Enhancements to VTK enabling Scientific Visualization in Immersive Environments

Patrick O'Leary* Kitware, Inc., USA
Sankhesh Jhaveri† Kitware, Inc., USA
Aashish Chaudhary‡ Kitware, Inc., USA
William Sherman§ Indiana University, USA
Ken Martin¶ Kitware, Inc., USA
David Lonie‖ Kitware, Inc., USA
Eric Whiting** Idaho National Laboratory, USA
James Money†† Idaho National Laboratory, USA
Sandy McKenzie‡‡ Kitware, Inc., USA

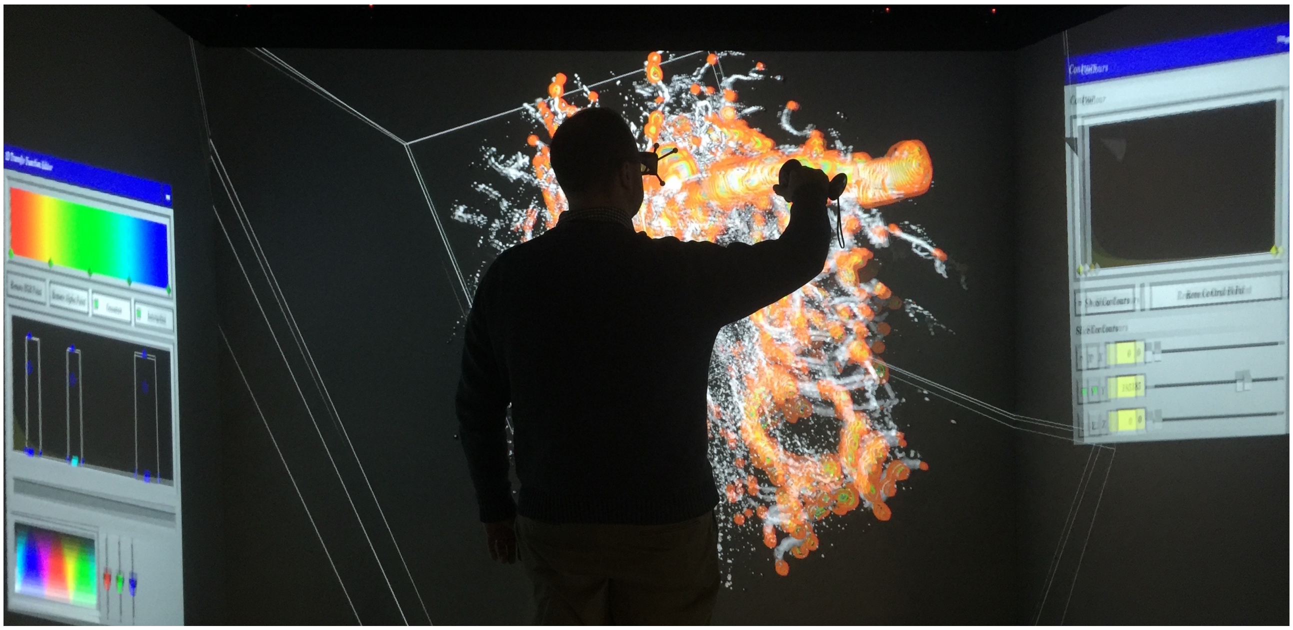

Figure 1: A Vrui based mixed contour and volume rendering of 256x256x256 aneurism data enabled by the Visualization Toolkit (VTK) in Idaho National Laboratory's Center for Advanced Energy Studies four-sided CAVE™.

Abstract

Modern scientific, engineering and medical computational simulations, as well as experimental and observational data sensing/measuring devices, produce enormous amounts of data. While statistical analysis provides insight into this data, scientific visualization is tactically important for scientific discovery, product design and data analysis. These benefits are impeded, however, when scientific visualization algorithms are implemented from scratch—a time-consuming and redundant process in immersive application development. This process can greatly benefit from leveraging the state-of-the-art open-source Visualization Toolkit (VTK) and its community. Over the past two (almost three) decades, integrating VTK with a virtual reality (VR) environment has only been attempted to varying degrees of success. In this paper, we demonstrate two new approaches to simplify this amalgamation of an immersive interface with visualization rendering from VTK. In addition, we cover several enhancements to VTK that provide near real-time updates and efficient interaction. Finally, we demonstrate the combination of VTK with both Vrui and OpenVR immersive environments in example applications.

Keywords: Scientific visualization, immersive environments, virtual reality

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality; H.5.2 [Information Interfaces and Representation]: User Interfaces—Interaction Styles

*e-mail: patrick.oleary@kitware.com
†e-mail: sankhesh.jhaveri@kitware.com
‡e-mail: aashish.chaudhary@kitware.com
§e-mail: shermanw@indiana.edu
¶e-mail: ken.martin@kitware.com
‖e-mail: david.lonie@kitware.com
**e-mail: eric.whiting@inl.gov
††e-mail: james.money@inl.gov
‡‡e-mail: sandy.mckenzie@kitware.com

1 Introduction

There is a growing body of evidence that demonstrates the measurable benefits that can be attained when exploring scientific data using immersive interfaces such as molecular research at the University of North Carolina at Chapel Hill [9], genetics at the National Center for Supercomputing Applications [8], oil well placement at the University of Colorado [16], confocal microscopy data at Brown University [38], and interdisciplinary immersive visual analytics at the Electronic Visualization Laboratory [29] to name but a few. While it is not scientifically verifiable, any time scientists express situations where they "discovered" some relationship in their data while immersed in a virtual reality (VR) system, we can make the case that the interface provided a utility that helped them advance their work.

Yet, knowing that there are benefits is only half the equation. The other half is cost. A considerable contribution to the cost—one that is often not formulated—is personnel time to get data into the VR system. That time expense is frequently exacerbated due to a lack of tools that allow data to be directly imbibed into a virtual environment.

A path that many research teams have taken is to use the established and feature-rich Visualization Toolkit (VTK). VTK is an application programming interface (API) that provides quick access to an expanse of scientific visualization rendering algorithms as well as to components for displaying and interacting with results on a desktop. While the concept of combining VTK with VR is sound, the compatibility of VTK with other rendering software presented a difficult challenge. There were several reasonably successful attempts at this amalgamation, but in the end, there were either too many inefficiencies to allow the software to be adequately interactive, or the melding was too fragile to maintain, as VTK and the VR libraries each evolved.

