Chemical Flooding in a Virtual Environment - A Survivor's Guide to VR Development

Wes Bethel
ewbethel@lbl.gov
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
The University of California
Berkeley, California 94720

Abstract

Building something which could be called "virtual reality" (VR) is something of a challenge, particularly when nobody really seems to agree on a definition of VR. We wanted to combine scientific visualization with VR, resulting in an environment useful for assisting scientific research. We demonstrate the combination of VR and scientific visualization in a prototype application. The VR application we constructed consists of a dataflow based system for performing scientific visualization (AVS), extensions to the system to support VR input devices and a numerical simulation ported into the dataflow environment. Our VR system includes two inexpensive, off-the-shelf VR devices and some custom code. A working system was assembled with about two man-months of effort.

We allow the user to specify parameters for a chemical flooding simulation as well as some viewing parameters using VR input devices, as well as view the output using VR output devices. In chemical flooding, there is a subsurface region that contains chemicals which are to be removed. Secondary oil recovery and environmental remediation are typical applications of chemical flooding. The process assumes one or more injection wells, and one or more production wells. Chemicals or water are pumped into the ground, mobilizing and displacing hydrocarbons or contaminants. The placement of the production and injection wells, and other parameters of the wells, are the most important variables in the simulation.

Introduction

The term "Virtual Reality" (VR) has different meanings to different people. Most people associate VR with head-mounted displays, data gloves, and position trackers. The devices, which often receive the most attention in the media. [1] states rather emphatically that the "dress code" of VR is a head-mounted display and a dataglove. The devices, while the most visible component of VR, are really but a single component of VR. The devices are the medium of communication with the user, the hardware used to implement a man-machine interface.

VR can be thought of as a number of interrelated systems, of which the hardware (the man-machine interface) is but one component. The other components consists of a model, usually geometric, which is rendered into an image, presented to the user, and made available for interaction. Dynamic behavior is also an integral part of a VR system. That is, the user and model interact in a well-defined way. The user may move about within the model, viewing from different vantage points, or the user may be permitted, depending upon the system, to alter the model.

Ivan Sutherland generally gets cited as being the source of most of the original ideas in computer graphics. He is cited in [2] as providing a description of the "Ultimate Display" in which objects in the "virtual world" "look real, sound real and act real." There is a correlation between how well we can achieve these goals and the amount of hardware required to get the job done. In other words, constructing a very "convincing" VR system, which includes full "immersion", support for full 3D auditory cues and haptic feedback, even assuming you could buy the gear off-the-shelf, would cost a prohibitive amount of money.

We have decided that we are willing to sacrifice some of "convincing" for both "affordable" and "we can get it today." There is a wide spectrum of options for VR hardware, supporting a range of environments from "immersive" to "window-on-a-world" VR. For a fairly complete description of the various flavors of VR, and more references about VR devices, refer to [3].

Unlike VR hardware, which is maturing at a respectable rate, software tools for building VR systems are, from this developer's point of view, much less mature. Basically, there are two options. Either you can purchase a so-called "toolkit", or you can build everything yourself. Toolkits are subroutine libraries for use by skilled programmers. Their primary usefulness is for creating links between defined user actions and the resulting change in the underlying geometric model. We use the term *dynamic* to refer to a link between a user action and a change in the model (as opposed to a change in viewpoint, or some other rendering parameter). A benefit of toolkits is that they often provide support for a fairly wide variety of VR devices. VR authoring systems, in contrast to a toolkit, are fairly new (immature), and provide a means for interactively, rather than programmatically, creating the geometric model and establishing dynamics.

Our interest is in scientific visualization, and in using VR as an enhancement to the existing environment and tools employed for doing visualization. We have used the Application Visualization System [4] (AVS) as the environment for constructing a VR system, leveraging upon the strengths of this system as well as our user's knowledge of this system. In this paper, we will describe the customizations used in enabling VR in AVS, and describe a prototype VR/Visualization application. This paper is written assuming the reader has working knowledge of module development in AVS.

Architecture of a VR+Scientific Visualization System

As we have described, VR systems consist of a geometric model, some input and output devices, and an environment in which user actions are processed, user input gathered, models possibly changed and images rendered and displayed.

In all VR systems, there is a model of the world which is rendered and presented to the user. The geometric model in our VR system (a scientific visualization system extended to support VR) is created by an AVS dataflow network,which implements several visualization techniques, such as isosurface calculation and direct volume rendering.

