CUMULVS Viewers for the ImmersaDesk*

Torsten Wilde, James A. Kohl, and Raymond E. Flanery, Jr.
Oak Ridge National Laboratory

Keywords: Scientific Visualization, CUMULVS,
ImmersaDesk, VTK, SGI Performer

Abstract. This paper will discuss the development of CUMULVS "viewers" for virtual reality visualization via ImmersaDesk/Cave systems. The CUMULVS (Collaborative, User Migration, User Library for Visualization and Steering) system, developed at Oak Ridge National Laboratory, is a base platform for interacting with high-performance parallel scientific simulation programs on-the-fly. It provides run-time visualization of data while they are being computed, as well as coordinated computational steering and application-directed checkpointing and fault recovery mechanisms. CUMULVS primarily consists of two distinct but cooperative libraries - an application library and a viewer library. The application library allows instrumentation of scientific simulations to describe distributed data fields, and the viewer library interacts with this application side to dynamically attach and then extract and assemble sequences of data snapshots for use in front-end visualization tools. A development strategy for integrating CUMULVS with the virtual reality visualization libraries and environments will be presented, including discussion of the various data transformations and the visualization pipeline necessary for converting raw CUMULVS data into fully rendered 3-dimensional graphical entities.

1. Introduction

Scientific simulation continues to proliferate as an alternative to expensive physical prototypes and laboratory experiments for research and development. Scientists often run software simulations on high-performance computers to obtain results in a fraction of the time, or at a higher resolution, versus traditional mainframe systems, PCs or workstations. Software experiments provide a cost-effective means for exploring a wide range of input datasets and physical parameter variations, but much infrastructure is required to enable the development of these parallel and distributed computer simulations. Teams of scientists often need to observe the ongoing progress of the simulation and share in the coordination of its control.

* Research supported by the Mathematics, Information and Computer Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

It is useful to visualize the ongoing simulation and interact with it at runtime, especially if the simulation runs for several days. For example, it could save time and money to discover that a simulation is heading in the wrong direction due to an incorrect parameter value, or because a given model does not behave as expected.

A proper visualization environment allows scientists to view and explore the important details of a simulation data set. For a 2-dimensional (2D) data set, a 2D visualization environment is sufficient. But for 3-dimensional (3D) problems, a 3D visualization environment is required to provide the best access to *all* information embedded in the data set. One of the most sophisticated ways to interact with a simulation in 3D is via a virtual reality environment. 3D vision is one of the core human senses, and scientists can use virtual reality to intuitively interact with a simulation using a fabricated “real-world” interface.

This paper describes work to build a 3D virtual reality viewer environment for high-performance applications that use CUMULVS as the interface to application data. The ImmersaDesk™ was chosen as the target display environment because it is less expensive and requires less space than a fully immersive CAVE™ environment. Yet the ImmersaDesk uses the same library interface as the CAVE™, so graphical views for an ImmersaDesk™ can also be displayed on a CAVE™ without modification. A variety of visualization software libraries are required to translate and convert raw data from CUMULVS into a form viewable on the ImmersaDesk™. Section 3 will describe these design choices, after a brief background section. Sections 4 and 5 present results and future work, respectively.

2. Background

2.1. CUMULVS

CUMULVS (Collaborative, User Migration, User Library for Visualization and Steering) [9,10] provides a fundamental platform for interacting with running simulation programs. With CUMULVS, a scientist can observe the internal state of a simulation while it is running via online visualization, and can “close the loop” and redirect the course of the simulation using computational steering. These interactions are realized using multiple independent front-end “viewer” programs that dynamically attach to, interact with and detach from a running simulation as needed. Each scientist controls his/her own viewer, and can examine the data field(s) of choice from any desired perspective and at any level of detail. A simulation program need not always be connected to a CUMULVS viewer; this proves especially useful for long-running applications that do not require constant monitoring. Similarly, viewer programs can disconnect and re-attach to any of several running simulation programs. To maintain the execution of long-running simulations on distributed computational resources or clusters, CUMULVS also includes an application-directed checkpointing facility and a run-time service for automatic heterogeneous fault recovery.

