displays

We are satisfied with the quality and usability of the stereo projection system that we setup. But at the same time the expectation of stereographical images will probably begin to raise in the following years. Stereographical solutions for games have been deployed for several years already, mostly in the form of Head mounted
displays (HMD) but never been really successful. But now there are beginning to appear consumer models of stereoscopic display screens [11] that target the gaming audience with back-up from the graphics card industry [10]. At the same time the movie industry are deploying more and more cinemas with stereo graphical displays and more and more movies are rendered in stereoscopic variations. These two trends will make people more used to stereo graphical displays and then the system we build up might need to be reevaluated to see if the quality is good enough for users that have a different reference. As a benefit for our goal with a usable stereoscopic display we can also see this as a bonus. Our future users will be used to stereo graphical content and will expect us to deliver this to them. And as one of the products of this project we can give them this. During this master thesis project the inauguration of Three-Dimensional Image Studio for Uppsala (3DIS4U) took place and is now open to all users at Uppsala University.
Chapter 8

Acknowledgments

I would like to thank Anders Hast my adviser for all the long and interesting discussion ranging over the whole field of visualization. I would also like to thank Ingela Nyström for her great support during this master thesis and by giving me the opportunity to do this very interesting thesis. Furthermore I would like to express my gratitude to Erik Vidholm, Filip Malmberg and the rest of the employees at Center of Image Analysis for many good advises and for creating a nice working environment. And finally I would like to thank my fellow master student Boris Berkau for many late and fun hours spent at CBA.
Chapter 9

Further reading

A good tutorial on VTK

- VTK Reference [4]

For more information on ParaView, we really recommend the

- ParaView Manual [13]

For more detail regarding stereoscopic system a good reference can be found in a free resource

- Stereoscopic Handbook [22]

Raycasting on GPUs was the subject of a tutorial at SIGGRAPH a few years back and is available for free online at

- SIGGRAPH course in raycasting on the GPU [26]

A good introduction to concurrent and multiprocessor programming can be found in the book by Matson et al.

- Patterns for parallel programming [29]
Bibliography


[3] The Visualization Toolkit vtk.org, retrieved on 080119


[7] Isis www.uppmax.uu.se/systems/isis-cluster/isis, retrieved on 080119

[8] MPI www.mpi.org, retrieved on 080322

[9] UPPMAX Support www.uppmax.uu.se/support, retrieved on 071204


[12] IBM DCV www-03.ibm.com/systems/deepcomputing/visualization/, retrieved on 080407


[18] Qutemol qutemol.sourceforge.net/, retrieved on 080212


[27] A multimodal Virtual Reality Interface for 3D Interaction with VTK. Arjan J.F. Kok and Robert van Liere. not published


[29] Patterns for Parallel Programming. Timothy Mattson, Beverly Sanders, Berna Massingill. Addison-Wesley Professional 2004

Appendix A

Converting an existing VTK program to stereo

This is a short tutorial on how to change an already existing VTK application written in Python to use the stereo wall. The example program is written to be as simple as possible but to still show the ease of use of the system. The lines that you need to add are the following. Be aware that the test scene given does not have a strong sense of depth, it just meant as an example on how to port your code.

- `renderWindow.StereoCapableWindow()` — This is already included in VTK and must be called before you create your window
- `renderWindow.StereoOn()` — Turn on stereo, also in default VTK
- `renderWindow.SetStereoTypeToInfitec()` — This is new, and sets the stereo to accommodate our stereo wall
- `renderWindow.FullScreenOn()` — As we always need fullscreen to view stereographical content we must call this. Also new to this function call is that we try to switch the display mode to horizontal span for Windows configurations.
- `renderWindow.CursorOff()` — This is voluntary but recommended as the pointer can interfere with depth cues given on the stereo wall

If the screen mode is changed by the FullScreenOn call, then we need to exit the application properly to return to the previous display mode. The resetting of the display mode is performed in the destructor of the window and if we force a quit of the application this will not be called. All standard calls that effect stereo viewing included in default VTK still apply, one of the more useful ones here is the `renderWindow.GetCurrentCamera.SetViewAngle(angle)` in order to change the separation between the virtual cameras.
#!/usr/bin/env python

# This shows how to convert a simple application to stereo

import vtk

# Create a cylinder
cylinder = vtk.vtkCylinderSource()

# Create a mapper and link it to the cylinder source
cylinderMapper = vtk.vtkPolyDataMapper()
cylinderMapper.SetInputConnection(cylinder.GetOutputPort())