Given that geometric model construction is the result of a scientific visualization process, we can conclude that all the knowledge and experience we have with these point-and-click-visual-programming visualization systems can be used for "doing" VR, at least insofar as creating the "model," and within the context of scientific visualization. This is important because one doesn't have to be an "expert" graphics programmer to use AVS (although it does help), and similarly, one shouldn't have to be an expert programmer to use VR. The user may construct models using visualization tools with which they are already familiar. No new learning is required, at least for this piece of the VR puzzle.

The remaining obstacles are managing the plethora of VR devices, and defining and implementing user-model dynamics. What are the options available to us if we wish to extend one of the dataflow visualization packages to support VR?

Previously, Sherman describes work [5] in which dataflow visualization packages are extended to do VR. Two modules, one for rendering and one for a VR input device are described. The renderer is a modified version of the package-supplied renderer, which required cooperative development between the package developer and the VR implementor. A tracker is used in association with an immersive display device. The tracker provides information about the viewer's location and view orientation. Sherman states that a renderer modification was required for gathering tracker information. Tracker input causes a change in rendering parameters (view parameters). The tracker was tighlty coupled with the renderer due to the delay induced by the dataflow model, where upstream modules pass data to downstream modules. In VR systems, particularly in immersive systems, a high frame rate is required to avoid side effects such as user disorientation (the user won't use the system if it makes her ill).

Sherman also describes a work-in-progress module for gathering information from VR input devices. The VR input device is used to locate a probe, allowing the user to query values in volume data.

Sherman's renderer is used to drive a Fakespace BOOM device (which provides view orientation information as well as acting as a display device). A VPL dataglove is used as a VR input device.

The similarity between our work and that of Sherman is in providing support for a VR input device. We use an unmodified version of the renderer supplied with AVS, and achieve VR output using a stereoscopic display device and the stereo capabilities of our graphics hardware (described later in this paper). Our VR input device is used in a different manner than that described by Sherman. Rather than using the input device to position a probe for data query, we use the device to set the position of objects (icons representing simulation parameters) in three space.

Other forays into combining VR and scientific visualization are described in [6] and [7]. Nearly all of the to-date published literature describing the combination of VR and scientific visualization are based upon systems which cost hundreds of thousands of dollars, and a significant commitment of human resources. What we describe here (and in [8]) is a VR system constructed from a couple of inexpensive, off-the-shelf VR devices, and some custom code, and assembled with about two man-months of effort. We explored the option of using toolkits, but concluded that, given our goals, custom code was the best choice.

The Input Device: The Spaceball

We decided to experiment with the Spaceball as our first VR input device. Among the deciding factors for choosing the Spaceball was that it was available, it is "cozy" (not psychologically intimidating), and that the user doesn't have to pick it up and carry it around. Our scientists sit literally for hours in front of a monitor doing visualization, so ergonomic issues are of great importance.

Our uses for VR input devices include specifying viewpoints, that is, allowing some way for the user to navigate through the virtual world created by their visualization program, as well as providing means for manipulating objects in three space. We found that some devices were better at supporting viewpoint manipulation, but were weaker in allowing a user to manipulate three dimensional objects. And other devices were better at manipulating objects, but weren't so straightforward to use in specifying viewpoints. We wanted to use a single device for both functions. The Spaceball seemed to provide the best fit for us in both of these categories. And it was relatively inexpensive (compared to the other VR devices on the market).

The Spaceball consists of a ball, about the size of a tennis ball (but without the green fur) mounted on a platform that sits on the desk. It doesn't have protruding wires, doesn't require soldering, etc. to make it work, and plugs into an RS-232 serial port. It is built from two annular spheres, the displacement between which causes data to be generated. Six dimensions of data are detected: three components of translational displacement, and three rotational components.

Spaceball Technologies distributes an X-client daemon with the Spaceball device (in binary form) along with some library archives and sample code. These items are handy for quickly bringing new applications online. More importantly, the use of an X-based protocol for obtaining information from a device means that VR devices can be used over a network. The significance of this fact becomes apparent when one considers the usefulness of a tool which is restricted to run on a particular piece of hardware (behind a locked door in the graphics lab) versus a tool which can be run over the network at the desktop (with the obvious performance degradation). In practice, we had some problems with implementing our application using the X-client daemon that came shipped with the Spaceball, which we are still investigating.

We obtained some publicly available code for the Spaceball which reads events and returns values indicating X, Y, Z displacement along with Yaw, Pitch and Roll (contact Spaceball Technologies for more information about this source code). The code is built around a UNIX select() system call (select() can't be used over the network in a general way, like the X-client daemon).