CUMULVS fundamentally consists of two distinct libraries that communicate with each other to pass information between application tasks and front-end viewers.

Together the two libraries manage all aspects of data movement, including the dynamic attachment and detachment of viewers while the simulation executes. The application or “user” library is invoked from the simulation program to handle the application side of the messaging protocols. A complementary “viewer” library supports the viewer programs, via high-level functions for requesting and receiving application data fields and handling steering parameter updates.

The only requirement for interacting with a simulation using CUMULVS is that the application must describe the nature of its data fields of interest, including their decomposition (if any) across simulation tasks executing in parallel. Using calls to the user library, applications define the name, data type, dimensionality / size, local storage allocation, and logical global decomposition structure of the data fields, so that CUMULVS can automatically extract data as requested by any attached front-end viewers. Given an additional periodic call to the `stv_sendReadyData()` service routine, CUMULVS can transparently provide external access to the changing state of a computation. This library routine processes any incoming viewer messages or requests, and collects and sends outgoing data frames to viewers.

When a CUMULVS viewer attaches to a running application, it does so by issuing a “data field request,” that includes a set of desired data fields, a specific region of the computational domain to be collected, and the frequency with which data “frames” are to be sent back to the viewer. CUMULVS handles the details of collecting the data elements of the subregion, or “view region,” for each data field. The view region boundaries are specified in global array coordinates, and a “cell size” is set for each axis of the data domain. The cell size determines the stride of elements to be collected for that axis, e.g. a cell size of 2 will obtain every other data element. This feature provides more efficient high-level overviews of larger regions by using only a sampling of the data points, while still allowing every data point to be collected in smaller regions where the details are desired.

Fig. 1. Data space of the viewer test application

2.2. Test Application

The simple test application for our viewer development calculates an expanding gas cloud. *Figure 1* shows the data space for this problem, which uses a rectangular mesh data topology. Potential visualization techniques for this kind of problem would include either data point visualization, contour rendering or volume rendering. For

point visualization, each element of the data set is represented by a distinct geometrical object (such as a sphere). The properties of each object depend on the data values for each corresponding element, e.g. the color of each sphere can represent the temperature value at each point. This technique applies well to small data sets but requires significant computational power for larger data sets. Contour rendering visualizes the surface of an object, e.g. an isosurface can be drawn for a chosen contour value, identifying uniform values within the data set. Volume rendering additionally renders the interior of such contoured objects. Volume rendering is important for translucent objects, like water or gas clouds, where scattered light passing through and reflecting from the data can reveal various internal characteristics. Here, changing properties inside the data field need to be considered for proper rendering. The best visualization techniques for our test application would be either volume or contour rendering; we have chosen contour rendering for our experiments due to constraints in the available visualization libraries (Section 3.2).

3. CUMULVS Viewer Design for Immersive CAVE Environments

The goal of this work was to visualize data sets collected by CUMULVS in an immersive environment such as the ImmersaDesk or CAVE. CUMULVS already supports several graphical viewers, using AVS5 and Tcl/Tk, so a true 3D virtual reality viewer would be a useful addition. Because the ImmersaDesk used for this development was driven by an SGI Workstation, our viewer has been designed to run on the SGI Irix operating system.

Fig. 2. Basic application structure

3.1. Viewer Concept

Our basic viewer structure is shown in *Figure 2*. Depending on interaction from the user, the “Data Collector” requests the desired data set from the user application. This raw data is then converted into the data format as needed by the “Visualizer”. In the Visualizer the data is converted into a useful visualization stream for the “ImmersaDesk” where the information is finally displayed.

To improve control over the viewer, the user can interact with the data collector, e.g. change the boundaries of the collected data fields, or change the frequency of data

collection. This type of control will be referred to here as “User Control Commands.” On the ImmersaDesk, the user can additionally interact directly with the virtual environment, e.g. zooming in/out of the data set and rotating the data set, independent of the ongoing data collection.

