ViSTA: a multimodal, platform-independent VR-Toolkit based on WTK, VTK, and MPI

Thomas van Reimersdahl, Torsten Kuhlen, Andreas Gerndt,
Jörg Henrichs and Christian Bischof
Computing Center, Aachen University of Technology
Seffenter Weg 23, 52074 Aachen, Germany
{ reimersdahl, kuhlen, gerndt, henrichs, bischof } @rz.rwth-aachen.de

Abstract

In this paper we present our virtual reality software toolkit ViSTA (Virtual Reality Software University of Technology Aachen), which we offer as public domain WorldToolKit add-on. We discuss the integration of VTK into ViSTA as well as an environment for parallelized visualization algorithms of VTK, present our attempts to resolve remaining difficulties, and introduce first results.

1 Introduction to ViSTA

The decision for the development of ViSTA (Virtual Reality Software University of Technology Aachen) was motivated by the different VR activities of over 20 institutes at Aachen University of Technology (RWTH) and companies of the region. The VR focus at RWTH Aachen is mainly on the visualization of numerical simulations, virtual prototyping, architecture, medicine and psychology. The heterogeneity of applications and particularly hardware platforms used by the different institutes and companies made it necessary to find a VR toolkit which on the one hand is powerful enough to permit high quality, interactive scientific visualization in a virtual environment, and which on the other hand is as flexible and platform-independent as possible. Although very sophisticated and comprehensive VR toolkits have been developed so far (e.g. [RAN98, TRA99]), most of them are based on the SGI Performer library and can therefore only be executed on costly Silicon Graphics computers or at most LINUX platforms.

Since the development of a new, cross platform VR toolkit from scratch was not practicable and desirable, we decided to use the commercial C library WorldToolKit (WTK, Engineering Animation Inc.). Although its programming concept is not very progressive, WTK has the advantage of supporting a wide range of platforms like SGI Irix, HP UX, Sun Solaris, LINUX and even Win32. While WTK covers basic functions like scene graph rendering and interfaces to different display and interaction devices, ViSTA as a public domain WTK-add-on concentrates on higher functionality tailored to the needs of the research projects in Aachen, and of scientific and industrial applications in general.

Currently, four partners are involved in the development of ViSTA, namely the Computing Center Aachen, the Research Center Jülich, and the Institutes of Technical Computer Science and Technical Acoustics at RWTH. Figure 1 gives an overview of the ViSTA functionality under development. Of particular interest are the integration of multimodal interaction techniques, especially haptics and acoustics, and physically-based modeling of virtual environments with applications in assembly task simulations [STE98, STE99].

The remainder of this paper focuses on the visualization of numerical simulations by ViSTA. A major drawback of WTK is that it has been primarily designed for the consumer market rather than for scientific applications, so it does not provide any functionality for the visualization of complex scientific data. As this is one of our major working areas, we decided again to profit from existing powerful tools instead of re-implementing visualization algorithms from scratch. Since platform independence was our principal motivation to create ViSTA, we decided to integrate the open source C++ library Visualization Toolkit (VTK) into ViSTA.

While section 2 of this paper is about the integration procedure of VTK into ViSTA, section 3 discusses a concept for the