Using this code, we created a library of subroutines for initializing and polling the Spaceball device. With this library, we were able to quickly build a variety of AVS modules which use the Spaceball. One such module controls the position and orientation of the camera in the geometry viewer, another module allows the user to perform three dimensional object transformations.

Controlling the Viewpoint

A general purpose AVS module was written to control the position and orientation of the camera with the Spaceball. With this module, data is read from the Spaceball, and the camera position is updated using AVS Geometry library calls.

The view model which we have implemented allows the user to change the eye position. The look-at vector is relative to the eye position, so when the eye position changes, the look-at direction remains unchanged. Translation data from the Spaceball cause the eye point to move. The look-at vector changes when rotation data is detected on the Spaceball. Thus, the user may change the eyepoint without changing the gaze direction, or may change the gaze direction without changing the eye point. Changes in both occur whenever the Spaceball generates both translational and rotational data.

The mathematics required to support viewpoint specification are straightforward, and involve mainly bookkeeping of current eye position, look-at vector, and the up vector. In our module, we implement this using static variables, and recalculate the eye position, look-at vector and up vector each time we receive more spaceball data. This involves a few trigonometric function calls and a couple of matrix multiplies.

The goal of our camera model is to allow the user to manipulate the viewpoint in a predictable and easty-to-understand way. For example, given an arbitrary viewpoint in space, when the user generates +Z translational data on the Spaceball, the camera moves "into" the scene. When the user generates Roll data on the ball, the up vector is changed (the eye point and look-at values don't change) so that only the Roll of the current view changes.

Transformations occur relative to the current position of the viewer. Absolute transformations are difficult to intuitively grasp, from a user's point of view. A user wants to move a camera to the "left" or "right," relative to the current position, not along some absolute X axis, which generally has no relationship to the user's "left" or "right."

Controlling Three Dimensional Objects

The Spaceball device is also used to move objects around in three space. The same goals which apply to the camera model apply to object transformations as well. When a user generates +X translational data (for example) on the Spaceball, they want to see the object move "to the right" on the screen. The mathematics required to support this behavior include an inverse world-to-view-to-screen transformation matrix, along with a model-to-world transformation if one is required. In our prototype, we did not implement a general purpose object transformation module (as we did with the camera module, which is general purpose).

Our pick and transform module was implemented as a coroutine module rather than as a subroutine module because the geometry viewer does not provide support (in AVS 5.0.1) for the Spaceball as an input device in particular, or for arbitrary input devices, in general. When we have a valid pick event (generated from within the geometry viewer), we then enter into "edit mode" in which we want Spaceball events to drive the network. The only entry point for these events into the network, given current AVS capabilities, is via a coroutine. Implementing entry points for data from arbitrary input devices, in general, will be accomplished using coroutines.

In general, the process of adding arbitrary input devices to AVS consists of two broad tasks: data is read from the device; data is injected into the AVS network in the form of camera position changes or object geometry changes, or as raw data which is in turn transformed into camera positions, changes in geometry, or used elsewhere in the AVS network. As Sherman discussed, there is an inherent delay in this type of approach due to the nature of the dataflow systems. We feel that as these systems evolve and mature in implementation, that the delay induced by passing data between modules will diminish and no longer be an issue.

The Output Device: Stereoscopic Viewing

We have a Kubota Pacific Denali, attached to a DEC 3000/400. The software supplied by Kubota to drive the Denali includes a stereo "screen" available through the X server. The resolution of the stereo screen is roughly NTSC at 120hz. The Denali, when being placed into stereo mode, generates, on a separate line, a square wave at 120hz which is used to activate a stereo shutter. This line plugs directly into a Tektronix polarizing shutter which is mounted onto the front of a monitor. The user dons passive polarizing glasses and is presented with left-then-right eye images at 120hz.

Within AVS, the version shipped for use on the DEC/Denali combination contains a module for use with the geometry viewer for controlling the interocular distance and focal depth parameters. There is some confusion about exactly in what coordinate system these units are specified. The general idea, though, is that the "stereo effect" is best achieved when the focal depth is set to lie somewhat "in the middle" of the scene being viewed, and the interocular distance is set wide enough so that there is separation, but not so wide as to give the user a headache. This is a soft parameter. Discussions with other users who make use of stereoscopic displays indicate that it is useful to employ a "calibration" program. This program permits the user to set the stereo viewing parameters to values which they find acceptable. It turns out that there is no hard-and-fast rule which can be used to specify "good" values for these parameters. The reason is that no two users have the same interocular distance, and values that work well for one user may give another a headache.