Consequently, the better solution was to adapt VTK to enable it to be more easily integrated into other rendering systems. Thus, we adapted VTK by adding new options for rendering. Rather than only being able to render into windows with graphics contexts created by VTK itself, it is now the possible to "externally" render into contexts provided by a collaborating system or even to integrate a VR system directly into VTK.

Immersive visualization efforts are often associated with research facilities that provide large-scale VR systems such as a CAVE™or another large-screen walk-in display. There is also a growing audience of potential VR users who can now gain access to immersive interfaces through the new abundance and low cost of head-mounted displays (HMDs). Ideally, there would be one solution to reach both audiences. While this is technically possible, the consumer systems offer a simpler approach that will entice many developers to follow that path. Thus, we offer two approaches, one that addresses the simpler solution of integrating VTK directly with Oculus or OpenVR and another that allows the integration of VTK with any full-fledged VR integration library that is capable of interfacing with CAVE™style and HMD displays.

OpenGL context sharing. Our `vtkRenderingExternal` VTK module provides a complete integration API that includes proper lighting, interaction, picking and access to the entire VTK pipeline. The module enables simple utilization for application developers with any OpenGL based VR Toolkit.

VR Toolkit embedding. The Oculus and OpenVR VTK modules directly support several immersive environments without the issues faced by previous work, and they provide a complete template for embedding other VR toolkits within VTK in future work.

Enhanced performance. As the nature of immersive interfaces, especially HMDs, requires high-performance rendering, our effort also includes VTK rendering enhancements such as the following:

- a new default OpenGL 3.2+ pipeline;
- dual depth peeling for transparency; and
- symmetric multiprocessing (SMP) tools and algorithms.

In the sections that follow, we illustrate how our amalgamation of VTK and VR toolkits supports our goals for enhancing scientific visualization through immersive environments.

2 Related Work

The use of scientific visualization in immersive environments is simply natural while tactically important for scientific discovery, product design and data analysis. There are several high-quality scientific visualization VR applications created from scratch using OpenGL directly [5, 27, 44, 39] such as the stunning graphics processing unit (GPU) accelerated hybrid volume and glyph approach for molecular dynamics and other visualizations in the CAVE2™ [41, 40]. These are certainly exemplary and valuable applications. When building immersive applications with scientific visualization requirements, much like when building desktop applications, it is often more efficient to leverage the open-source visualization toolkit (VTK) [43].

Throughout the past two decades, several research teams and developers have integrated VTK with immersive environments to varying degrees of success. Four fundamental approaches are available to enable VTK in an immersive system:

- geometry transport;
- OpenGL context sharing;
- VR toolkit embedding; and
- OpenGL intercept.

Our recent enhancements to the VTK platform contain solutions for the desired integration that present a number of contributions. Therefore, we review related work for these areas.

Geometry transport An early approach to VTK-VR integration was to use the `vtkActorToPF` library [28]. In this approach, the generation of visualization geometry is decoupled from the rendering of the geometry. (See Figure 2.) The geometry is generated by VTK in the form of actors that consist of polygons and properties. `vtkActorToPF` transforms these actors into pf-Geodes (nodes) that are included in a Performer (or OpenSceneGraph) scene graph. The geometry is created by VTK, and the scene graph is rendered without VTK. Only geometry is transformed. Cameras, lights, rendering and interaction are not incorporated. Several applications utilized the equivalent `vtkActorToOSG` for an OpenSceneGraph-based scene graph [10] or directly into OpenGL [34]. Others have used VTK in a preprocessing step to produce geometries or textures eliminating the need for a direct connection to VTK [6].

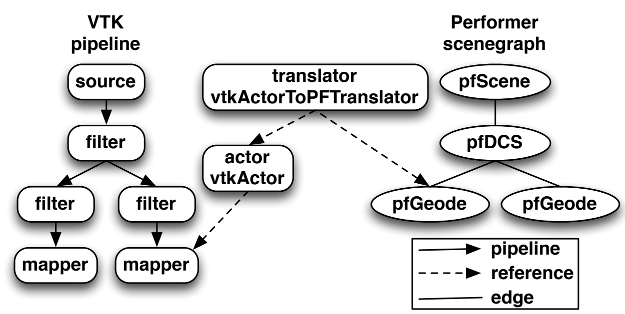

Figure 2: VTK, vtkActorToPF and Performer interaction diagram. (Recreated from Paul Rajlich [28].)

VTK can be used to create, transport and save geometry without rendering. As effective as this approach can be, the loose coupling of VTK and the VR toolkit creates more obstacles than benefits from an application developer's perspective. The approach is, therefore, not built upon by this work.

OpenGL context sharing Rather than share just the VTK geometry, the application developer would like to use all of the VTK API from within his or her immersive application. In VTK, the renderer and render window classes are responsible for rendering scenes. VTK creates its own window and associates an OpenGL context with that window for the renderer to use. The following lists what an OpenGL context represents: all of the state; the default framebuffer; and everything affiliated with OpenGL with respect to the renderer, the window and the application. The application developer seeks to simply share the OpenGL context from the VR toolkit with a third-party rendering software (e.g. Delta3D [30], OpenSceneGraph [51] and VTK).

In previous work by Sherman et al. [46] and others, Delta3D and OpenSceneGraph were quickly modified to instead use the windows and the associated OpenGL contexts of a VR integration library such as Vrui [25]. OmegaLib [13] software for hybrid reality display environments also implements its own OpenGL context-sharing module for VTK, `omegaVtk`. The AlloSystem [1] for the AlloSphere Research Facility originally had an OpenGL context-sharing VTK plugin, but it has recently (October 2016) changed to our `vtkRenderingExternal` VTK module, which is presented in this paper. These solutions are generally limited in their integration. Specifically, a rendering library that is unaware of the actual viewing matrix does not calculate lighting correctly, and picking input operations do not conform to the shifted rendering. As the VTK API adapts and adds enhancements in time, these implementations are generally locked in to older versions of VTK.

Our `vtkRenderingExternal` VTK module formalizes this integration, providing lights, interaction and picking connectivity that are lacking in other implementations. The module also offers the application developer complete access to the VTK pipeline.

VR toolkit embedding A similarly time-proven approach is based on the modification of the VTK renderer and render window [50, 17, 45, 3]. To render in an immersive environment, derived classes of `vtkRenderer` and `vtkRenderWindow` are created, which depend on fundamental calls to the VR toolkit. Thus, VTK based applications can simply exchange these two items to run on the desktop or the immersive environment. vtkCave [48] for the CAVELib [31], followed by vjVTK [7] and VR JuggLua [37] for VRJuggler [4], created third-party software, essentially deriving the `vtkRenderWindow` and `vtkRenderer` classes. Outside of VTK, lighting and interaction were not shared, which resulted in troublesome behavior.