# Create an actor and link it to the mapper
cylinderActor = vtk.vtkActor()
cylinderActor.SetMapper(cubeMapper)
cylinderActor.RotateX(45.0)

# Create a renderer and a render window. Also add a interactor so
# we can spin our cylinder a bit
ren = vtk.vtkRenderer()
renWin = vtk.vtkRenderWindow()

# setup the window to be stereo capable, turn on stereo and set the
# stereo mode
renWin.StereoCapableWindowOn()
renWin.StereoRenderOn()
renWin.SetStereoTypeToInfitec()

# hide the cursor
renWin.HideCursor()

renWin.AddRenderer(ren)
iren = vtk.vtkRenderWindowInteractor()
iren.SetRenderWindow(renWin)

# Add the actors to the renderer, set the background and size
ren.AddActor(cylinderActor)
ren.SetBackground(0.1, 0.2, 0.4)
renWin.SetSize(200, 200)

# turn on fullscreen and switch display mode if on Windows
renWin.FullScreenOn()

# Initialize the interactor before the event loop
iren.Initialize()

# Render
renWin.Render()

# Start the event loop.
iren.Start()
Appendix B

Source code for the stereo library

/*
 * Stereoscopic Library
 * Helper functions for porting an application to use with a single
 * planar
 * stereoscopic display plane.
 * Depends on the OpenGL API and NVIDIA’s control panel API (NVCP)
 * author: Martin Ericsson
 *
 * Lacks error handling and is written for clarity.
 */

#pragma once

#ifndef STEREOGRAPHIC_H
#define STEREOGRAPHIC_H

// function pointer to render function
#include "main.h"

namespace StereoscopicLibrary {

    // const... use them to set modes
    const static int ToeInProjection = 0x00;
    const static int ParallelProjection = 0x01;
    const static int OffAxisProjection = 0x02;
    const static int QuadBuffer = 0x13;
    const static int SideBySideBuffer = 0x14;
    const static int DisplayModeStandard = 0x20;
    const static int DisplayModeClone = 0x21;
    const static int DisplayModeHorizontalSpan = 0x22;

    // the main class
    class StereoGraphics {
    public:
        StereoGraphics();
        ~StereoGraphics();
        void switchDisplayMode(int displayMode);
        void resetDisplayMode();
        void update();
        void crossEyes();
        void setMousePosition(int x, int y);
    }
void setBufferType(int bufferType);
void setBufferDimensions(int width, int height);
void setEyeSeparation(float eyeSeparation);
void setFieldOfView(float fieldOfView);
void setProjectionMethod(int projectionMethod);
void setViewPosition(float eyeX, float eyeY, float eyeZ,
                     float upX, float upY, float upZ,
                     float atX, float atY, float atZ);
void setViewDistance(float viewDistance);
int getBufferMode();
int getProjectionMode();
void renderPointer();
void reshape(int width, int height);

private:
    // private consts, made to make the code a bit more readable
    const static int LeftEye = 0;
    const static int RightEye = 1;

    bool setDisplayMode(int mode);
    void normalize(float &x, float &y, float &z);
    float toRadians(float angle);

    GLfloat eyeSeparation, fieldOfView, nearPlane, farPlane,
            viewDistance, eyeX, eyeY, eyeZ, upX, upY, upZ, atX, atY, atZ;
    GLint eye, projectionMode, bufferMode, oldDisplayMode,
         bufferWidth, bufferHeight, mouseX, mouseY;
    GLboolean switchedDisplayMode, switchEyes, invertMouseYAxis;
    HINSTANCE hLib;
};

#endif
#include "StereoGraphics.h"

using namespace StereoscopicLibrary;

StereoGraphics::StereoGraphics () {
    eye = LeftEye;
    invertMouseYAxis = true;
    switchedDisplayMode = false;
    eyeSeparation = 1.f;
    projectionMode = ToeInProjection;
    switchEyes = false;
    fieldOfView = 180.f * atan(192.0/320.0) / 3.1415f;
    printf("FOV:%f\n", fieldOfView);
    nearPlane = 0.1f;
    farPlane = 100.f;
    viewDistance = 15.f;
    hLib = NULL;
}