3.2. Viewer Pipeline

The purpose of the visualization pipeline is to create a scene, from the collected data sets, that can be properly interpreted by the ImmersaDesk (see *Figure 3*). The CAVE library [1], which is used to access the ImmersaDesk, recognizes OpenGL graphics primitives so the scene *could* be created in OpenGL. The problem with OpenGL, however, is that such programming is at the lowest abstraction level. The programmer has more flexibility in this format, but this approach adds significant time and complexity to the viewer development. For our development this is not cost effective, so we chose to utilize a higher-level toolkit with more built-in graphical objects to construct our viewer.

Fig. 3. Basic visualization pipeline

With the increasing demand for data visualization, many so-called “visualization toolkits” have been developed, like VTK (Visualization Toolkit) from Kitware [3,5], WTK (World Tool Kit) from Sense8 [12], Open Inventor [13] and Java3D [14]. These toolkits help to create visualization pipelines by providing libraries for high-level graphics programming, or using direct visual programming. Two visualization toolkits that provide an applications programming interface (API), and are widely used, are VTK and WTK. VTK is freeware but doesn’t support multi-process rendering to multiple channels, which is required for a CAVE environment. WTK is a commercial product that supports immersive environments in its latest version. Because our resulting CUMULVS viewer will be freely distributed, and could be used as a template for the creation of other viewers, our solution must avoid use of commercial products. Therefore VTK was chosen as our base visualization system.

However, without multi-process rendering to multiple channels, a 3D virtual reality (VR) scene can’t be directly created for a CAVE environment using VTK. To solve this problem, SGI Performer [8] was integrated into our visualization pipeline. SGI Performer is a high-performance rendering package for SGI workstations with the capability for multi-process rendering to multiple channels. To translate VTK graphical objects into a form usable by SGI Performer, another tool called *vtkActorToPF* [4] is applied to convert a VTK Actor into an SGI Performer node. Using a special library called *pfCAVE* [2,6,7], SGI Performer can interface with the CAVE library, thereby completing our visualization pipeline!

The final structure of the Visualizer is shown in *Figure 4*. Raw data is converted into a VTK data structure (see section 3.3) in the “data converter.” Next the visualization data is created, as a `vtkActor`, using VTK visualization techniques. The `vtkActor` data is converted into SGI Performer node data using `vtkActorToPF`. The view scene is created and rendered using SGI Performer and `pfCAVE`. Finally the virtual scene is displayed on the ImmersaDesk.

Because `vtkActorToPF` converts only a `vtkActor` into SGI Performer node data, we must use contour rendering as our visualization technique. VTK uses a different actor, “`vtkVolume`”, to store volume-rendered information, and no appropriate translator from `vtkVolume` to SGI Performer currently exists. Therefore volume information cannot yet be transferred to SGI Performer in this way.

Fig. 4. Final visualization pipeline

3.3. CUMULVS to VTK Data Conversion

Internally, VTK data types are derived from the organizing input data structure, and specify the relationship between cells and points [see also 11]. The first criterion is whether the structure is regular, i.e. has a single mathematical relationship between all composing points and cells, or irregular. Regular, or structured, data has a rectangular geometry and topology. The data in our test application are arranged on a rectangular lattice. The rows, columns and planes of the lattice are parallel to the global x-y-z coordinate system. The topology is built from voxels (cubes) that are arranged in a grid structure. Therefore our CUMULVS data set naturally maps to a structured point data set in VTK. If the data set topology and geometry of the source data set were unknown, it would be possible to use the unstructured grid data set as a general source object. But using this object requires more memory and computational resources. This is a motivation for writing individual viewers for different visualization problems, because it is very important in real time visualization to get the best possible performance. The structured point data set in VTK is represented using the data dimension, an origin point and the data spacing. The dimension of data is a 3-vector

(n_x, n_y, n_z) , specifying the number of points in the x, y and z directions. The origin point is the position in 3-dimensional space of the minimum x-y-z coordinate. Each voxel is identical in shape, with the spacing vector specifying the length in each of the x-y-z directions. VTK uses “Column Major” or FORTRAN storage order to store data elements. Because CUMULVS allows viewers to specify the target storage order of the data set to be collected, it is possible to correctly visualize data from both C and Fortran programs using our resulting viewer.