Our system is built with two monitors. One is a stereo monitor and is connected to the Denali. The other monitor is connected to the 8bit graphics display on the DEC. When the VR application is run, the main AVS menus appear on the monocular screen, and the output from the geometry viewer is routed to the stereo screen. This arrangement has thus far worked well, capitalizing on the high-resolution of the monocular display for UI functions, and allowing an entire monitor to be dedicated to stereo viewing. With some Xserver configuration modification, the mouse cursor can be moved easily between the stereo and mono screens.

In extending AVS to create a VR system, we observe that a problem with many VR systems, particularly immersive systems, is that implementing a good, usable GUI is a difficult proposition. For example, in one system we looked at, that of a virtual operating room, the operating table was in the "center" of the room, but in order to change system parameters, the user was required to look away from the table to a menu that was mapped onto one of the walls in the virtual room. The act of looking back and forth was tedious and cumbersome and interfered with the use of the system. Given the amount of screen real estate consumed by the AVS GUI, two monitors, one for stereo viewing and one for the UI is a necessity. Compared to looking "over your shoulder" to access a menu (as in immersive systems), looking at the stereo then mono monitor (which are placed next to each other on a table) to access the UI proves to be simple and does not hinder the use of the system.

The Chemical Flooding Project

In our experience, many important lessons can be learned by first constructing and evaluating prototypes prior to general-purpose solutions. With this in mind, we undertook a project to evaluate the usefulness of VR as a technology, combining VR input and output devices with AVS in order to solve a particular problem. In this project, we ported a simulation for chemical flooding into AVS, and used a VR input device to manipulate its 3D input parameters, and a VR output device to enhance the display of the resulting visualization.

The context of chemical flooding is set in environmental remediation or secondary oil recovery from a reservoir. There is hydrocarbon or contaminant in the ground which we want to remove. Short of digging a huge hole, the way to remove chemicals from the subsurface is to pump other chemicals into the ground which mobilize and displace the existing substances. The chemical flooding simulation models this process, computing concentrations of various chemicals at specified spatial locations in time. It is beyond the scope of this paper to describe chemical flooding in technical depth. Refer to [9] for more information. Chemicals are pumped into the ground using injection wells, and removed from the ground using production wells. Determining an optimal placement of wells so as to maximize the amount of contaminant removed is the goal of this project. The intent is to allow for the best use of human intuition based upon the human's ability to process visual depiction of simulation results, and to allow the user to put a particular well "over there by that thing in the ground" using the VR input device.

The chemical flooding simulation has numerous parameters, some of which are scalar, some of which are three dimensional. The scalar parameters are easily mapped to standard dials and buttons in AVS. The three dimensional parameters are the locations of the wells. The Spaceball is used to specify the locations of the wells in three-space.

In implementing the VR interface and porting the simulation into AVS, we wrote three application-specific modules and one general purpose module. The general-purpose module provides the means to control the camera position and orientation using the spaceball. There is no relationship between the application-specific modules and the camera-Spaceball module.

One of the three application specific modules is the simulation itself. We added hooks to the Fortran code to change how the simulation disposes of its data at each time step (the interface to the AVS field structure), as well as wrote a C-language wrapper which implements the AVS description function, manages coroutine events, and so forth. The output from the simulation is an AVS field. The field contains information about chemical concentrations at each node in the finite difference grid at each time step. At present, this field is three dimensional (a three dimensional array at each time step). A four dimensional field would permit the user to use an appropriate slicing tool ("new ortho slicer," for example, contributed by Lawrence Berkeley Laboratory to the International AVS Center) to look at three dimensional hyperslices from four dimensional data. The output may be visualized using a variety of tools, such as the isosurface generator.