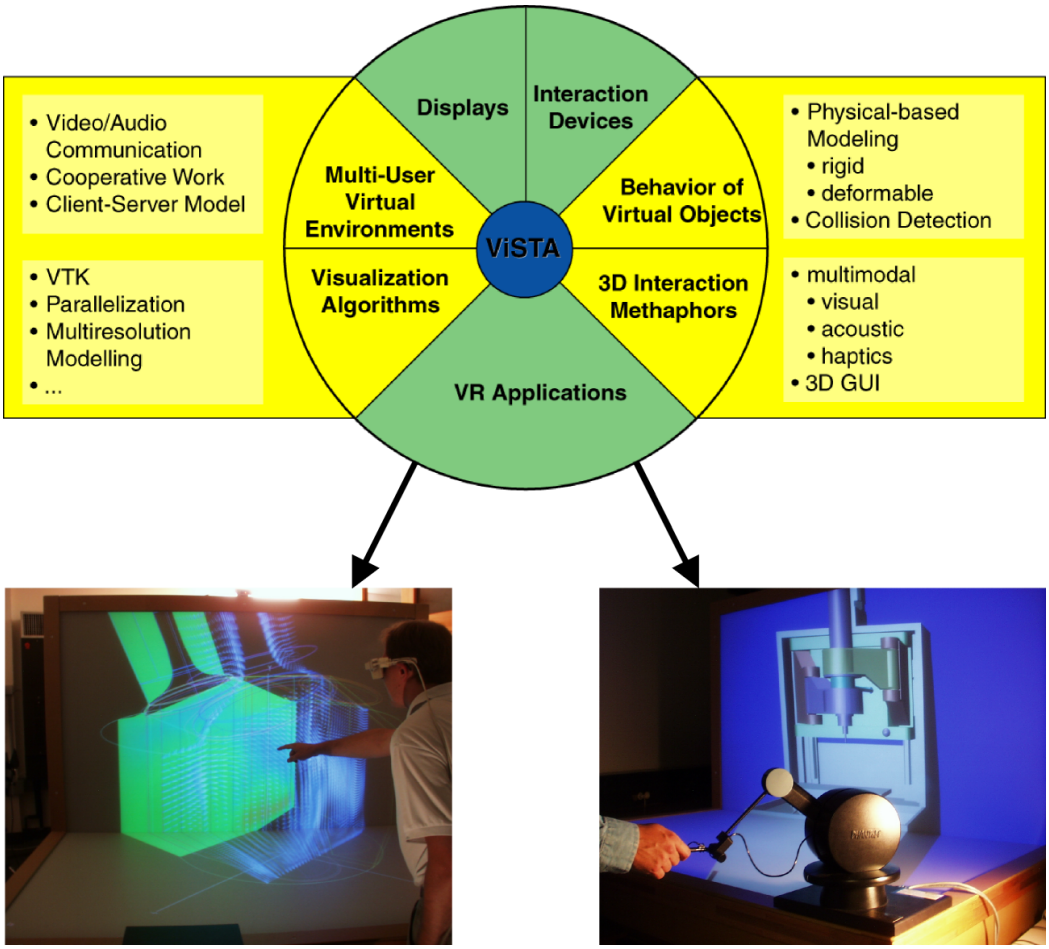

Figure 1: Functional view of ViSTA. The two examples of applications realized with ViSTA show a VR-based visualization of air flow into the cylinder of a combustion engine (left) and Virtual Prototyping of a parallel robot kinematics with force feedback (right).

parallelization of visualization work like, for example, the calculation of streamlines in CFD, in order to get the short response times needed for an interactive exploration of simulation results. We have realized a first test implementation of this concept on a PC cluster, using the cross platform Message Passing Interface (MPI) for the communication between single cluster nodes.

2 Integration of VTK in ViSTA

The Visualization Toolkit VTK [SCH98, VTK] is an established toolkit which contains many advanced algorithms for three-dimensional computer graphics, visualization and imaging. However, VTK does not offer any functions that relate to virtual reality. Though the option of rendering images in a stereoscopic way is supported, VTK does not offer any tracking functionality that enables viewer-centered projections, or 3-D interaction metaphors.

In the following subsections we firstly describe different methods for the integration of VTK into a VR software toolkit, compare them with regard to the feasibility within ViSTA, explain our decisions and consider some implementation details.

2.1 Known Methods

The so far published approaches of integrating VTK into a VR software toolkit have primarily been developed for the CAVE Library (CAVELib™, developed by the University of Illinois at Chicago and distributed by VRCO, Inc.) and can be categorized into two approaches, the actor transformation method and the renderer adaptation method.

Actor Transformation Method

The first approach is the Actor Transformation Method. Here, the VTK actors, the graphic objects that are being considered by VTK in the process of rendering, and their characteristics are transformed into graphical primitives of the virtual reality software toolkit.

One implementation of this method is the VtkActorToPF [RAJ] software package. VtkActorToPF contains two different execution models, an explicit and an implicit one. Both are intended for the use of VTK within Iris Performer, a proprietary rendering library distributed by SGI, and is provided in the CAVE Library. The VtkActorToPF translation function explicitly transforms any VTK actor with its geometry, color information and properties into a node within the performer scene graph. The implicit execution model is implemented by the VtkActorToPFTranslator class. Every instance of this class gets assigned a VTK actor. Then, this VTK actor is automatically translated by the VtkActorToPF function into a performer node. Whenever the special VTK actor changes, the corresponding performer node is automatically retranslated.

Another implementation is VTK2CAVE [HAL]. In contrast to the VtkActorToPF package, VTK2CAVE generates at the end of the translation process pure OpenGL commands within a special framework of the CAVE Library. Here, the VTK actors are directly transformed via a new vtkRenderer class, namely the cvtkCAVERenderer class. The translation process is invoked by adding a new VTK actor to the instance of cvtkCAVERenderer. The VTK actor is then inserted into a list which controls the actors to be rendered. In each frame update, the whole list is traversed, item by item, and translated into OpenGL statements, which are drawn instantly. By this way, the cvtkCAVERenderer is like an adapted VTK OpenGL renderer, from the view of the application programming interface.