In this work, we have created two new VTK modules based on either the Oculus [12] or the OpenVR [49] software development kits (SDKs). OpenVR is an API developed by Valve to support its SteamVR ecosystem that is compatible with the HTC Vive and other VR hardware [42]. Oculus provides software development kits more closely tied to its equipment for mobile, desktop and web VR. The Oculus and OpenVR VTK modules directly support several immersive environments without the issues faced by previous work. These modules provide a complete template for embedding other VR toolkits within VTK in future work.

OpenGL intercept A fourth possible means for melding VTK into a virtual environment system is the *OpenGL intercept* method [19, 20, 52, 47, 32], in which middleware is inserted between the application and the graphics card at runtime. With this technique, closed-source applications can be rendered so that the head-tracked perspective rendering overrides the internal view matrix to provide the VR experience. Thus, this technique enables basic desktop tools to be used with an immersive interface—albeit a limited interface, given the open-loop nature of grabbing the rendering but not connecting back to the parameter interface. Yet, the perspective rendering alone can be extremely valuable and can allow scientists, engineers or medical researchers to interact with their desktop tools in a whole new way. Many of these methods, however lack full functionality in immersive environments, which limits their usefulness to end users. Pure OpenGL, without any modifications or additions, is sure to work with interception. The difficulties of using intercept methods are that they require more coding and tagging, and they are not guaranteed to work at all. These difficulties are becoming more apparent with OpenGL 3.0+, especially when using a core OpenGL profile.

In contrast to the OpenGL intercept method, the work in this paper is for the application developer, and there is no intention to eliminate the need to write code. This work aims to make it easier to develop scientific visualization immersive applications by leveraging VTK.

Enhanced performance Near real-time update of scientific visualization metaphors is crucial in immersive environments. The field has seen several proposed solutions from decoupling rendering and processing to parallel visualization [11, 50]. This effort stands apart from all these previous efforts. Valiant as they were, the efforts were ultimately lost to time, as VTK has continued to evolve, making it difficult for tacked-together components to remain in sync with the API. Rather, by providing rendering access from within the VTK API itself, new tools can rely on a stability that has not been available for techniques that perform functions outside the bounds of the API design. Often, such techniques access internal features that do not have the assurance of stability. As a commercially supported open-source tool, the rendering performance of VTK is continually being advanced. VTK has been around since 1993. The toolkit has over 100,000 repository commits from more than 250 contributors. Having the latest algorithm implementation requires using the existing implementation in VTK or contributing the algorithm to VTK.

We present recent enhancements to VTK that significantly impact immersive environment application development. The OpenGL 3.2+ pipeline, described in Hanwell et al [18], provides the most dramatic improvement in performance. We have supplemented this work with Bavoil and Myers dual depth peeling [2], as well as with symmetric multiprocessing (SMP) tools and algorithms, to address the performance issues for transparent geometry and computationally intense algorithms (e.g., isosurfaces).

Finally, our work will eventually allow application developers to leverage portable, threaded data parallel algorithms that are capable of running on next-generation hardware from VTK-m [33]. VTK-m has shown impressive results in its first two years of development. We plan to make the filters in the VTK-m repository available in the next major release of VTK. Thus, if VTK-m is contained in VTK, then applications developed with the integrations described in this paper can use VTK-m.

3 Approaches

To achieve broader usage and to work as closely as possible to the standard application development workflow, it is important to require few, if any, changes to either the VR toolkit or the third-party scientific visualization software. For VTK, this was accomplished by adding new features that fit within the existing architecture. VTK provides a well-defined rendering pipeline through the `RenderWindow`, `Renderer`, `Camera`, `Actor`, and `Mapper` classes. This precise pipeline definition and the clear-cut API of VTK enabled us to primarily build upon existing code. In the next section, we cover details on these components from the architecture point of view. In the implementation sub-section, we provide in-depth details of features we implemented to support configurable immersive scientific visualization applications.

3.1 OpenGL context sharing

Traditionally, VTK creates and manages its own OpenGL context as well as the data objects within the scene. The objective of this work was to bring the high-quality scientific visualization, computing and rendering capabilities of VTK to VR environments in a way that is easier to develop and maintain. By bringing VTK into virtual environments created by interface-specific tools such as VRUI and FreeVR, we provide the means necessary to build interactive, three-dimensional (3D) scientific visualizations to the developers of the VR community.

3.1.1 Architecture

Integrating VTK into external rendering systems required overriding some of the behavior of the `vtkRenderWindow`, `vtkRenderer`, and `vtkCamera` classes. `vtkRenderer` is attached to `vtkRenderWindow`, `vtkMapper` is attached to `vtkActor`, and `vtkCamera` is attached to `vtkRenderer`. In a typical VTK application, the `RenderWindow` class is responsible for creating a rendering context as well as for defining the width and the height of the visualization viewport. The `Renderer` class is responsible for rendering one or more `Actor` objects and managing the viewport within the `RenderWindow` class. The `Actor` class is a drawable entity, which uses a `Mapper` object to render specific data within a `Renderer` object. Figure 3 shows the classes and their interactions.

Each of these components has its corresponding derived classes that implement the API with OpenGL, the underlying graphics API for VTK. OpenGL provides VTK with the ability to use hardware acceleration that ultimately leads to better visualizations and to near real-time performance, which many interactive applications require, especially ones that are designed for immersive environments. Each component of VTK participates in a specific way and communicates with other components via the public API. For instance, `RenderWindow` typically creates the context in which `Renderer` draws entities (e.g., `Actor` objects).

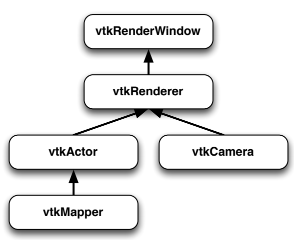

Figure 3: vtkRenderWindow, vtkRenderer, vtkCamera, vtkActor and vtkMapper interaction diagram.

Since `vtkRenderWindow` typically creates the context, and `vtkRenderer` controls the rendering of individual objects in a scene of a given viewport, the rendering pipeline is constructed with properties and other attributes set specifically to support this most general use case. In the case of external environments, however, the context is created outside of VTK. Non VTK graphical elements (such as the graphical user interface (GUI)) may be rendered before or after the VTK rendering. In addition, the environment may render its own visualization objects in the same context. To handle this situation, we have introduced a new module in VTK called `vtkRenderingExternal` that comprises four new classes: `vtkExternalOpenGLRenderWindow`, `vtkExternalOpenGLRenderer`, `vtkExternalOpenGLCamera` and `ExternalVTKWidget`.