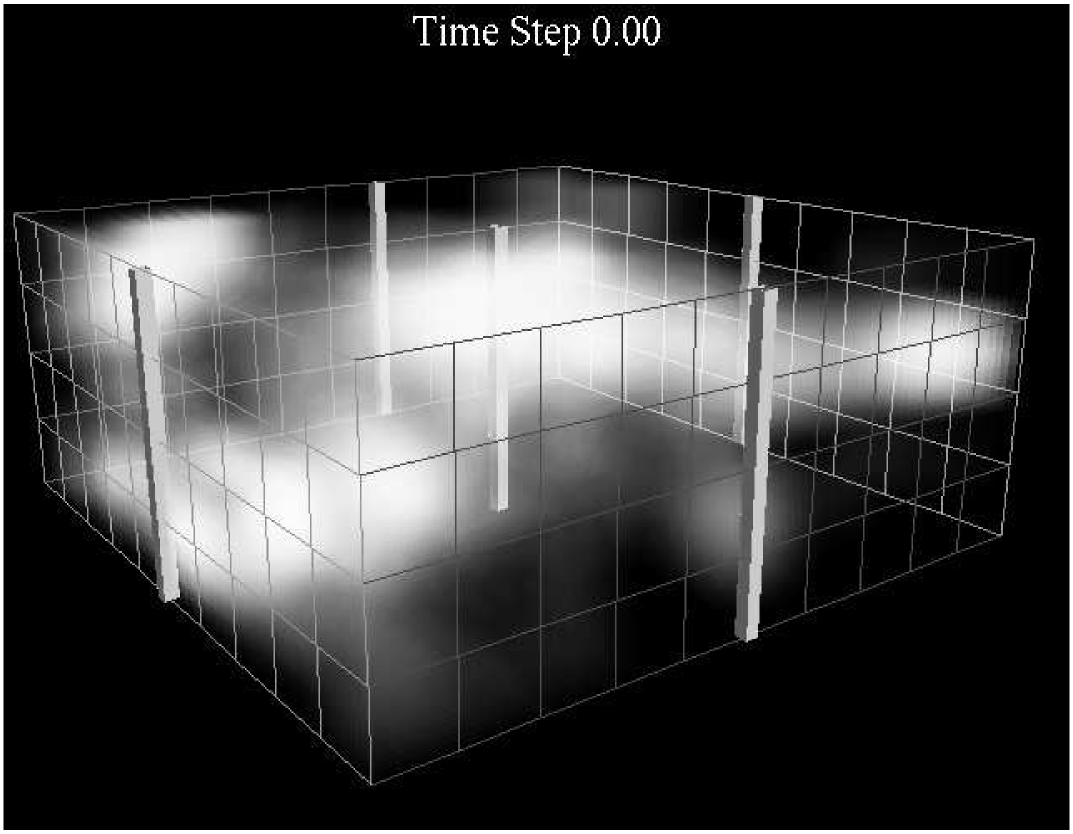

Figure 1

Finite Difference Grid, Well Icons and Volume Rendered
Subsurface Permeability

The other two modules read (and possibly change) simulation input data files. One of the modules reads the file containing information about the finite difference grid used in the simulation, and generates a field with information about subsurface physical parameters, such as rock permeability, at each grid point. The grid may be visualized using a number of parameters. Figure 1 shows the finite difference grid, along with volume-rendered subsurface permeability. Areas of high opacity represent regions of low permeability (high resistance to flow).

The other application-specific module reads the simulation file which contains information about the location and type of wells used in the simulation. The module outputs AVS geometry which represents the wells. In this module we implement the wells editing interface. Thus, this module interfaces to the Spaceball library, and uses the upstream transform structure. When a pick event is generated by the geometry viewer, the name of the picked object (contained in the upstream transform structure) is checked against the list of object names which are assigned to the well icons. If a match is detected, the module enters into "well editing mode." Upon entry, the selected well is highlighted. A highlighted well is represented using a visually loud color. In addition, a text string appears on the screen indicating if the well is a production or injection well, and the well type (pressure or rate constrained), as well as the coordinates of the well location within the finite-difference grid. An inverse view-to-world transformation is computed. Spaceball translational events are processed to compute the new location of a well in finite-difference grid coordinates. As the events are processed, the well-icon geometry is changed to reflect the new location within the grid (the display changes to reflect the new well position). The grid coordinates in the informative text string associated with the highlighted well are updated to provide additional feedback to the user as the well is located with the VR input device. To exit well-edit mode, the user again "picks" the highlighted well with the mouse.