Renderer Adaptation Method

The second approach, the Renderer Adaptation Method, integrates VTK by adapting the OpenGL renderer to the virtual reality toolkit. Vtkcave [MCS] belongs to this method. Here, the OpenGL renderer is nearly the same as in VTK. In vtkcave, two classes vtkCAVERenderer and vtkCAVERenderWindow are added which are derived from the vtkOpenGLRenderer class and the vtkOpenGLRenderWindow class. The key to this implementation lies in four statements in the start and initialization process of the render window. In detail, all the windowing calls that would have been done by the parent class, vtkOpenGLRenderWindow, are replaced. So, during the initialization process the Windows-Id, the Display-Id and the Context-Id are set properly to the ones of the CAVELib™ application, as well as the right Context-Id in the start process. Further, the frame function of the render window has to be overloaded by a no-operation statement, because the parent methods call a swap buffer, and the CAVELib™ is responsible for that. Finally, the camera based rendering is removed.

Comparison

All of these integration models have their advantages and disadvantages. All these conceptual designs are practical from an application perspective, because there are only few statements necessary to get live VTK in CAVELib™ applications. In our assessment we focus on the reusability of the models, on their ability to track changes of VTK and optimizing techniques.

The reusability of the integration models depends on the toolkits that are involved. VtkActorToPF is reusable within all software toolkits which provide an IRIS performer interface. Since Vtk2CAVE has its own OpenGL framework, it can be integrated into every software toolkit with an OpenGL interface. Only the shared-memory handling is specific to the CAVELib™ and has to be adapted. On the other hand, vtkcave is the most reusable implementation. It has only few CAVELib™ specific render window statements, which belong to the CAVELib™. However, vtkcave is directly applicable where VTK is using OpenGL.

The translation processes of VtkActorToPF and VTK2CAVE only take into account the VTK actors. Here, the VTK volumes are not involved, but in vtkcave they are. Furthermore, the cameras and lights of VTK are not considered in VtkActorToPF and VTK2CAVE, because the concepts in the CAVELib™ are used instead. In the vtkcave method the camera based rendering is removed, but the lights are used.

In any implementation we need to be able to track changes in VTK. In our view, this requires the least effort in the VtkActorToPF method. The vtkcave method will take more time, because the new class has to be completely merged with the old one, in the worst case. In addition, the changed OpenGL statements have to be extracted explicitly in the VTK2CAVE method because of its own OpenGL renderer.

In order to optimize the execution of the VTK pipeline and to enhance the degree of interactivity VtkActorToPF and VTK2CAVE offer a solution. In both methods, shared memory is used on which different processes operate. One separated process is responsible for the execution of the VTK pipeline and one other for the rendering process. When the

update process of the VTK pipeline has completed, the results are written into the shared memory. Then, the renderer takes and visualizes them. In this fashion, the time for obtaining the desired results is minimized in comparison with the sequential execution of the VTK pipeline and the rendering process.

In summary, we can recommend the VtkActorToPF method for the integration of VTK into another VR toolkit if there exists an IRIS Performer interface within it. In case there is only the OpenGL interface available however, the VTK2CAVE method is preferable to the vtkcave method because of performance issues. However, vtkcave is easier to adapt to another VR toolkit than the others, because there are only few lines of code to be modified.

2.2 Implementation

ViSTA is based on WTK, and with the current release 9 of WTK there exists for the first time the concept of OpenGL nodes in the scene graph. Thus, we only had the choice between adapting the VTK2CAVE or the vtkcave method.

We chose the approach of adapting the OpenGL renderer because of the simplicity of the implementation design and because it allows for a greater degree of flexibility in tracking VTK changes. The basis for our implementation was vtkcave. In adapting this approach, it is important to ensure that WTK and VTK have the same X11-context. In our case the X11-context was dictated by WTK. One problem is that both toolkits also possess their own OpenGL context and that the toolkits cannot simply interchange these contexts. In producing a common context, each of the existing WTK and VTK contexts must be examined carefully, and care must be taken to ensure that ViSTA's light sources remain the same. Moreover, when the actors are being selected, the sensor data still in WTK format must be transformed into related VTK coordinates so that the VTK picking method functions properly.