`vtkExternalOpenGLRenderWindow` - This class is an extension to the `vtkGenericOpenGLRenderWindow` class, which provides a platform-agnostic VTK OpenGL window. The external render window class prevents a new VTK render window from being created and instead uses an existing OpenGL context. It is also responsible for fetching stereo parameters from the parent OpenGL application and setting them on the VTK pipeline.

`vtkExternalOpenGLRenderer` - This class derives from `vtkRenderer` and provides all of its features and functionalities. The external renderer offers an API that prevents it from clearing the OpenGL color and depth buffers at each frame. This ensures that the main application holds control over the OpenGL context and preserves rendered elements, of which VTK is unaware, in the scene.

`vtkExternalOpenGLCamera` - This class inherits `vtkCamera` and enables the projection and modelview matrices to be set on the camera. As a result, the external rendering framework can easily set the view and orientation on the VTK camera. The external camera also uses this scene information to compute accurate lighting matrices.

`ExternalVTKWidget` - This is a collective implementation that provides a plug-and-play approach to the `vtkRenderingExternal` module. It allows the consumer application to use all the new classes as described above in just one step. The overarching application needs only to instantiate this class to use the external rendering capabilities of VTK. The `ExternalVTKWidget` creates a new external render window or uses one provided to it from the external library/application.

3.1.2 Implementation

Some of the most important prerequisites of this work were seamless stereo rendering and user interaction between the rendering native to the base VR toolkit and the rendering embedded in VTK.

Stereo Rendering The OpenGL context maintains the state machine, in which OpenGL commands change the state of the system or query a particular state as needed. To support stereo rendering, we utilized the OpenGL context, set by the VR toolkit, to determine the type of stereo (Quad Buffer, Side-by-Side, or Top-Bottom stereo) and simply render using the OpenGL context. This context sets the active buffer, stenciling, etc. Such settings are made only once, immediately after the context has been created, and they are maintained by the VR toolkit over time.

2D and 3D Interface Widgets In most cases, VTK elements will not be the only objects in a scene. There will probably be some GUI elements that will also be rendered. Thus, the VTK rendering will be mixed with other OpenGL elements. The new `ExternalRenderer` class does not clear the depth or color buffers. It leaves this task to the display integration library or application. The depth buffer can then act to allow OpenGL elements to be mixed (composited) in 3D space, with closer elements occluding farther ones.

User Interactions Generally, in the case of a VR toolkit, interactions such as navigation in the scene space, grab, and rotation of various scene objects are handled by the VR integration library (e.g., Vrui). VTK has its own classes and methods for interaction and scene-object manipulation. To synchronize the navigation in these two systems, the `vtkExternalOpenGLCamera` class has been added. This class empowers the external application to manage camera interaction for VTK objects. We added an OpenGL Graphics Library (OpenGL) query in the external renderer, which uses the GL state system to get the projection and the modelview transformation matrices. These two matrices determine the location and the orientation of the end user's eye (camera) in the scene. `vtkExternalOpenGLRenderer` sets these matrices on `vtkExternalOpenGLCamera`. Setting these matrices directly on the camera leaves the camera parameters such as position, focal point, and view-up direction as incorrect values. Therefore, we compute appropriate viewing coordinates for the camera by multiplying the modelview matrix with the initial default position of the camera in the OpenGL coordinate system. Once, everything is set on the camera, the navigation and lighting works as expected by the end user.

The application itself handles the secondary kind of interactions such as slicing and clipping of the scientific datasets. VTK provides classes (filters) to perform thresholding, clipping, slicing, etc. These filters take parameters such as thresholding value, slicing position and clip position. In our implementation, the application receives the 6-DOF tracker position data, and based on the mode of the application, it uses this information to set appropriate values on a specific filter. For example, the left-hand controller might be used to position the clipping plane. This integration is straightforward because our module makes the coordinate system consistent between the two rendering systems.

3.1.3 Enabling `vtkRenderingExternal`

This work has been merged into VTK as of the release of version 7.0, which is available at `http://www.vtk.org`. To enable this module when compiling VTK with `CMake`, set `Module_vtkRenderingExternal` to `ON`. (The default is `OFF`.)

3.2 VR Toolkit Embedding (Oculus and OpenVR)

The potential VR user base has grown profusely with the emerging proliferation of consumer-level HMD VR displays along with their associated software ecosystems such as SteamVR from Valve. For developers who are willing to specifically target this audience, perhaps excluding users of CAVE™style VR displays, a simpler VTK-VR alternative is also available. The trade-offs—for developers who do not already have expertise in a full-fledged VR integration library—are avoiding the programming of the alternative VR integration library and immediately gaining access to HMDs compatible with Oculus or OpenVR but not to other VR display systems.

To enable using consumer VR devices with VTK, we created Oculus and OpenVR modules in VTK . Our goals are to allow VTK programs to use consumer VR devices with few changes, if any, and to support natural interaction when using them. If the executable is linked to the `vtkRenderingOpenVR` or `vtkRenderingOculus` modules, the object factory mechanism will replace the core rendering classes (e.g., `vtkRenderWindow` and `vtkRenderer`) with the VR specialized versions in VTK. This also entails changes to the interaction model, as VR and VR devices are 3D in nature, since single and multi-touch events occur in 3D space as opposed to the more traditional 2D screen space. To this end, we created new classes that support natural picking and interaction given 3D input devices.

One example of this low barrier of entry is incorporating consumer VR support into the latest release of ParaView. With minimal changes to ParaView, we are enabling researchers to load up existing visualizations and send them to VR to explore them with HTC Vive or Oculus Rift. Researchers can bounce back and forth between the traditional desktop and consumer VR experiences with the press of a button. This type of integration targets our goal of providing a solution with a low barrier of entry.

3.2.1 Implementation

The integrated VR support contains the following classes as drop-in replacements in VTK.

`vtk(OpenVR/Oculus)RenderWindow` - This is a derived class of the `RenderWindow` class. This class holds the initialization and main interfacing to the consumer VR toolkit.

