# PVR
# An Architecture for Portable VR Applications

Robert van Liere and Jurriaan D. Mulder

Center for Mathematics and Computer Science CWI,
P.O. Box 94097, 1090 GB Amsterdam, Netherlands
{robertl,mullie}@cwi.nl

**Abstract.** Virtual reality shows great promise as a research tool in computational science and engineering. However, since VR involves new interface styles, a great deal of implementation effort is required to develop VR applications.
In this paper we present PVR; an event-based architecture for portable VR applications. The goal of PVR is to provide a programming environment which facilitates the development of VR applications. PVR differentiates itself from other VR toolkits in two ways: First, it decouples the coordination and management of multiple data streams from actual data processing. This simplifies the programmer's task of managing and synchronizing the data streams. Second, PVR strives for portability by shielding low-level device specific details. Application programmers can take full advantage of the underlying hardware while maintaining a single code base spanning a variety of input and output device configurations.

## 1 Introduction.

Virtual reality shows great promise as a research tool in computational science and engineering. The analysis of 3D data sets (read from disk or computed on the fly) may benefit from the interface styles provided by virtual reality. By providing additional depth and viewing cues, the virtual reality interface can aid the unambiguous display of 3D structures. In addition, virtual reality input interfaces allow the direct and intuitive exploration of the data, as well as providing control over the application through widgets integrated into the interface.

However, the development of VR applications is not easy. In order to fulfill the real-time requirements posed on VR applications, programmers must learn many new and non-standard methods. Technical issues that make these methods more difficult than traditional 3D graphics methods are mostly related to: (i) the management of multiple I/O data streams at predictable and time critical intervals, (ii) 3D interaction with the virtual world, and (iii) the large variety of display and input devices with each device having individual characteristics and constraints. Virtual reality development environments will need to provide high-level support for each of these issues, as well as support found in traditional 3D graphics toolkits, such as for modeling and rendering.

Event-driven programming models are well known to graphical user interface programmers. Instead of explicitly polling and responding to user actions, applications register event handlers and the underlying user interface toolkit will call the handler when a particular event occurs. Event-driven models can ease the programmer's task since the control and coordination of the user interface objects is left to the underlying toolkit.

In this paper we present the design and implementation of a portable virtual reality architecture, PVR. The architecture was inspired by our work on distributed computational steering [1]. The motivation for designing and implementing PVR was to provide an application programmer an easy to use and portable development environment. Ease of usage is accomplished by decoupling the coordination of multiple data streams from actual data processing. This simplifies the programmer's task of managing, synchronizing and processing multiple I/O streams. Portability is realized by allowing the programmer to abstract from specific input and output devices.

The format of the paper is as follows: First, in section 2, we briefly survey related work. In section 3 we present the underlying concepts of the PVR architecture and discuss design considerations. In section 4 we discuss implementation aspects of the PVR architecture. Finally, we provide two examples on how the environment is used: a molecular dynamics visualization and the analysis of nuclear division.

## 2 Related work.

Bryson has extensively studied the application of virtual reality interfaces in scientific visualization. Although the work has mostly been related to studies in the virtual windtunnel, the lessons have lead to generalized requirements with regard to implementation issues concerning computation, graphics and data management (eg. [2, 3]). Requirements related to real-time performance and natural "anthropomorphic" VR interfaces are discussed in some detail.

Numerous academic and commercial efforts have focussed on the development of distributed virtual reality applications. Often these efforts have resulted in generic toolkits; many of these have similar objectives and motivations as PVR. For example, a non-exhaustive list of widely used VR toolkits is: Alice [4], AVOCADO [5], Bamboo [6], COVISE [7], DIVE [8], Lightning [9], and World Toolkit [10]. It is beyond the scope of this paper to give an exhaustive overview of each of these toolkits.

The MR toolkit was one of the first toolkits to decouple the simulation from the rendering in order to achieve the real-time requirements of VR user interfaces [11]. MR provides many high-level features that are needed in the developing VR applications: support for distributed computing, workspace management, performance monitoring, input devices abstractions, data sharing, etc.