StereoGraphics::~StereoGraphics () {
    resetDisplayMode();
}

/*
 * Tries to enter the desired display mode
 * Throws an error otherwise
 */

void StereoGraphics::switchDisplayMode(int displayMode) {
    if (hLib == NULL) {
        hLib = LoadLibrary(L"nvcl.dll"); // the L is not a spelling error, it is a cast
        if (hLib == NULL) {
            fprintf(stderr, "Cannot load nvcl.dll\n");
            return;
        }
    } else {
        memset(&displayInfo, 0, sizeof(displayInfo));
        displayInfo.cbSize = sizeof(displayInfo);
        displayInfo.dwInputFields1 = 0xffffffff;
        displayInfo.dwInputFields2 = 0xffffffff;
    }
}

bool StereoGraphics::setDisplayMode(int mode) {
    NVDISPLAYINFO displayInfo = {0};
    fNvGetDisplayInfo pfNvGetDisplayInfo = (fNvGetDisplayInfo)::GetProcAddress(hLib, "NvGetDisplayInfo");
    if (pfNvGetDisplayInfo == NULL) {
        printf("Unable to get a pointer to NvGetDisplayInfo\n");
        return false;
    } else {
        memset(&displayInfo, 0, sizeof(displayInfo));
        displayInfo.cbSize = sizeof(displayInfo);
        displayInfo.dwInputFields1 = 0xffffffff;
        displayInfo.dwInputFields2 = 0xffffffff;
    }
}
oldDisplayMode = displayInfo.nDisplayMode;

if (NVdtcfgex == NULL)
{
    printf("Unable to get a pointer to NVdtcfgex\n");
    return false;
}

switch (mode) {
    case DisplayModeStandard:
        NVdtcfgex("setview0_standard");
        printf("Display mode set to standard\n");
        break;
    case DisplayModeClone:
        NVdtcfgex("setview0_clone");
        printf("Display mode set to clone\n");
        break;
    case DisplayModeHorizontalSpan:
        NVdtcfgex("setview0_hspan");
        printf("Display mode set to horizontal span\n");
        break;
    default:
        fprintf(stderr, "Unknown displaymode requested");
        break;
}

return true;

/*
 * Explicitly resets the displaymode to the previous one
 * Also called implicitly by the destructor
 */
void StereoGraphics::resetDisplayMode() {
    if (switchedDisplayMode) {
        setDisplayMode(oldDisplayMode);
        switchedDisplayMode = false;
        FreeLibrary(hLib);
    }
}

void StereoGraphics::setEyeSeparation(float eyeSeparation) {
    this->eyeSeparation = eyeSeparation;
}

void StereoGraphics::setFieldOfView(float fieldOfView) {
    this->fieldOfView = fieldOfView;
}

/*
 * Updates the current modelview matrix and projection matrix
 * to the previously setup projection modes.
 * Alternates between left and right eye automagically
 */
void StereoGraphics::update() {
    // setup draw area
    if (bufferMode == QuadBuffer) {

eye == LeftEye ? glDrawBuffer(GL_BACK_LEFT) : glDrawBuffer(
    GL_BACK_RIGHT);
}

else {
    int leftBorder = (eye == LeftEye) ? 0 : bufferWidth / 2;
glViewport(leftBorder, 0, bufferWidth/2, bufferHeight);
glScissor(leftBorder, 0, bufferWidth/2, bufferHeight);
}

// set the projection
float direction;
if((eye == LeftEye) ^ switchEyes) {
    direction = eyeSeparation * 0.5f;
} else {
    direction = -eyeSeparation * 0.5f;
}
float modelview[16];
glGetFloatv(GL_MODELVIEW_MATRIX, modelview);
float rightX = atY * upZ - upY * atZ;
float rightY = atZ * upX - upZ * atX;
float rightZ = atX * upY - upX * atY;
normalize(rightX, rightY, rightZ);
glLoadIdentity();
float top = nearPlane * tan(toRadians(fieldOfView*0.5f));
float aspect = 0.5f * (float)bufferWidth/bufferHeight;
float leftF = -aspect * top + direction * nearPlane / viewDistance;
float rightF = aspect * top + direction * nearPlane / viewDistance;