In course of time, we realized that from the view of software engineering the actor translation method better fits to the concept of scene graphs and object-oriented design. Hence, it is only important to know some fundamental methods, which are inherent for all actors. All actors own the method to draw themselves. Each specialized actor knows best how to draw all its graphical properties. But all actors can be triggered using a common public interface. In this case we use RenderOpaqueGeometry() and RenderTranslucentGeometry() to draw all kinds of actors. Only one method (namely MakeCurrent()) of the original VTK render window class must be overridden and then all refresh actors draw directly into the WTK window. For a common concept we implemented three additional classes (figure 2): COpenGLNode, IOpenGLDraw and CWtkVtkWindow. CWtkVtkWindow implements the abstract class IOpenGLDraw. The implemented method DrawDisplayObjects() invokes the drawing methods of all used actors.

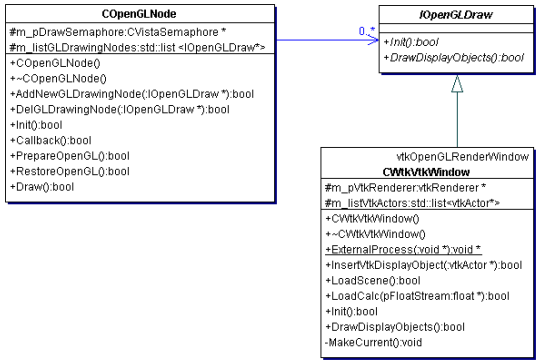

Figure 2: vtkActor translator design concept for WTK

We use the abstract class IOpenGLDraw to implement a concept for our own software ViSTA. It hides the specific realization of all objects that are derived from this interface. These objects can be pure OpenGL objects, some kind of VTK actors, or something totally different, as long as the underlying OpenGL-API is satisfied. An object of the class COpenGLNode collects all objects derived from this interface. COpenGLNode inserts a new specialized WTK OpenGL node into the scene graph of WTK. It consists of a callback that traverses an internal list to invoke all stored objects that own the IOpenGLDraw interface.

3 Parallelization of the VTK interface

3.1 Design Concept

The time necessary for computing the different views using different algorithms (streamlines, isosurfaces [BAJ99, TAN96], …) is an important factor for user-acceptance of a virtual reality toolkit. Large data sets and/or complex algorithms need a large amount of computing time, which makes interactive use of visualization applications nearly impossible. Since memory and computing power of a single machine are insufficient for many applications, we have decided to employ

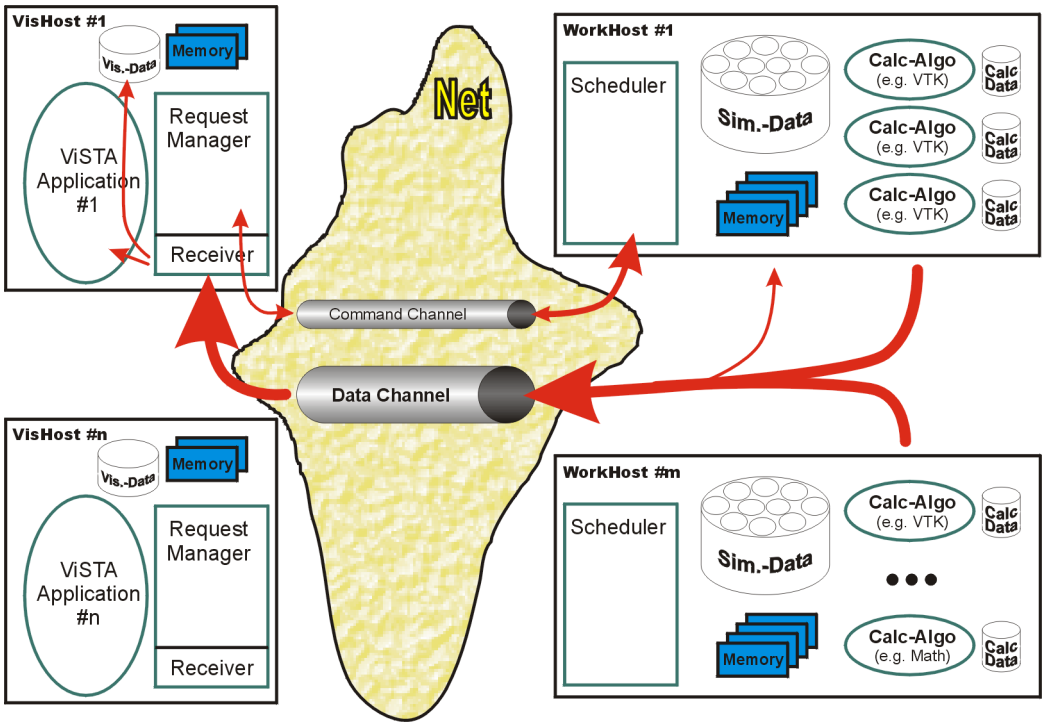

Figure 3: important components and data flow

parallel computing to speed up ViSTA. The time consuming tasks are processed on dedicated parallel machines, so called *work hosts*, while the result is visualized on *visualization hosts*, where specialized graphics hardware is available.