Researchers at IBM have used multiple workstations to support the real-time requirements of VR user interfaces [12]. Their virtual world architecture addresses the requirements of virtual reality for high performance computing,

concurrency and synchronization of multiple events, and flexibility. Processes communicate through a central event-driven user interface management system (UIMS). The UIMS uses a collection of rules to synchronize concurrent input/output events. It also provides flexibility by clearly separating the interaction techniques from the virtual world application.

The CAVE library is an API of C functions and macros to control the operation of the CAVE [13]. The CAVE library takes care of all the tasks that have to be performed to correctly operate the CAVE. CAVE functions keep all the devices synchronized, produce the correct perspective for each wall and provide applications with the current state of all CAVE elements.

Despite similar objectives and functionality, PVR differs from these toolkits in many ways. First and foremost is the emphasis of PVR on the ease of usage for the VR application programmer. PVR provides a modular architecture that shields the management and synchronization of multiple data streams. This is achieved by decoupling the coordination and management of multiple data streams from actual data processing. Programmers need only to supply the processing modules, while the coordination of the data streams is maintained by PVR. Second, PVR strives for portable applications. Portability goes beyond portability of low-level input devices. The PVR architecture allows flexible mapping from high-level interaction abstractions onto lower-level device abstractions. Also, although not unique, PVR allows for the flexible mapping of logical to physical output devices.

## 3 PVR Architecture.

### 3.1 Internal Structure of PVR

An overview of the PVR architecture is given in figure 1. The architecture consists of one or more processes attached to a bus. Processes communicate using an event-based mechanism. This mechanism is conceptually easy to understand: after a process attaches to the bus, it registers patterns with the bus that describe the events it is interested in. This allows each process to set an *event filter*. When an event is posted on the bus, it is redistributed to the interested processes. Each process is parameterized with a user supplied callback. The bus will invoke the callback whenever an event occurs that the process is subscribed to.

PVR supports a small number of predefined events, such as `PVR_REDRAW`, `PVR_MOTION`, `PVR_PRESS`, etc. In addition, applications may define events to denote any application specific action.

Processes are categorized in one of four types:

1. Render processes are responsible for rendering. The application defined callback will be executed when a redraw is necessary. Often, particular output device configurations will require more than one render process; i.e. the CAVE will use one process per wall whereas a HMD might have two.

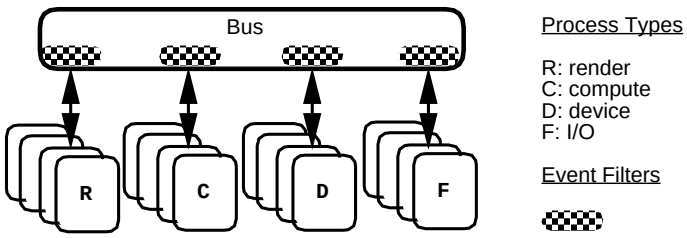


**Fig. 1.** The PVR architecture consists of one or more processes attached to a bus. Each process registers event filters that describe events that the process is interested in. A process posts events to the bus to signal the completion of an action. The bus redistributes events to processes according to their event filters.

2. Device processes are responsible for device handling. An event will be posted each time the state of a device changes (i.e. button press, sensor motion, etc). The event report will contain the information of the device.
3. File processes are responsible for all file I/O. Having designated file processes is useful, since I/O tends to be very slow. File processes are also responsible for I/O transactions with external services. For example, performing transactions with data base management systems, CORBA servers, or external simulations.
4. Compute processes are responsible for all other required computation. This may be the complete simulation itself, or the computation of visualization techniques (streamlines, iso-surfaces, etc). It is the programmer's choice to determine the granularity of the compute processes.

A shared data facility allows multiple processes to share data structures. This facility allows processes to allocate and lock shared data structures in a simple and flexible way. Pointers to shared data structures can be stored in events.

### 3.2 PVR's Mechanisms

Rather than discussing the architecture in detail, we highlight three mechanisms unique to PVR.