`vtk(OpenVR/Oculus)Renderer` - This is a derived class of the `Render` class. Consumer VR devices exist in real-world coordinates such as meters, while the world coordinate system of a visualization can be anything from microns to astronomical units. As such, movements in the real world need to be scaled and translated into reasonable movements in the world coordinates of the visualization. The `vtk(OpenVR/Oculus)Renderer` class computes a reasonable scale and a translation, which are then used to compute the view and input device transforms. The class also sets an appropriate default clipping range expansion.

`vtk(OpenVR/Oculus)Camera` - This is a derived class of the `Camera` class. `vtk(OpenVR/Oculus)Camera` gets the matrices from the VR library to use for rendering and integrates them with the model, view and world matrices from the visualization. It contains a scale and a translation that are designed to map world coordinates into the HMD space. Accordingly, the application developers can keep world coordinates in the units best suited to their problem domains, and the camera will shift and scale into units that make sense for the HMD.

`vtk(OpenVR/Oculus)RenderWindowInteractor` - VTK is designed to pick and interact based on two degrees of freedom, desktop X and Y mouse/window coordinates. In contrast, VR provides X, Y and Z 3D world coordinates as well as 3D orientations. The `vtk(OpenVR/Oculus)RenderWindowInteractor` class catches controller events and converts them to mouse/window events. In addition, this class also stores the world coordinate positions and orientations for the styles or pickers that need them. `vtk(OpenVR/Oculus)RenderWindowInteractor` supports multiple controllers through the standard `pointerIndex` approach that VTK uses for multi-touch.

`vtkInteractorStyle3D` - In concert with the VR specialized `vtkRenderWindowInteractor` classes, we derived the `vtkInteractorStyle3D` class to use 3D world coordinate events to manipulate `Actor` objects and handle multi-touch 3D events such as scaling or translating the world to real transforms. This class provides a grab-and-move style of interaction that is common to OpenVR and other VR toolkits.

`vtkPropPicker3D` - Finally, the derived `vtkPropPicker3D` class determines what `Actor` objects or `Prop` objects VTK picks. Note that `Prop` is an abstract superclass for any object that can exist in a rendered scene (either 2D or 3D). It defines the API for picking; the LOD manipulation; and the common instance variables that control visibility, picking, and dragging. The `vtkPropPicker3D` class uses the 3D world coordinate from a VR device as the picking value, as opposed to using a 2D event and intersecting a ray, which is slower.

These derived classes work from within VTK to provide seamless access to cameras, lighting, interaction and the complete VTK pipeline.

3.2.2 Enabling the OpenVR and Oculus Modules

To use VTK with Oculus or OpenVR support, first download VTK 7.1 or later from the VTK repository on GitHub. (See `http://www.vtk.org`.) To enable these modules, use `CMake` to set `Module_vtkRenderingOpenVR` or `Module_vtkRenderingOculus` to `ON`. (The default is `OFF`.) The `CMake` build process will prompt for some external libraries such as Simple DirectMedia Layer 2 (SDL2) and the OpenVR or Oculus SDK, as appropriate. Ensure to build an optimized version of VTK to maximize performance while using these new capabilities.

3.2.3 Future Developments

The consumer VR support is currently in the beta phase and has been tested on HTC Vive and Oculus Rift. Moving forward, we look to add support for the features such as overlays, which provide support for user interface components. We also expect to include more event interactions, Oculus touch controller support, and measurement widgets.

3.3 Performance enhancements

VTK is one of the most commonly used libraries for visualization and computation in the scientific community. Primarily written in C++, VTK provides classical and model visualization algorithms to visualize structured, unstructured and point datasets on desktop, mobile, and web environments. VTK provides state-of-the-art implementations that are accessible via an API call. The benefit of using VTK comes from the fact that having the latest algorithm implementation simply requires using the existing implementation from the open-source, community-driven VTK repository or contributing one.

Allowing VTK to function at the levels needed for head-tracked rendering entailed making many other enhancements to the overall VTK system: using modern OpenGL, rendering transparencies with dual depth peeling and expanding the use of multi-threading.

3.3.1 OpenGL 3.2+

The legacy rendering code in VTK is a group of implementation modules collectively called "OpenGL." Through a grant from the National Institutes of Health, the OpenGL group has been rewritten as a drop-in replacement set of implementation modules collectively called "OpenGL2." This work aims to support rendering on modern graphics cards [18].

The results have been nothing short of spectacular. Polygon rendering demonstrates a tenfold speed-up for the first frame rendering, followed by a two-hundredfold speed-up for subsequent frames for up to thirty million triangles. The previous volume rendering was also GPU aware, and thus, the improvement is a modest but substantial twofold speed-up.

To realize these performance enhancements, VTK now uses an OpenGL 3.2+ context, which is available on fairly low-end modern GPUs. For those application developers using the X11 window system on a Mac OSX system, xQuartz does not currently provide a suitable OpenGL context. As xQuatrz utilizes newer versions of Mesa going forward, we expect future versions will eventually meet the OpenGL2 requirements.

3.3.2 Dual Depth Peeling

As we developed several example programs that leverage the `vtkRenderingExternal` module, we found that the rendering performance slowed, as transparency was introduced into the scene. We have developed a dual depth peeling algorithm to overcome this issue.

In OpenGL, polygons are broken up into fragments through the rasterization process. Each fragment corresponds to a pixel. An OpenGL fragment shader is a customizable program that determines the color of a fragment, where all fragments for a single pixel are blended to determine the final color of the pixel. Composing multiple translucent fragments into a single pixel must be done carefully. There are three common strategies to this composition:

- **Simple Alpha Blending** - Fragments are processed (blended using just alpha) in random order. It is very fast, but it provides unpredictable and generally incorrect results.
- **Sorted Geometry** - Geometry must be resorted each time the camera moves using `vtkDepthSortPolyData`. Sorting is an expensive (slow) operation, but it provides generally consistent results with some artifacts, where primitives overlap.
- **Depth Peeling** - Fragments are extracted and blended in a multipass render. Therefore, the process requires multiple geometry render passes.

VTK by default uses depth peeling. To enhance rendering performance with transparency we implemented `vtkDualDepthPeelingPass`, which was originally proposed by nVidia in 2008 [2]. Dual depth peeling extends traditional depth peeling by extracting two layers of fragments per pass: from the front and back simultaneously. It uses a two-component depth buffer to track peel information and three types of geometry passes:

- **InitializeDepth** - This is a pass that initializes buffers using opaque geometry information.
- **Peeling** - This is a repeated pass that extracts and blends translucent geometry peels. It extracts both near and far peels while blending far peels into the accumulation buffer.
- **AlphaBlending** - This is an optional pass to clean up unpeeled fragments that are used with occlusion thresholds.