While designing the parallelization concept, two different kinds of users have been considered: first the application developer, who employs ViSTA to develop applications which visualize data, and second the end user, who uses these applications. The application developer wants to focus on the application and not on the complex parallelization aspects. Therefore, the parallelization aspects should be hidden from the developer. Furthermore, it was our goal that ViSTA should still be able to work as a sequential application, since some applications might not at all benefit from parallelization, e.g. because the response time of a single machine might be quick enough, and since not every potential end user or application developer has access to a parallel machine. Additionally, developing and testing on a single machine (with smaller data sets) is much more efficient than on a distributed system.

The end user on the other hand wants smooth interaction and quick responses to her requests. The same application, which was developed with ViSTA, might be run on either a sequential or on a parallel machine, depending upon the size of the data set.

Furthermore, imagine several end users at different places working together at the same time in analyzing one data set. Since we consider this kind of cooperative work as important applications for the future, ViSTA should support developing such multi-user virtual environments. It is the task of the application developer to decide how cooperation between the end users should be supported by the application and to actually synchronize the different visualization hosts. As will be shown in the next section, ViSTA nevertheless contains some additional internal synchronization to avoid creating bottlenecks or situations, in which one user monopolizes the work hosts, so that other users cannot work.

3.2 Design

Figure 3 shows the parallelization concept of ViSTA. On the left hand side, the visualization hosts are shown. Each of these hosts runs a copy of the same ViSTA application. As a reaction to user commands e.g. a request to compute and display streamlines at specified points, a *request* is created within ViSTA. Each of these *requests* is assigned a priority, which will actually determine how fast a request is processed. This *request* is then passed on to a *request manager*, which chooses one of the work hosts, which are depicted on the right hand side of figure 3 for completing the *request*. The *request manager* is an internal part of ViSTA. These *request managers* have to synchronize with each other, so that a single

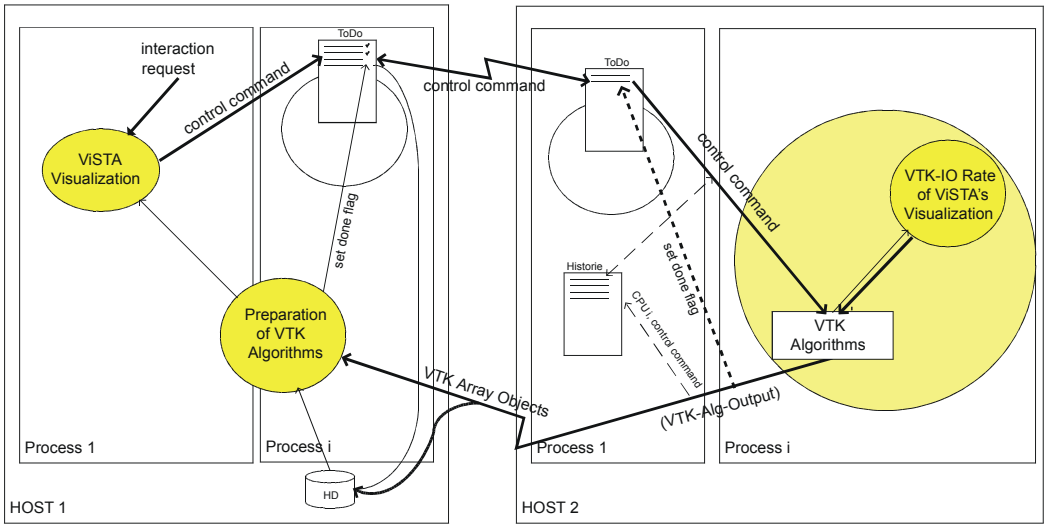

Figure 4: advanced control flow

work host is not overloaded with requests, while other work hosts are idle.