1. Device independence.
   Virtual reality environments may have a plethora of display, tracking, and input devices, each operating in their own physical coordinate space. PVR is designed to be portable, allowing applications to take full advantage of the underlying hardware while maintaining a single code base spanning multiple device configurations. For example, applications are developed independently of the tracker geometry, room geometry and device configurations.
   PVR uses the notion of a workspace that removes the application's dependency on the physical location of the devices that it uses. Workspace management functionality provides two sets of transformation mappings; a mapping between application workspace and standard workspace and a mapping

between standard workspace and device workspace. These two mappings allow applications to use device independent input and output spaces. The mapping between standard workspace and device workspace is defined in a configuration file. The application can define the mapping between application workspace and standard workspace. PVR events encapsulate device information, so processes that register device events are device independent.

2. Synchronization and Data Ordering.
   Events are used by a process to signal the end of an action. In this way a sequence of processes can be set up; each process sends an event to the next process in the sequence. However, when processes operate on one data set in parallel, additional mechanisms must be provided. For example, consider the case that one process computes iso-surfaces and a second process computes streamlines in a time dependent data set. Both geometries must be rendered in the same frame.
   PVR provides a data framing mechanism to order data from parallel processes. Applications may define a frame structure which consists of an number of data slots and an event. Processes may put data in a slot. As soon as all slots are filled the bus will distribute the event. This signals that the frame is complete. The familiar notion of a barrier can be implemented with the framing mechanism.
   A time line for a frame with four slots is diagrammed in figure 2. Processes may fill data into a slot at any time and in any order. Only when the last slot is filled will the bus invoke the callback of those processes that have registered `EVENT`.

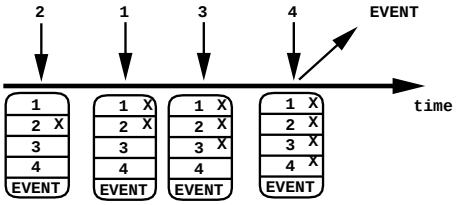


**Fig. 2.** A frame with four slots and event `EVENT`. Processes add data to a slot. The bus will distribute `EVENT` as soon as all slots have new data, signaling a new frame.

The bus can also manage a stack of frames. If a process refills a slot before the frame is filled, the bus will place the data in the next frame in the stack. In this way processes do not need to wait until a frame is complete, but can continue to process new data.

3. Distributed Virtual Reality.
   In addition to the single user virtual reality, multi-user distributed virtual reality can provide added value in those cases where collaboration is needed. Two cases are distinguished: First, multiple remote environments are linked

together to form one single distributed environment in which users have the ability to manipulate a common data set. Second, multiple users can participate in a single environment.
PVR includes a mechanism for distributed applications, whereby several users on different machines can share a virtual environment. This is achieved by connecting remote busses together. A posted event will be redistributed to a remote bus, according to event registration patterns on the remote bus (see figure 3).

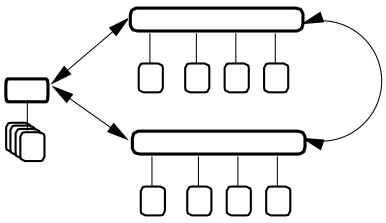

**Fig. 3.** Multiple busses are connected to form a distributed architecture. Events are redistributed according to event filters on the remote bus.

From the perspective of a process, it makes no difference whether the application is distributed. The bus will invoke the process's callback whenever an event occurs that is designated for the process, irrespective of whether the event was posted from a process on the local or remote bus.

## 4 Implementation.

The PVR architecture sketched in the previous section can be implemented in a number of ways. The current implementation uses a number of portable supporting software packages (see figure 4). The implementation relies on POSIX threads for process support, GLUT and OpenGL for windowing and rendering support, and the University of North Carolina's tracker library for a portable tracking interface. Higher level rendering packages can be used on top of OpenGL; for example: Inventor, OpenGL Optimizer or OpenGL Volumizer can be used within the rendering thread.

The current implementation supports an Electrohome Retrographics Display, a Fastrak tracking system for head tracking, a flying joystick for spatial input, spaceball, and other input devices such as keyboard and 2D mouse. Although the current implementation is targeted towards high-end SGI hardware and IRIX 6.5, there is no reason that it can not be ported to other hardware/software environments[1].