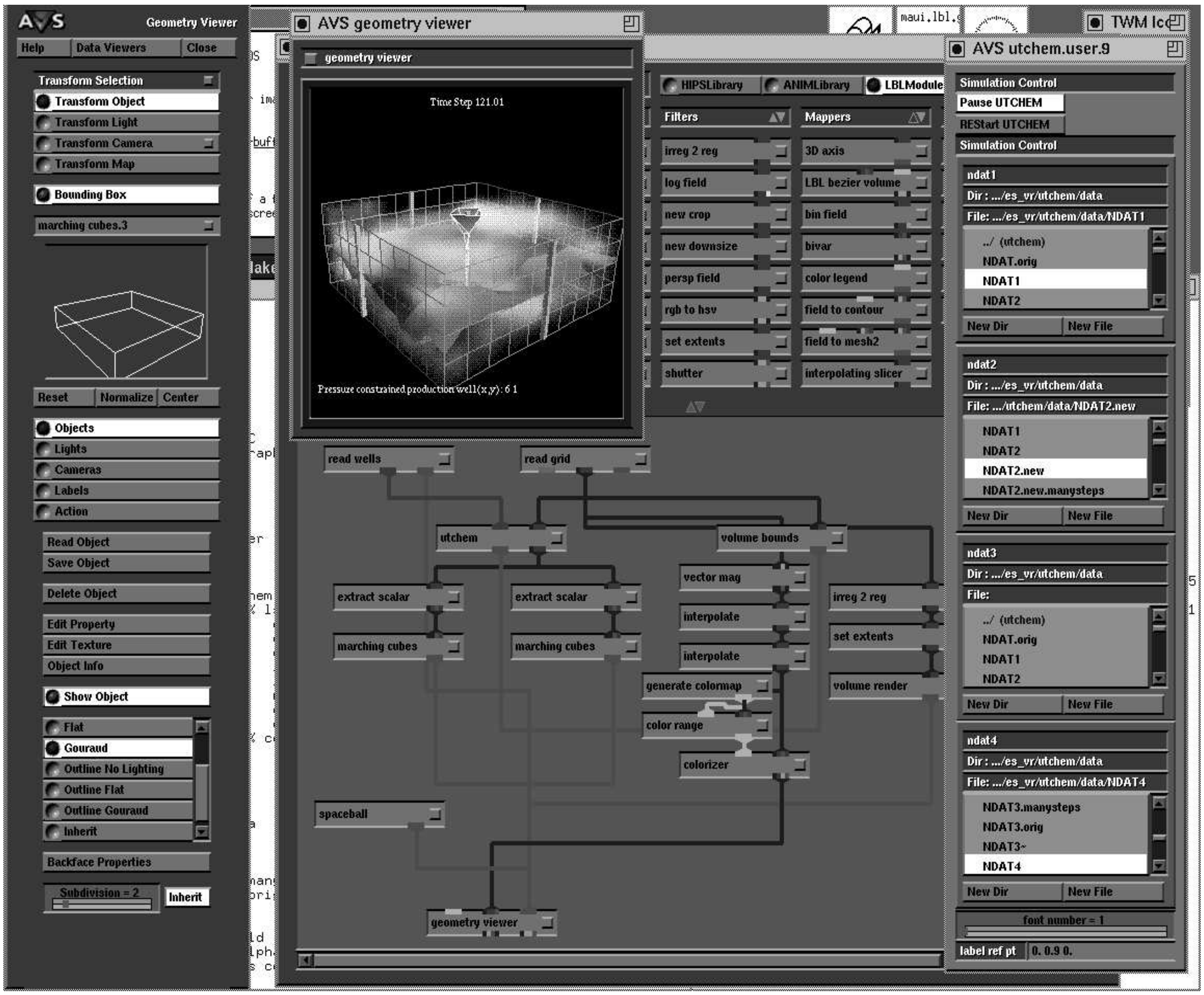

Figure 2.

Screendump of AVS Network showing the visual program implementing the Chemical Flooding VR Prototype.

Figure 2 is a screendump of an AVS network for the complete network used to implement the VR/ Chemical Flooding project. There is a hidden connection between the Geometry Viewer and the "wells" module which is the transport for the upstream transform structure.

Figure 3 shows two isosurfaces of chemical concentration from the simulation during the middle of a run, rendered along with the finite difference grid and volume-rendered subsurface permeability. Figures 4 through 6 show several time steps of the simulation, and a video accompanies this paper showing complete simulation runs for several different placement configurations of the wells.