After determining the topology and size of the application data structure and creating a corresponding VTK equivalent, the data from the CUMULVS data collector can then be transferred into VTK. Due to the internal data organization in VTK, the CUMULVS data elements must be copied one-by-one into the VTK structured point data set. The data dimension and spacing must also be properly set to allow correct interpretation of the VTK data. The CUMULVS “cell size” regulates the number of collected data points along each axis. If we take a 1-dimensional data field with the boundaries $\min_x = 1$ and $\max_x = 10$, and a cell size $c_x = 1$, all 10 data points are collected. Converting this data into VTK, the dimension would be $(\max_x - \min_x + 1) = 10 = n_x$ and the spacing would be 1 because the number of data points is the same as the field size. If the CUMULVS cell size is changed to $c_x = 2$ then only half the data are collected, skipping every other element. In this case, VTK must have a dimension of $n_x = 5$ to account for the spacing of 2 (spacing equals the CUMULVS cell size). To solve this transformation problem in general, the VTK data dimension is calculated depending on the CUMULVS cell size for each data dimension, e.g. n_x / c_x . Because VTK requires an integer value as dimension the *ceiling* of this result is used. This same principle is applied along each axis for multi-dimensional spaces. Given this mapping the scalar data can be copied from CUMULVS directly into a `vtkScalarArray`, which is a VTK object for storing an array of scalars. After all the CUMULVS data is copied into it, the `vtkScalarArray` object is directly linked into the `vtkStructuredPoints` source object.

3.4. Final Viewer Structure

The final viewer structure is implemented in a single process as shown in *Figure 5*. The main event loop calls all handlers in a specific order. First the data set is collected, and then the visualization pipeline is executed. After that, any User Control Commands are handled and then the visualization handler is called again. This is done to improve the smoothness of the virtual reality interface.

4. Results

Figure 6 shows a sample view from the resulting viewer. To visualize our test application, nearly ½ million data elements are processed. The coloring of the sphere is done using a VTK “Elevation Color Mapper” that produces a color gradient based on the distance from the sphere origin. Otherwise the color of the entire sphere would be uniform. Unfortunately, the long visualization pipeline has some drawbacks in

terms of the interactive response time when using this model. The time taken to handle the various visualization and rendering tasks creates delays in displaying view frames and reacting to user control requests. The most time consuming steps of the pipeline are copying the CUMULVS data into the VTK data structure and conversion into the proper formats for visualization. Because User Control Commands for data collection are separated from the visual interaction, multiple handler routines have to be called in the main viewer event loop. User interactivity was improved by adding a special User Control Command that stops the pipeline execution after the data set is collected and visualized. This makes it possible for the user to examine and manipulate the data set on the ImmersaDesk without interruption. A button press triggers collection of the next data set and restarts execution of the visualization pipeline. The paused mode can be repeatedly enabled using the User Control Command interface.

Fig. 5. Final Viewer Structure

5. Summary/Future Work

Overall, this viewer is a good starting point for applying data collected by CUMULVS for use with custom 3D virtual reality viewers in CAVE-like environments. The essential portions of the visualization pipeline, including CUMULVS, VTK, vtkActorToPF and pfCAVE, are shown in implementation and behavior. The implemented visual interface is sufficient for most simple 3D visualizations and can be customized and adapted to a specific user application as needed. The use of VTK, vtkActorToPF, SGI Performer and pfCAVE as a

visualization pipeline is a cost effective way to develop a 3D VR viewer using CUMULVS as a data collector.