This algorithm provides a twofold speed-up for compositing in the appearance of transparent geometry.

3.3.3 vtkSMPTools

The field of parallel computing is advancing rapidly due to innovations in GPU and multicore technologies. The VTK community is working to make parallel computing for scientific visualization easier by introducing `vtkSMPTools`, an abstraction for threaded processing that uses different libraries such as Threaded Building Blocks (TBB) [21], OpenMP [36] and XKaapi [15]. The typical target application is coarse-grained shared-memory computing as provided by mainstream multicore, threaded central processing units (CPUs) such as i5 and i7 architectures from Intel.

For several of the example programs that utilize the `vtkRenderingExternal` module, we leveraged new contouring algorithms in VTK that are readily parallelizable using `vtkSMPTools` and are still incredibly efficient in serial mode. These algorithms are called `vtkFlyingEdges2D` and `vtkFlyingEdges3D`. While the OpenGL2 group improves rendering performance, `vtkSMPTools` can be used to enhance the geometry generation performance for scientific visualization.

4 RESULTS

To demonstrate the utility our VTK enhancements provide scientific visualization efforts and to test various use cases, we have implemented three kinds of applications for the `vtkRenderingExternal` approach as well as a simple example using `vtkOpenVR`. Using the Vrui VR integration library, we created the following applications: GeometryViewer, VolumeViewer and MooseViewer. As the names suggest, GeometryViewer enables end users to load geometry files from a file; VolumeViewer renders a structured dataset using the GPU based volume rendering technique in VTK; and MooseViewer renders a multi-block unstructured dataset as a geometry or a volume, depending on the end user's interactive selections.

4.1 Immersive Environments

A variety of immersive environment display styles exist, from HMDs, to low-cost IQ-stations [46], to a four- or six-sided CAVE^{TM}. Immersive applications need to support a large number of immersive environments, as each has its strength and applicability in real world scenarios. We have tested our work in the following virtual environments:

- a four-sided CAVE^{TM};
- a low cost IQ-station; and
- an HTC VIVE HMD.

In the first two cases, the `vtkRenderingExternal` module was used with the Vrui VR toolkit to provide the configuration necessary to run the application. For HTC VIVE, we leveraged `vtkOpenVR`.

4.2 Vrui Implementation

The task of a VR toolkit is to shield an application developer from the particular configuration of an immersive environment, such that applications can be developed quickly and in a portable and scalable fashion. There are three important parts of this overarching goal are: encapsulation of the display environment, encapsulation of the distribution environment and encapsulation of the input device environment.

The Vrui VR toolkit supports fully scalable and portable applications that run on a range of immersive environments, starting from a laptop with a touchpad, to desktop environments with special input devices such as space balls, to full-blown immersive VR environments ranging from a single-screen workbench to a multi-screen tiled display wall or CAVE™. Applications using the Vrui VR toolkit are written without a particular input environment in mind, and Vrui-enabled immersive environments are configured to map the available displays and input devices to the application such that they appear to be written natively for the environment. For example, a Vrui application running on the desktop should be as usable and intuitive as any 3D application written specifically for the desktop.

We developed some example applications that serve as validation of this effort. There is an example within the VTK source tree for the `vtkRenderingExternal` module that renders a VTK sphere in an OpenGL Utility Toolkit (GLUT) window. We also developed three advanced applications that illustrate VTK rendering within a Vrui created OpenGL context. These applications exhibit varying capabilities of the VTK infrastructure leveraged by the `vtkRenderingExternal` module.

4.2.1 GeometryViewer

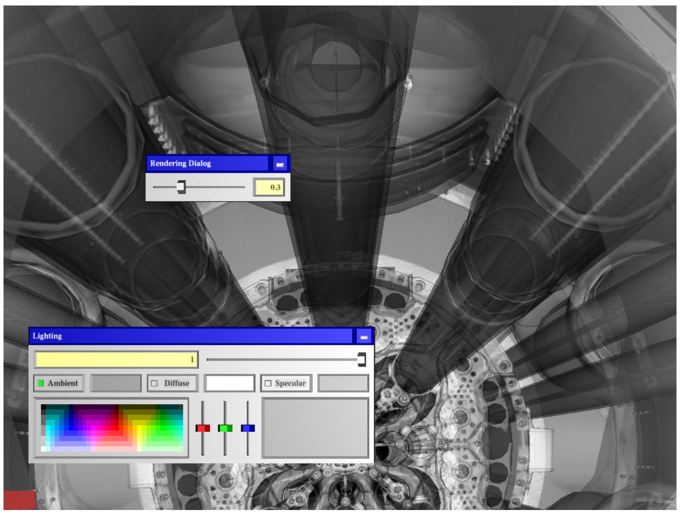

Figure 4: Geometry representing Idaho National Laboratory's Advanced Test Reactor (ATR) core. The geometry is used to virtually understand maintenance processes in this extreme environment

GeometryViewer [22] reads and renders a Wavefront (.obj) file that defines a geometry. It reads the file using the standard `vtkOBJFileReader` that creates `vtkPolyData` from the geometry. The `vtkPolyData` is then mapped, using the polydata rendering pipeline in VTK, as a `vtkActor` object. The main menu of the application allows the user to center the geometry to the screen as well as to change its representation. The *Center Display* button calculates the transformation from the current camera position and direction to the center position. The *Rendering Options* sub-menu allows the end user to change the opacity of the *vtkActor* object, leveraging our work on dual depth peeling, as well as its representation to either the points, the wireframe or the surface. In addition to VTK level modifications, the application has support for OpenGL level widgets (e.g., `glClipPlane`). This shows that native OpenGL operations can also be interactively performed when using the VTK rendering pipeline.

In Figure 4, we show the Vrui user interface UI with the *Rendering Options* dialog that allows us to adjust the transparency of the Advanced Test Reactor (ATR) core. In addition, we used the Vrui user interface to build an interface for the lighting color that seamlessly maps between Vrui and VTK.

This example has the functionality contained in the integration between Vrui and Delta3D presented in Sherman et al. [46]. Both applications were simple to develop, but the performance of GeometryViewer is faster, especially when rendering transparent geometry. In addition, GeometryViewer integrates the lighting between the VR toolkit and VTK, in contrast to the Vrui/Delta3D integration, which requires modification the material textures to provide false lighting.