The *request* is then forwarded to the chosen work host, where it is received by a *scheduler*. This *scheduler* selects a minimum and maximum number of nodes, which are to be utilized for the given *request*. These numbers depend on factors like computational speed, available memory and the capacity of the network of the machine. Algorithms might actually slow down if too many nodes are used or if the network is too slow, so the number of nodes to use to fulfill a given *request* depends on the machine, on which the *request* is executed.

The *request* is then inserted into the work queue of the *scheduler*, which is sorted with descending priorities. As soon as a sufficient number of nodes for the *request* with the highest priority are available, the scheduler selects the actual nodes (up to the maximum number assigned to the request), which will process the *request*. The selected nodes then start computing, and one of the nodes will send the result to the *receiver* on the visualization host, which sent the *request*. The *receiver*, which is the last part of ViSTA, is responsible for passing the result to the application.

This concept makes use of two different channels, as depicted in figure 3. The *command channel*, through which the requests are passed from the *request manager* to the work hosts, needs only very little bandwidth, since the messages sent along this channel are rather small (typically less than a hundred bytes). On the other hand, the results, which were computed on the work hosts, are quite large, up to several mega bytes. At this point, a network with a high bandwidth is necessary.

The concept as described offers one potential optimization feature: *requests* might be computed in advance, preliminary to the actual user request. Even for large data sets and complex algorithms the work hosts will not be busy all the time, since the user will need some time to analyze the displayed data. During this time, *requests* can be computed in advance and then be cached on the visualization hosts. If a *request manager* receives a *request*, it first checks with its local cache, whether this request has already been computed. If so, the result is taken from the cache and immediately displayed. For this optimization, *requests* with a very low priority can be generated. On the other hand, it is necessary to suspend any process working on any low priority *request*, when an actual *request* from the end user arrives.

Figure 4 shows in more detail the data flow when using one visualization host and one (parallel) work host.

3.3 Implementation

A first prototype implementation was implemented using the Message Passing Interface (MPI) as communication library [MPI, MPICH, OMP, GRO95]. The prototype only supports one visualization host and one work host. This allowed us to quickly code and test the described concept. A drawback of this approach is the necessity to link the MPI library to the application, even for sequential applications. For the final version of ViSTA, we consider using a specialized TCP/IP protocol for the communication between the visualization hosts and the work hosts.

The big challenge for the usability of this concept is the parallelization of the calculation procedures such as the VTK algorithms. This

is currently being undertaken in a collaboration of Kitware Inc. and the Los Alamos National Lab (LANL) as well as Kitware Inc. and the Argonne National Lab (ANL) (Kitware Inc. press releases: [VTK]). We could immediately make use of parallel version of any VTK algorithm.

For this prototype we implemented a simple parallel function for computing streamlines. The complete data set, which is to be visualized, is read on each node of the parallel work hosts. The computation of streamlines is then divided equally on the available nodes, where the result is computed independently of the other nodes. Since the visualization host expects exactly one result for each *request*, the computed streamlines are combined on one node and then sent to the visualization hosts. Work in the area of meta-computing has shown that it might actually reduce communication time when messages over a slower network are combined [HEN98].

This simple design was feasible, since the test cases we used were small enough, so that the pre-computed data can be read completely on one node. In general, the whole raw data set is required for the calculation, which might be too large for one node. Thus fragmentation strategies for each specific occasion have to be found (e.g. [BAJ99]).

The *request manager* and *receiver* on the visualization hosts are implemented as separate threads, which communicate via shared memory.

3.4 Demo Application

The visualization toolkit (VTK) includes some demo data sets. One of these examples is called the kitchen demo. It contains the result of a computational fluid dynamics (CFD) analysis of the air circulation inside a kitchen. It is a small structured grid with a dimension of 28 x 24 x 17 points. The visualized data set contains scalar values that indicate the pressure at certain positions. The data set is by courtesy of Dr. L. Besse from ETH, Zurich.

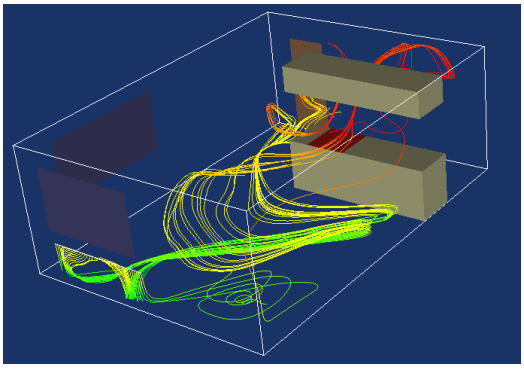

Figure 5: the kitchen demo

Based on that demo data set we have constructed an application to test the scalability and performance of our implementation of a parallelized ViSTA concept. We used the kitchen CFD data set to calculate and visualize streamlines (figure 5).