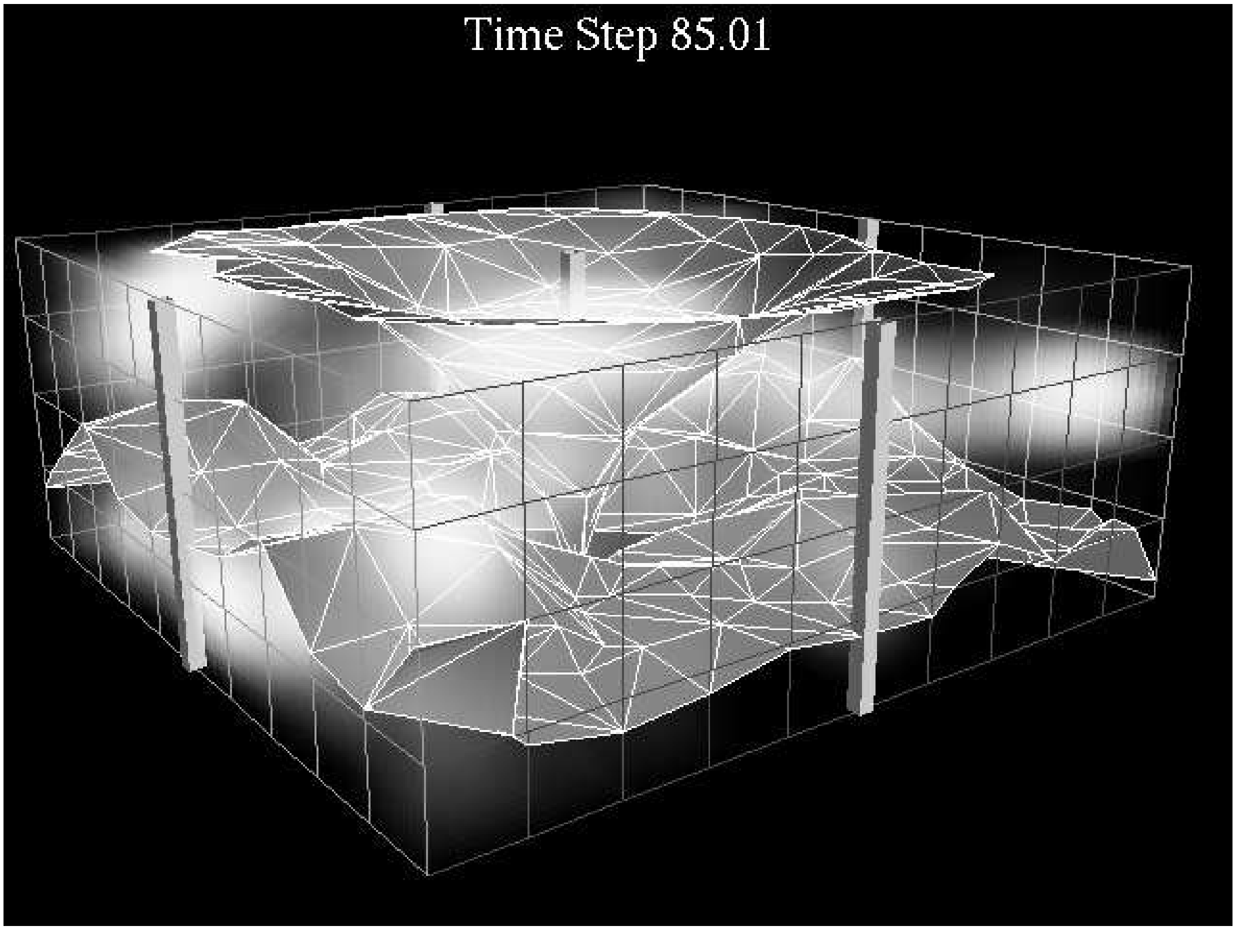

Figure 3.

Isoconcentration surfaces of water (top) and mobilized oil (bottom)
combined with volume rendered subsurface permeability.

Conclusion

We have identified several components of a VR system: a geometric model; a rendering and display component.; and a mechanism for allowing the user to interact with the model as well as the rendering and display process. Using this model of VR architecture, we have described our prototype system., which demonstrates a low-cost but highly effective approach to combining scientific visualization with VR.

With our prototype application, chemical flooding in a virtual environemnt, the user can experiment with the placement of wells within the finite difference grid, converging on an optimal solution of well placement by taking advantage of the human visual processing system.

The user community has been very excited about using VR interfaces to this and other simulations. We are presently applying this technology to other visualization problems.