4.2.2 VolumeViewer

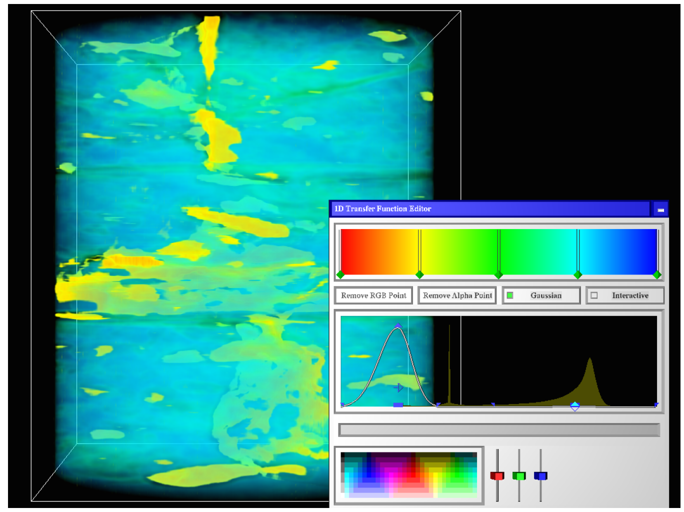

Figure 5: A digitized well "rock" core. The yellow isosurfaces isolate the oil trapped within the shale rock.

VolumeViewer [24] reads and renders VTK ImageData (.vti) files that define structured points datasets. The application instantiates a pipeline that allows volume rendering of the dataset. Several pre-defined color maps help to change the mapping of scalar values to colors. A *Transfer Function* editor allows changes to be made to the color and opacity of the rendered volume.

Figure 5 depicts our implementation of a transfer function editor using the Vrui user interface. In addition, we leverage `vtkFlyingEdges3D` to display the oil isosurfaces in yellow and blend the results in the volume.

Widgets such as `Isosurfaces`, `Contours` and `Slice` provide VTK level operations that can be carried out on the dataset. To circumvent possible interaction problems when dealing with large datasets, a low-resolution mode is provided that downsamples the dataset. This lets the end user fulfill actions quickly and then revert back to the full size, when they ready to visualize the complete dataset.

Immersive volume visualization applications are fundamental for a number of scientific research domains, and it's not surprising that there exists a number of partial solutions in the public domain. VolumeViewer has similar features contained in both Visualizer [5] and Toirt Samhlaigh [35]. Both applications used complex, from scratch, OpenGL algorithms that are time-consuming to develop. Visualizer uses a standard GPU OpenGL Shading Language (GLSL) shader-based ray-casting algorithm. Toirt uses an octree-based space-skipping GPU GLSL shader-based view-aligned algorithm to provide higher performance on larger volumes. In contrast, VolumeViewer requires less than a hundred lines of VTK API to implement roughly the same GPU GLSL shader-based ray casting used in Visualizer. The performance of VolumeViewer is equal to or exceeds that of the other tools. In addition, by modifying a line or two of VolumeViewer code, the application can accept data from any of the hundred-plus supported scientific data formats—a feat that would require the implementation of readers, data structures and algorithms in either of the other two applications.

4.2.3 MooseViewer

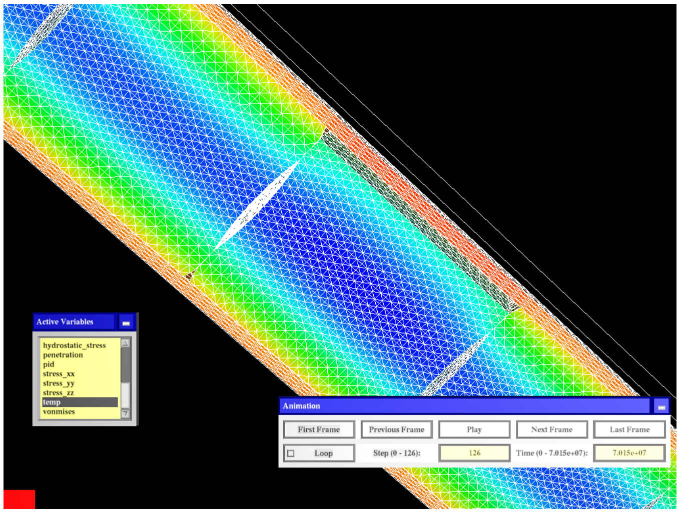

Figure 6: A MOOSE Framework application, BISON, simulates a nuclear pin with missing cladding on one of the fuel pellets.

MooseViewer [23] reads and displays Moose framework [14, 26] ExodusII (.ex2, .e) files in immersive environments. The application uses `vtkExodusIIReader` to read geometry defined in ExodusII files as well as associated attributes (e.g., temperature, burnup, etc.). The application permits only user-selected variables to be loaded as data arrays, thus reducing memory overhead. A "*Color By*" sub-menu is dynamically populated with user-selected variables. The sub-menu maps the chosen variable scalars to colors using the selected color map. An interesting capability of the application is animation of the dataset over time. The "*Animation*" dialog helps play through the time steps with controls for looping and stepping through the time steps.

In Figure 6, we see surface geometry colored by the selected temperature attribute animated using the "*Animation*" dialog.

Figure 7: A polygonal rendering of a sample dataset by VTK for OpenVR on HTC Vive.

Like Visualizer [5], MooseViewer provides a number of scientific visualization techniques for simulation data, but MooseViewer was created in weeks. Enhancements are added by simply plugging in alternative VTK algorithms to increase/scale performance. Although it is focused on the ExodusII file format, MooseViewer makes accepting data from other scientific data formats a simple task. MooseViewer outperforms Visualizer using the VTK OpenGL 3.2+ rendering pipeline, but pure rendering performance fails to highlight the unique immersive-specific level-of-detail algorithms in Visualizer. Visualizer is a fantastic immersive application. If an end user has data in one of its accepted formats, we encourage that person to leverage some of the unique Visualizer features.

4.3 OpenVR Implementation

This example creates a trivial VTK pipeline that reads a polygonal geometry file using the `vtkPLYReader` and maps it to the scene using `vtkOpenVR` classes described above. As seen in Figure 7, the rendering classes create a stereo pair from the view and warp it to the HTC Vive camera model. The example is available as a test case under the `vtkOpenVR` module in the VTK source.

The modest amount of code needed to put VTK generated polygons into the Vive HMD attests to the modularity and complete integration of the existing VR framework—in this case, `vtkOpenVR`.