[1] The implementation for the Pyramid CAVE and a HMD underway. The CAVE implementation will be completed before the April workshop.

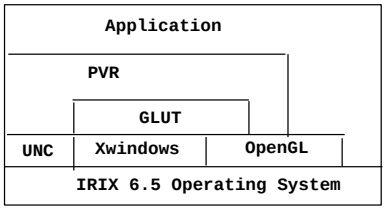


**Fig. 4.** The layered implementation of the PVR architecture.

## 5 Applications.

### 5.1 Molecular Dynamics Visualization.

The application is a molecular dynamics simulation of an eutectic mixture of molten lithium and potassium carbonates at 1200 Kelvin[2]. The movements of 158 $Li^+$ ions, 98 $K^+$ ions and 128 $CO_3^{2-}$ ions in a cubic box are calculated during 2500 time steps. Of particular interest is the study of the coordination number; this is the number of $CO_3^{2-}$ ions in a neighborhood of a $Li^+$ or $K^+$ ion. Typically, $Li^+$ ions are enclosed by 4 carbonates, and the $K^+$ ions by 6 carbonates.

Figure 5 shows two snapshots of the interface. Yellow spheres are used to display $Li^+$ions, green triangles with 3 red vertices and a grey center are used to display the carbonate ions. A convex hull is used to visualize the coordination between a $Li^+$ ion and related $CO_3^{2-}$ ions. The dashed red-white lint shows a measuring tape (each line segment represents 0.25 nm.), which is used to monitor distances between ions.

Virtual reality techniques are used to study the structure and the dynamics of the simulated molten carbonates. Immersive environments are very suited to perceive valuable spatial information from the displayed data. For example, in what pattern the ions are diffusing, the dynamic nature of a coordination, in what way are carbonate ions oriented towards the metal ions, etc. In addition, immersive environments can provide added functionality to directly manipulate the data, such as direct manipulation of ions for selection and direct manipulation for measuring distances and angles between ions.

The bus configuration for the molecular dynamics visualization is shown in figure 6. It consists of 6 processes: one file, one render, two compute and two device processes. The file process reads in a sequence of data sets from disk. Every time step a data set is read, the file process will send an application specific event (`TIME_STEP`) to the bus. One device process manages head movements and posts a `REDRAW` event when applicable. The second device process manages the spatial joystick and posts a `JOYSTICK` event when the position, orientation or button

[2] Thanks to Jos Tissen (Unilever) and Jack van Wijk (TU Eindhoven) for providing the data. The 3D visualization techniques and their implementation are discussed in [14].

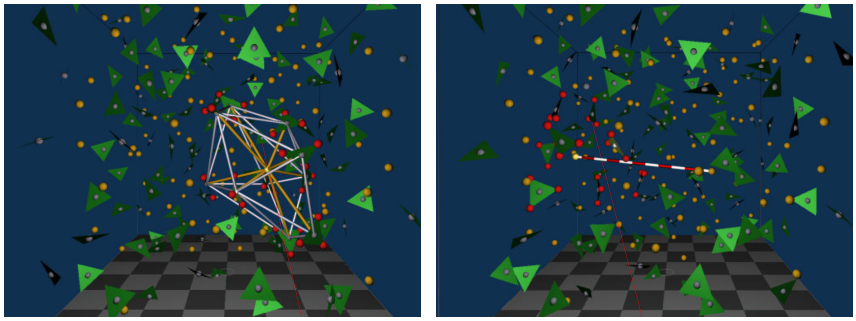

**Fig. 5.** Molecular dynamics of an eutectic mixture of molten lithium and potassium carbonates. Left uses a convex hull to visualize a coordination of a selected $Li^+$ ion. Right uses a measuring tape to monitor distances between two selected ions.

state of the joystick has changed. One compute process computes the convex hull (triggered by the `TIME_STEP` and `JOYSTICK` events). The second compute process computes distances between atoms (triggered by the `TIME_STEP` and `JOYSTICK` events). Both compute processes post a `REDRAW` event. The render process redraws the scene (triggered by the `REDRAW` event). If appropriate, multiple redraw requests will automatically be collapsed into a single redraw.