3.5 Test Environments

The primary goal was to separate the system that should execute the visualization and the system, which is optimized for parallel computation. In our standard stand-alone VR environment we use a high performance graphics workstation of SGI, the Onyx-2 Infinite Reality 2 (4 MIPS 10000, 195 MHz, 2 GByte memory, 1 graphics pipe, 2 raster managers), which should finally be used as our visualization engine.

This system was coupled to the Siemen's hpcLine here at the Computing Center, a Linux cluster with 16 standard PC nodes (each consists of two Intel-PII processors, 400 MHz, 512 KByte level-2 cache, 512 MByte system memory), which are connected via a high performance network (SCI network, Scali GmbH) [SCI]. This Linux cluster can reach 12.8 Giga flops [HPCL].

The parallel version was compared to sequential versions for the Onyx and Linux. We could only measure the time needed for actually computing the data, since the WTK library is not yet available for Linux. Furthermore, we used a high performance PC graphics workstation designed by Intergraph. The Intergraph's TDZ 2000 GT1 specification is dual processor Xeon 500 MHz, 1 GByte system memory and two Intense 3D Wildcat 4000.

To determine the impact of the network bandwidth, we used different MPI implementations. On one hand we used the native MPI implementation on the SGI and the hpcLine (ScaMPI), which offers a peak bandwidth of 95 / 80 MBytes/s respectively. But it is not possible to let these different libraries work together to couple an application on both platforms. Therefore, we used MPICH, which was developed by the Argonne National Laboratory (ANL). MPICH is a free MPI implementation available for many platforms, in particular for each of our target architectures, and supports data conversion using XDR. Using MPICH it was possible to couple different machines and let them work together. Since MPICH does not support the SCI network, the internal bandwidth of the hpcLine was reduced to about 10 MBytes/s. The Onyx and the hpcLine are connected with a 100 MBits/s fast Ethernet.

3.6 Results

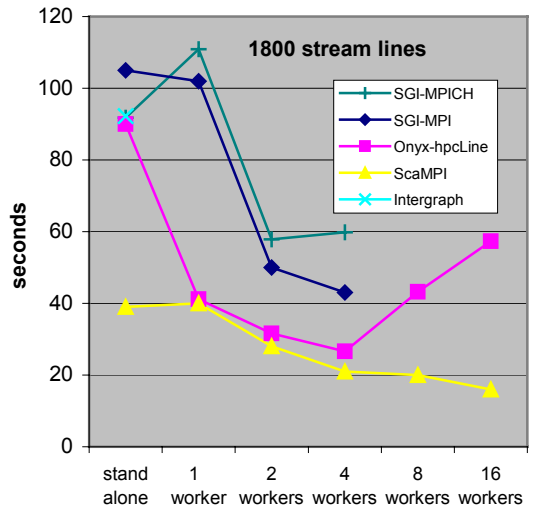

Figure 6: comparison of run time results

Figure 6 shows the result of the kitchen application when computing 1800 streamlines. Given is the time between initiating the event at the visualization hosts and receiving the results at the same host (remember that we could not actually visualize the result on the hpcLine). The number of workers does not include the scheduler, which was additionally running.

The first result is, that the Linux cluster is much faster in computing the results than the SGI. This supports the claim of the so called meta-computing, where hosts of different architectures work together to solve one problem [HEN98]. Both machines show a good speed up, if they are used alone.

Of course, using only one worker does not achieve any speed up (compared to the sequential run), since the sequential application does not do any work while the streamlines are computed, so no work is done in parallel. But when more workers are used, the actual run time is reduced.

The influence of a fast network can be seen when comparing the result of the hpcLine alone and the coupled system SGI/hpcLine. On the coupled system, MPICH was used, so that only the slower fast Ethernet was used to transmit the data on the work host. As a result, a slow down was reached when 8 or more workers where used. This supports the importance of fast networks and the design issue that the scheduler on each work hosts decides on the number of nodes to use for a given request.

4 Conclusions and Future Work

The integration of VTK into our virtual reality software ViSTA as well as the development of a flexible parallel environment for visualization tasks are ongoing work, which should lead to a comprehensive VR toolkit being capable to handle complex data sets interactively. In comparison to existing VR toolkits, our efforts are towards a scalable cross platform tool running on “off the shelf” parallel hardware like PC clusters.