switch(projectionMode) {
    case ToeInProjection:
        gluLookAt(eyeX + rightX * direction, eyeY + rightY *
            direction, eyeZ + rightZ * direction,
            eyeX + atX * viewDistance, eyeY + atY *
                viewDistance, eyeZ + atZ * viewDistance,
            upX, upY, upZ);
        break;
    case ParallelProjection:
        gluLookAt(eyeX + rightX * direction, eyeY + rightY *
            direction, eyeZ + rightZ * direction,
            eyeX + rightX * direction + atX, eyeY + rightY *
                direction + atY, eyeZ + rightZ * direction +
            atZ,
            upX, upY, upZ);
        break;
    case OffAxisProjection:
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluFrustum(leftF, rightF, -top, top, nearPlane, farPlane);
        //glTranslatef(direction, 0.0, 0.0);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        //gluLookAt(eyeX, eyeY, eyeZ, eyeX + atX, eyeY + atY, eyeZ
            + atZ, upX, upY, upZ);
        gluLookAt(eyeX + rightX * direction, eyeY + rightY *
            direction, eyeZ + rightZ * direction,
            77
\[ \text{eye} = (\text{eye} == \text{LeftEye}) \? \text{RightEye} : \text{LeftEye} ; \]

```cpp
void StereoGraphics::setViewPosition(float eyeX, float eyeY, float eyeZ, float upX, float upY, float upZ, float atX, float atY, float atZ)
{
    this->eyeX = eyeX;
    this->eyeY = eyeY;
    this->eyeZ = eyeZ;
    this->upX = upX;
    this->upY = upY;
    this->upZ = upZ;
    this->atX = atX - eyeX;
    this->atY = atY - eyeY;
    this->atZ = atZ - eyeZ;
    normalize(atX, atY, atZ);
}
```

```cpp
void StereoGraphics::setProjectionMethod(int projectionMethod) {
    switch(projectionMethod) {
    case ToeInProjection :
        // set
        reshape(bufferWidth, bufferHeight);
        break;
    case ParallelProjection :
        // set
        reshape(bufferWidth, bufferHeight);
        break;
    case OffAxisProjection :
        // set
        break;
    default:
        // throw error
        break;
    }
    this->projectionMode = projectionMethod;
}
```

```cpp
void StereoGraphics::setBufferType(int bufferType) {
    switch(bufferType) {
    case QuadBuffer :
        bufferMode = QuadBuffer;
        break;
    case SideBySideBuffer :
        glEnable(GL_SCISSOR_TEST);
        bufferMode = SideBySideBuffer;
        break;
    ```
```c
default:

    break;
}

int StereoGraphics::getBufferMode() {
    return bufferMode;
}

int StereoGraphics::getProjectionMode() {
    return projectionMode;
}

void StereoGraphics::setBufferDimensions(int width, int height) {
    bufferWidth = width;
    bufferHeight = height;
}

void StereoGraphics::reshape(int width, int height) {
    setBufferDimensions(width, height);
    if (bufferMode == SideBySideBuffer) {
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(fieldOfView, 0.5f * float(bufferWidth) / bufferHeight, nearPlane, farPlane);
        glMatrixMode(GL_MODELVIEW);
    } else {
        glViewport(0, 0, bufferWidth, bufferHeight);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(fieldOfView, float(bufferWidth) / bufferHeight, nearPlane, farPlane);
        glMatrixMode(GL_MODELVIEW);
    }
}

void StereoGraphics::setMousePosition(int x, int y) {
    if (bufferMode == SideBySideBuffer)
        mouseX = x > (bufferWidth / 2) ? x - (bufferWidth / 2) : x;
    else
        mouseX = x;
    if (invertMouseYAxis)
        mouseY = bufferHeight - y;
}