Fig. 6. Screenshot of the CAVE simulator

There are several advantages to using VTK. It is “inexpensive” (VTK and `vtkActorToPF` are freeware, and `pfCAVE` comes with the CAVE library). It also enables fast prototyping, i.e. new viewers can be created by swapping out VTK objects in the visualization pipeline. On the other hand, the pipeline has a relatively long execution time – mostly due to the data conversions between different entities. Volume rendering is not currently possible because no translator from a `vtkVolume` to a SGI Performer scene node yet exists. Further, in order to solve specialized visualization problems, deep knowledge is required of all parts of the pipeline, including possible interactions. The documentation of the freeware is sometimes insufficient for new and inexperienced users, exacerbating this problem.

To illustrate the adaptability of our 3D CUMULVS viewer as a viewer template, a CUMULVS viewer for pure VTK was created at the end of this project. Using the VTK `MyWindowRendererInteractor` class provided by Mark Hoebeke, the visualization using `pfCAVE` was rewritten to support display using VTK only. The advantage of this viewer is that no expensive hardware is required, and because of the common use of VTK this viewer variation is a useful addition to the CUMULVS software package. The disadvantage of this viewer is that, because VTK is used at several places in the visualization pipeline, user visual interaction is currently only possible with data collection disabled.

Several improvements could be made to the viewer design in order to improve real time visualization and make the user interaction smoother. The structure could be converted from single process into multi-process, where the individual processes could execute independently. Distinct processes could be created for CUMULVS data collection, the steps of the visualization pipeline and the display on the ImmersaDesk. The problem here is that a communication interface for exchanging objects and/or data is required. Other improvements could be made to optimize the copying of the

CUMULVS data into VTK. Furthermore, the User Control Commands could be integrated into the visual user interface, eliminating one handler invocation in the viewer event loop.

Appendix A – References

- [1] David Pape, Carolina Cruz-Neira, Marek Czernuszenko, “*CAVE User’s Guide, CAVE Library version 2.6*”, online documentation from the Electronic Visualization Laboratory, University of Illinois at Chicago, USA, 1997
- [2] D. Pape, “*pfCAVE CAVE/Performer Library (CAVELib Version 2.6)*”, online documentation from the Electronic Visualization Laboratory, University of Illinois at Chicago, USA, 1997
- [3] Will Schroeder, Ken Martin, Bill Lorensen, “*The Visualization Toolkit an object-oriented approach to 3D graphics*”, 2nd Edition, Prentice Hall PTR, 1998
- [4] Paul Rajlich, “An Object Oriented Approach To Developing Visualization Tools Portable Across Desktop And Virtual Environments”, M.Sc. Thesis University of Illinois, 1995 *vtkActorToPF*, <http://hoback.ncsa.uiuc.edu/~prajlich/vtkActorToPF>
- [5] Kitware, *VTK*, <http://www.kitware.com>
- [6] CAVERNUS user group, CAVE Research Network Users Society, <http://www.ncsa.uiuc.edu/VR/cavernus>
- [7] Dave Pape, Online resort for CAVE programming, <http://www.evlweb.eecs.uic.edu/pape/CAVE/prog>
- [8] Silicon Graphics Computer Systems, “*IRIS Performer™ Programmer’s Guide*”, document number: 007-1680-030
- [9] G.A. Geist, J.A. Kohl, P.M. Papadopoulos, “*CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications*”, INTL Journal of High Performance Computing Applications, Volume II, Number 3, August 1997, pp. 224-236
- [10] J.A. Kohl, P.M. Papadopoulos, “*CUMULVS user guide, computational steering and interactive visualization in distributed applications*”, Oak Ridge National Laboratory, USA, Computer Science and Mathematics Division, TM-13299, 02/1999
- [11] K.J. Weiler, “Topological Structures for Geometric Modeling”, Ph.D. thesis, Rensselaer Polytechnic Institute, Troy, NY, May 1986
- [12] Sense8, *World Tool Kit*, <http://www.sense8.com/>
- [13] TGS Inc., *Open Inventor*, <http://www.tgs.com/>
- [14] SUN Microsystems Inc., Java 3D™, <http://java.sun.com/products/java-media/3D/>