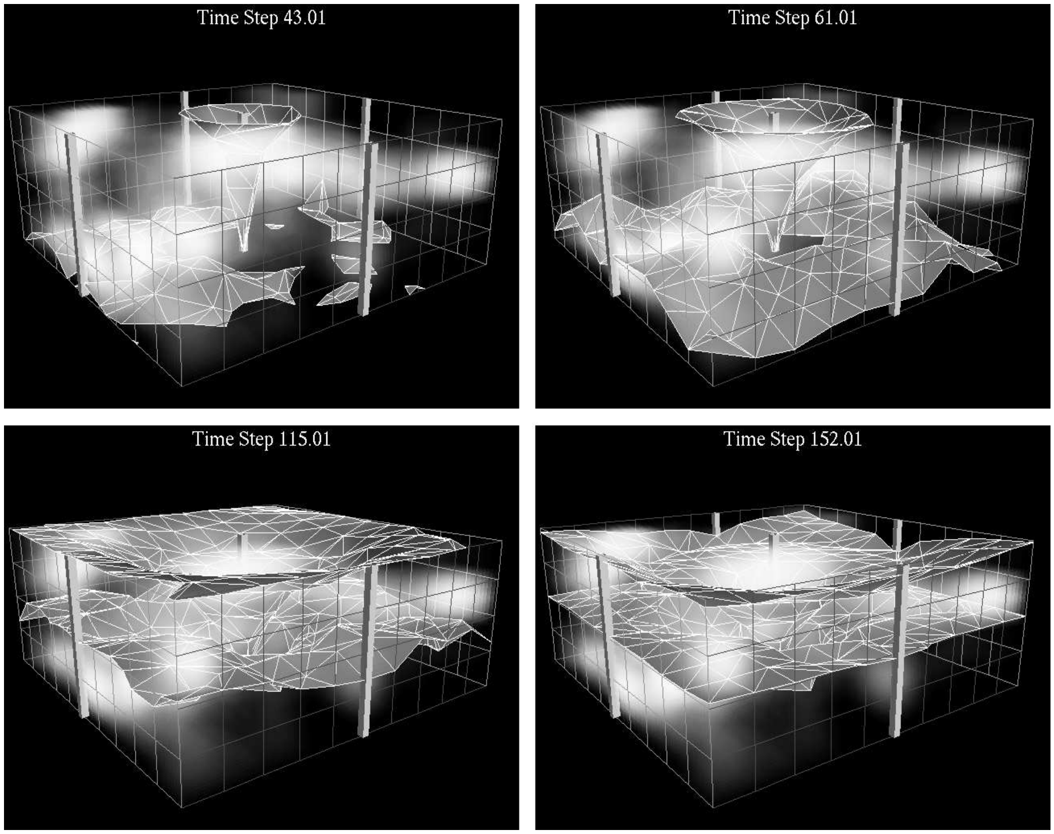

Figure 4.

Several time steps from the Chemical Flooding simulation.

Acknowledgement

This work was supported by the Director of the Scientific Computing Staff, Office of Energy Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. The author wishes to thank Harvard Holmes of Lawrence Berkeley Laboratory for suggestions and comments which have helped to clarify many of the concepts presented in this paper.

References

[1] Negromonte, N., Virtual Reality: Oxymoron or Pleonasm?, *Wired,* Volume 1, Number 6, December 1993.

[2] Bishop, G., and Fuchs, H. (co-chairs), Research Directions in Virtual Environments, Report of an NSF Invitational Workshop, University of North Carolina - Chapel Hill, March 1992, *Computer Graphics*, Volume 26, Number 3, August 1992.

[3] Isdale, J., "What is Virtual Reality? A Homebrew Introduction and Information Resource List," Version 2.1, October 1993. Available via ftp from ftp.u.washington.edu.

[4] Upson, C., Faulhaber, T., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., Van Dam, A., The Application Visualization System: A Computational Environment for Scientific Visualization, *IEEE Computer Graphics and Applications,* Volume 9, Number 4, July 1989.

[5] Sherman, W., Integrating Virtual Environments into the Dataflow Paradigm, Fourth Eurographics Workshop in ViSC, Workshop Papers, April 1993.

[6] Bryson, S., and Levit, C., The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows, *Proceedings of IEEE Visualization 91*, pp17-24, 1991.

[7] Cruz-Neira, C., Sandin, J., DeFanti, T., "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE," *Computer Graphics,* Proceedings of Siggraph 1993, pp 135-142.

[8] Bethel, W., Jacobsen, J., Holland P., "Site Remediation in a Virtual Environment," *Proceedings of IS&T/SPIE Symposium on Electonic Imaging Science and Technology*, January 1994.

[9] Datta-Gupta, A., Pope, G., Sephernoori, K., Thrasher, R., A Symmetric Positive Definite Formulation of a Three-Dimensional Micellar/Polymer Simulator, *SPE Reservoir Engineering,* pp. 622-632.