While our work has been on the aspect of rendering VTK scenes in a virtual environment so far, our current activities concentrate on interaction aspects, i.e., the use of VTK functionality from within the virtual environment. The general master-worker concept for parallelization we presented in this paper is in the implementation phase right now. The first simple prototype we have implemented so far shows promising run time results and emphasizes the importance of fast network components between parallel nodes. Other work in process is about the support of more platforms. For instance, we recently started tests with a MPICH version for the Win32 platform [WMPI].

References

[BAJ99] C.L. Bajaj, V. Pascucci, D. Thompson, X.Y. Zhang (1999). “Parallel Accelerated Isocontouring for Out-of-Core Visualization”. Parallel Visualization and Graphics Symposium, Oct. 25-26, 1999, San Francisco, California.

[DER99] F. Deriggi Jr., M. Kubo, A. Sementille, J. Brega, S. Santos, C. Kirner (1999). “CORBA Platform as Support for Distributed Virtual Environments”. Proceedings of IEEE Virtual Reality 1999 Conference, March 13-17, Houston, Texas, pp. 8-13.

[GRO95] W. Gropp, E. Lusk, A. Skjellum (1995). „Using MPI – Portable Parallel Programming with the Massage-Passing Interface”. MIT Press, Cambridge, Massachusetts.

[HEN98] J. Henrichs (1998). “Optimizing and Load Balancing Metacomputing Applications”. Proceedings of the International Conference on Supercomputing (ICS-98), pp. 165-171, ACM press, July 13-17 1998. (also available as technical report FZJ-ZAM-IB-9805, Research Centre Jülich, ZAM, August 1998).

[RAN98] D. Rantzau, K. Frank, U. Lang, D. Rainer, U. Wössner (1998). “COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data”. Proceedings of the 2nd Workshop on Immersive Projection Technology (IPT '98), Ames, Iowa 11.-12. May 1998.

[SCH98] W. Schroeder, K. Martin, B. Lorensen (1998). „The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics“, Prentice Hall.

[STE98] R. Steffan, T. Kuhlen, A. Loock (1998). „A Virtual Workspace Including a Multimodal Human Computer Interface for Interactive Assembly Planning." IEEE International Conference on Intelligent Engineering Systems (INES'98), Vienna.

[STE99] R. Steffan, T. Kuhlen, F. Broicher (1999). „Design of multimodal feedback mechanisms for interactive 3D object manipulation." In: Bullinger, H.-J. and Ziegler, J. (Eds.): Human-Computer Interaction: Ergonomics and User Interfaces, Vol. 1 of the Proc. of the 8th International Conference on Human-Computer Interaction, Munich, pp. 461-465, Lawrence Erlbaum Associates, Inc. Mahwah, New Jersey.

[TAN96] N. Tang, T.S. Newman (1996). „A vector-parallel Realization of the Marching Cubes", Technical Report: TR-UAH-CS-1996-01: UAH Computer Science Department, University of Alabama, Huntsville, ftp://concept.cs.uah.edu/techreports/

[TRA99] Henrik Tramberend: "Avocado: A Distributed Virtual Reality Framework", Proceedings of IEEE Virtual Reality 1999 Conference, March 13-17, 1999, Houston, Texas, pp. 14-21

Web Resources

[HAL] M. Hall, NCSA Visualization and Virtual Environments, University of Illinois. http://zeus.ncsa.uiuc.edu/~mahall/

[HPCL] http://www.rz.rwth-aachen.de/hpc/hpcLine

[MCS] MCS Division at Argonne National Laboratory, University of Chicago. http://www-fp-mcs-anl.gov/fl/software/index.htm, ftp://ftp.mcs.anl.gov/pub/futures-lab/software/vtkcave.tar.gz

[MPI] http://www.mcs.anl.gov/mpi/index.html

[MPICH] http://www.mcs.anl.gov/mpi/mpich

[OMP] http://www.openmp.org

[RAJ] P. Rajlich, NCSA Visualization and Virtual Environments, University of Illinois. http://brighton.ncsa.uiuc.edu/~prajlich/vtkActorToPF/

[SCI] http://www.scali.com/

[VTK] http://www.kitware.com

[WMPI] WMPI: http://dsg.dei.uc.pt/wmpi, MPICH-NT: http://www.mcs.anl.gov/~ashton/mpich.nt

[WTK] http://www.sense8.com

[ZAM] Central Institute for Applied Mathematics, Research Center Juelich, http://www.kfa-juelich.de/zam/