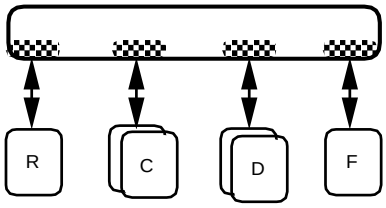


**Fig. 6.** PVR Molecular Dynamics Visualization configuration.

The performance of the PVR implementation is better than the original sequential program. The original program sequentially polls input devices, reads a data set from disk, computes the convex hull, and renders the graphics. The PVR implementation exploits parallelism by interleaving computation, device handling, rendering and file I/O. In general, these performance observations are valid for other PVR applications as well. PVR programmers may improve performance by interleaving (eg reading the next data set while rendering the previous one) and parallelism (executing multiple techniques simultaneously). The tradeoff to be made is the granularity of processes that do computing versus the small overhead introduced by sending events over the bus. When necessary,

the programmer can change the process granularity by joining/splitting callback procedures.

### 5.2 Nuclear Division.

One of the problems studied by researchers at the E.C. Slater Institute of the University of Amsterdam is that of nuclear division, i.e. *mitosis*. In the process of mitosis, two sets of microtubles interact to form the mitotic spindle. Of particular interest is the monitoring of the microtubles in the spindle over time.

Data sets have been acquired from a confocal light microscope and displayed using PVR[3]. Figure 7 shows two snapshots of the interface. The two spindle poles are clearly shown at 5 and 11 o'clock. The render process uses OpenGL Volumizer for the actual volume rendering. Simple colormap editing operations are available to allow interactive manipulation of the transparency values. Currently there is no interaction other than colormap manipulation and simple viewing transformations. The combination of stereoscopic graphics and head tracking allow the researcher to walk around the spindle, thus effectively perceiving the complex spatial relationships of the microtubules in the spindle.

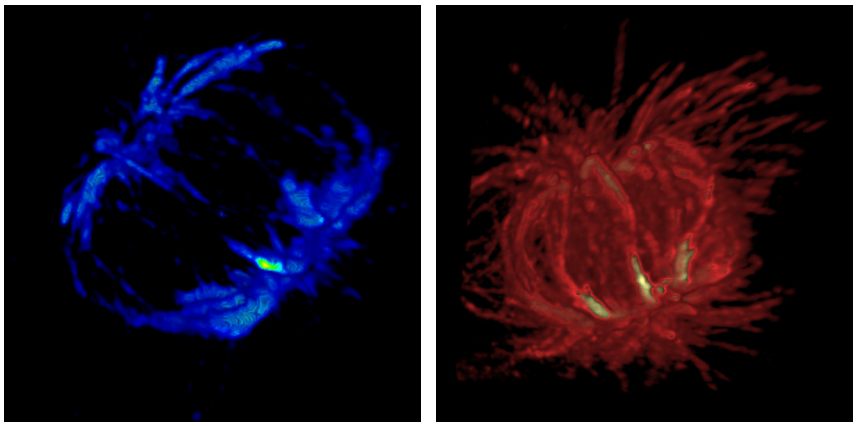

**Fig. 7.** Microtubules in a spindle during nuclear division. Left uses a blue-yellow color map with high transparancy values, right uses a red-yellow color map with lower transparancy values. Head tracked stereo display allow reseachers to effectively perceive the complex spatial relationships of the microtubules.

The bus configuration for the spindle is shown in figure 8. It consists of 4 processes: one file, one render and two device processes. The file process reads in the data sets from disk (currently, only one data set is read). After a frame

[3] Thanks to Roel van Driel (ECSI, UvA) and Hans van der Voort (Scientific Volume Imaging B.V.) for providing the data.

is read, the file process will send an application specific `TIME_STEP` event to the bus. One device process manages the head positions. This is exactly the same callback procedure as in the molecular dynamics application. The second device process edits a new colormap, and sends the application specific `NEW_CMAP` event to the bus. The render process redraws the scene (triggered by the `REDRAW`, or `NEW_CMAP` events). The `NEW_CMAP` event indicates that a new colormap must be used by the rendering process.