void StereoGraphics::renderPointer() {
    GLdouble posX, posY, posZ, oX, oY, oZ;
    GLdouble model_view[16];
    glGetDoublev(GL_MODELVIEW_MATRIX, model_view);
    GLdouble projection[16];
    glGetDoublev(GL_PROJECTION_MATRIX, projection);
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    gluProject(0.0, 0.0, 0.0, model_view, projection, viewport, &oX, &oY, &oZ);
```

if (eye != LeftEye)
    gluUnProject(mouseX, mouseY, oZ, model_view, projection, viewport, &posX, &posY, &posZ);
else
    gluUnProject(mouseX + bufferWidth/2, mouseY, oZ, model_view, projection, viewport, &posX, &posY, &posZ);
glPushMatrix();
glTranslatef(posX, posY, posZ);    
glColor3f(0.8f, 0.45f, 0.8f);
    glutSolidSphere(0.1, 10, 10);
    glPopMatrix();
}

void StereoGraphics::crossEyes()
{
    switchEyes = !switchEyes;
}

void StereoGraphics::normalize(float &x, float &y, float &z) {
    float length = 1.f / sqrt(x*x + y*y + z*z);
    if (length != 0.f) {
        x *= length;
        y *= length;
        z *= length;
    }
}

float StereoGraphics::toRadians(float angle) {
    return 3.1415 * angle / 180.f;
}

void StereoGraphics::setViewDistance(float viewDistance) {
    this->viewDistance = viewDistance;
}
Appendix C

Comments on using ParaView

To run ParaView locally without distributing computations over several processors with render server, data server and client just type

> paraview

The command to run a ParaView server is

> pvserver

This is the recommended way of launching a ParaView server and this set up both render plus data servers on the machine where the command is issued. The default port for communication is 11111 and should be open on all machines at the laboratory. To run a larger job it is recommended that a server is setup at the portofix machine by the command

> mpirun -np x pvserver

where x is the number of cores that we will use for filtering and rendering. Value between two and eight is recommended. As we are not interested in displaying any visualizations on the server it is useful to add the flag –use-offscreen-rendering to the server command. The simplest way to connect to the server once it is up and running is to click the connect button in the ParaView client and choose “portofix” from the list. All the setting are already preconfigured in that setting. To connect to the Isis cluster log in to your UPPMAX account and either make a batch script or request realtime access. The easiest way to get a connect to the cluster is to start the client on the local machine first and connect to “Isis reverse”. Then start the server on Isis with the flag -rc -client -host=hostaddress to make a reverse connection to the client. For further information on how to use the program from another client or other question regarding the cluster refer to Uppmax home page and the user guides found there [9].
Appendix D

A simple Chromium test script

# Chromium script example to make a non stereoscopic aware application
# produce stereoscopic output
# author Martin Ericsson

import sys
sys.path.append( "../server" )
from mothership import *
import cmatrix

Application = "StereoTestApp"
StereoMode = "SideBySide"

# parse args
args = sys.argv[1:] # skip program name
while len(args) > 0:
    arg = args[0]
    if arg == "-mode":
        StereoMode = args[1]
        args = args[2:] # chop off first 2 args
    elif arg[0] != "-":
        Demo = arg
        print "--> Note: you may have to tweak the config's stereo parameters!"
        args = args[1:] # chop off first arg
    else:
        print "Unknown option %s" % arg
        sys.exit(1)

# parameters
EyeSeparation = 0.325
Width = 2.0
NearClippingPlane = 1.0
FarClippingPlane = 100.0
FocalDistance = 4.0
# cromium reference

```python
cr = CR()
cr.MTU( 1024*1024 )
```

# set stereo mode

```python
if StereoMode == "SideBySide":
    TileWidth = 400
    TileHeight = 200
    Aspect = 1.0
else:
    TileWidth = 400
    TileHeight = 400
    Aspect = float(TileWidth) / float(TileHeight)
```

# Setup tilesort SPU

```python
tilesortspu = SPU("tilesort")
tilesortspu.Conf('bucket_mode', 'Test All Tiles') ##'Uniform Grid')
tilesortspu.Conf('force_quad_buffering', 1)
tilesortspu.Conf('stereo_mode', StereoMode)
```

# Setup app node w/ tilesort SPU

```python
clientnode = CRApplicationContext()
clientnode.StartDir( crbindir )
clientnode.SetApplication( Application )
clientnode.AddSPU( tilesortspu )
```

# Setup server node with render SPU

```python
renderspu = SPU('render')
renderspu.Conf( 'window_geometry', [0, 0, TileWidth, TileHeight] )
renderspu.Conf( 'fullscreen', 1)
servernode = CRNetworkNode()
servernode.AddTile( 0, 0, TileWidth, TileHeight )
servernode.Conf('optimize_bucket', 0) # just to be safe
```

# set protocol and default port

```python
tilesortspu.AddServer( servernode, protocol='tcpi', port=7000 )
```

# Need to specify the left and right view/projection matrices

```python
Width = 0.5 * Width # want half width below
s = NearClippingPlane / FocalDistance
```
top = s * Width
bottom = -top
for eye in range(2):
    # View matrix
    v = crmatrix.CRMATrix()
    if eye == 0:
        v.Translate(+EyeSeparation, 0, 0)
    else:
        v.Translate(-EyeSeparation, 0, 0)

    # Projection matrix
    p = crmatrix.CRMATrix()
    if eye == 0:
        left = s * ((Width * -Aspect) - EyeSeparation);
        right = s * ((Width * Aspect) - EyeSeparation);
    else:
        left = s * (Width * -Aspect + EyeSeparation);
        right = s * (Width * Aspect + EyeSeparation);
    p.Frustum(left, right, bottom, top, NearClippingPlane, FarClippingPlane)

    if eye == 0:
        servernode.Conf('view_matrix', v.ToList())
        servernode.Conf('projection_matrix', p.ToList())
    else:
        servernode.Conf('right_view_matrix', v.ToList())
        servernode.Conf('right_projection_matrix', p.ToList())

    # run
    cr.Go()