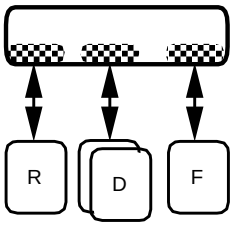


**Fig. 8.** PVR Nuclear Division configuration.

## 6 Conclusion.

In this paper we introduced PVR, an architecture for portable VR applications. PVR facilitates the development of highly concurrent yet synchronized virtual reality interfaces by decoupling the coordination of the data streams from actual data processing. Two VR applications were used to illustrate PVR's governing ideas.

PVR differentiates itself from other VR toolkits in two ways. First, PVR is completely event driven. Programmers only supply callback routines which are executed whenever necessary. Application specific events can be posted to signal the completion of an application action. Second, PVR strives for portability by shielding low-level device specific details from the programmer. A single code base can be used to span a variety of workspace and input device configurations.

In the future we will extend PVR with a number of new concepts. These will include the usage of the Fahrenheit scene graph for distributed VR applications, composition of higher level virtual input devices and other forms of high-level interaction [15], incorporation of multiple 2D and 3D workspaces in a single application, and event tracing/monitoring tools.

## References

1. R. van Liere, J.A. Harkes, and W.C. de Leeuw. A distributed blackboard architecture for interactive data visualization. In R. Yagel and H. Hagen, editors, *Proceedings Visualization '98*, pages 235–244. IEEE Computer Society Press, 1998.

2. S. Bryson. Virtual reality in scientific visualization. *Computers & Graphics*, 17(6):679–685, 1993.
3. S. Bryson. Real-time exploratory scientific visualization and virtual reality. In L.J. Rosenblum et al, editor, *Scientific Visualization: Advances and Challenges*, pages 65–86. Academic press, 1994.
4. R. Deline. Alice: A rapid prototyping system for three-dimensional interactive graphical environments. Technical report, Computer Science Department, University of Virginia, 1993.
5. F. Hasenbink. Avocado system. Technical report, GMD Department of Visualization and Media Systems Design, Bonn St. Augustin, 1997.
6. K. Watson and M. Zyda. Bamboo - a portable system for dynamically extensible, real time, networked, virtual environments. In *1998 IEEE Virtual Reality Annual International Symposium*, pages 252–260. IEEE Computer Society Press, March 1998.
7. D. Rantzau and U. Lang. A scalable virtual environment for large scale scientific data analysis. *Future Generation Computer Systems*, 14(4):215–222, 1999.
8. C. Carlsson and O. Hagsand. Dive - a platform for multi-user virtual environments. *Computers & Graphics*, 17(6):663–669, 1993.
9. J. Landauer, R. Blach, M. Bues, A. Rosch, and A. Simon. Towards next generation virtual reality systems. In *Proceedings of the IEEE Conference on Multimedia Computing and System*, Ottawa, 1997. IEEE Computer Society Press.
10. World toolkit. Technical report, Sense8, 1997. http: // www.sense8.com/ products/ worldtoolkit.html.
11. C. Shaw, J. Liang, M. Green, and Y. Sun. The decoupled simulation model for virtual reality systems. In *Proceedings of the CHI 92 Conference on Human Factors and Computing Systems*, pages 321–328, 1992.
12. P. Appino, J. Lewis, L. Koved, D. Ling, D. Rabenhorst, and C. Codella. An architecture for virtual worlds. Technical Report RC 16446, IBM Research Division, T.J. Watson Research Center, 1991.
13. D. Pape, C. Cruz-Neira, and M. Czernuszenko. CAVE users guide. Technical report, Electronic Visualization Laboratory, University of Illinois at Chicago, 1996. http: // www.evl.uic.edu /pape / CAVE / prog / CAVEGuide.html.
14. J.J. van Wijk and J.T.W.M. Tissen. Visualization of molecular dynamics. In *Proceedings of the Fourth Eurographics Workshop on Visualization in Scientific Computing*, Abingdon, UK, 1993.
15. J.D. Mulder. Remote object translation methods for immersive virtual environments. Presented at the 1998 Virtual Environments Conference & 4th Eurographics Workshop, June 1998.