5 Conclusion

As the user base for VR flourishes, there will be many new users who look to use VR as a tool for scientific visualization. Rather than write new algorithms and tools entirely from scratch, our extensions to the VTK system lowers the hurdles to cleanly meld community-tested high-quality visualization algorithms into existing VR integration libraries that can immersively render to all types of immersive systems, from large walk-in CAVE^{TM}style displays to consumer-grade HMDs designed for games and game ecosystems (such as SteamVR).

VTK has also been enhanced in ways that provide more efficient, and, therefore, faster rendering—orders of magnitude faster in many cases. Combined, we have moved VTK forward to where it can be **the** tool of choice for immersive visualization development.

Acknowledgments

The presented work was made possible due to support in part by funding from U.S. Department of Energy (Office of Nuclear Energy) Fast Track SBIR award DE-SC0010119, Dan Funk (program manager) and Idaho National Laboratory's Center for Advanced Energy Studies. Funding for embedding OpenVR, the OpenGL 3.2+ pipeline and SMP tools comes from the National Institutes of Health under the grant NIH R01EB014955 "Accelerating Community-Driven Medical Innovation with VTK."

References

[1] X. Amatriain, J. Kuchera-Morin, T. Hollerer, and S. T. Pope. The allosphere: Immersive multimedia for scientific discovery and artistic exploration. *IEEE MultiMedia*, 16(2):64–75, April 2009.

[2] L. Bavoil and K. Myers. Order independent transparency with dual depth peeling, 2008.

[3] R. Belleman. *Interactive Exploration in Virtual Environments*. PhD thesis, University of Amsterdam, 2003.

[4] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *Proceedings of the Virtual Reality 2001 Conference (VR'01)*, VR '01, pages 89–, Washington, DC, USA, 2001. IEEE Computer Society.

[5] M. I. Billen, O. Kreylos, B. Hamann, M. A. Jadamec, L. H. Kellogg, O. Staadt, and D. Y. Sumner. A geoscience perspective on immersive 3d gridded data visualization. *Comput. Geosci.*, 34(9):1056–1072, Sept. 2008.

[6] G. O. Bivins. A texture-based framework for improving cfd data visualization in a virtual environment. Master's thesis, Iowa State University, 2005.

[7] K. Blom. vjVTK: a toolkit for interactive visualization in Virtual Reality. In R. Hubbold and M. Ling, editors, *Proceedings of Eurographics Symposium on Virtual Environments (EGVE) '06*, 2006.

[8] R. Brady, J. Pixton, G. Baxter, P. Moran, C. S. Potter, B. Carragher, and A. Belmont. Crumbs: a virtual environment tracking tool for biological imaging. *Biomedical Visualization*, 82(30):18–25, 1995.

[9] F. P. Brooks, Jr., M. Ouh-Young, J. J. Batter, and P. Jerome Kilpatrick. Project grope — haptic displays for scientific visualization. *SIGGRAPH Comput. Graph.*, 24(4):177–185, Sept. 1990.

[10] K. M. Bryden, D. McCorkle, and A. Bryden. Ve-suite. Online at *http://www.vesuite.org/*, 2016.

[11] S. Bryson. Virtual reality in scientific visualization. *Commun. ACM*, 39(5):62–71, May 1996.

[12] facebook. Oculus. Online at *https://developer.oculus.com/*, 2016.

[13] A. Febretti, A. Nishimoto, V. Mateevitsi, L. Renambot, A. Johnson, and J. Leigh. Omegalib: A multi-view application framework for hybrid reality display environments. In *2014 IEEE Virtual Reality (VR)*, pages 9–14, March 2014.

[14] D. R. Gaston, C. J. Permann, J. W. Peterson, A. E. Slaughter, D. Andr, Y. Wang, M. P. Short, D. M. Perez, M. R. Tonks, J. Ortensi, L. Zou, and R. C. Martineau. Physics-based multiscale coupling for full core nuclear reactor simulation. *Annals of Nuclear Energy*, 84:45 – 54, 2015. Multi-Physics Modelling of {LWR} Static and Transient Behaviour.

[15] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1299–1308, Washington, DC, USA, 2013. IEEE Computer Society.

[16] K. Gruchalla. Immersive well-path editing: investigating the added value of immersion. In *Virtual Reality, 2004. Proceedings. IEEE*, pages 157–164. IEEE, 2004.

[17] D. Hannema. Interaction in Virtual Reality. Master's thesis, University of Amsterdam, 2001.

[18] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila. The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards. *SoftwareX*, 1?2:9 – 12, 2015.

[19] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proceedings of the SIGGRAPH Conference*, volume 1, pages 129–140, 2001.

[20] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, , and J. T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics (TOG)*, 21(3):693–702, 2002.

[21] Intel. TBB. Online at *https://www.threadingbuildingblocks.org/*, 2016.

[22] S. Jhaveri, D. Lonie, and P. O'Leary. GeometryViewer. Online at *https://github.com/VruiVTK/GeometryViewer*, 2016.

[23] S. Jhaveri, D. Lonie, and P. O'Leary. MooseViewer. Online at *https://github.com/VruiVTK/MooseViewer*, 2016.

[24] S. Jhaveri, D. Lonie, and P. O'Leary. VolumeViewer. Online at *https://github.com/VruiVTK/VolumeViewer*, 2016.

[25] O. Kreylos, G. Bawden, T. Bernardin, M. I. Billen, E. S. Cowgill, R. D. Gold, B. Hamann, M. Jadamec, L. H. Kellogg, O. G. Staadt, and D. Y. Sumner. Enabling scientific workflows in virtual reality. In *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*, VRCIA '06, pages 155–162, New York, NY, USA, 2006. ACM.

[26] I. N. Laboratory. MOOSE Framework. Online at *http://mooseframework.org/*, 2016.

[27] J. LaViola, Prabhat, A. Forsberg, D. H. Laidlaw, and A. van Dam. Virtual reality-based interactive scientific visualization environments. In *Interactive Visualization: A State-of-the-Art Survey*. Springer Verlag, 2008.

[28] J. Leigh, P. J. Rajlich, R. J. Stein, A. E. Johnson, and T. A. Defanti. Limbo/vtk: A tool for rapid tele-immersive visualization. In *CDROM proc. of IEEE Visualization '98, Research Triangle Park, NC*, pages 18–23, 1998.

[29] G. E. Marai, A. G. Forbes, and A. Johnson. Interdisciplinary immersive analytics at the electronic visualization laboratory: Lessons learned and upcoming challenges. In *IEEE VR Workshop on Immersive Analytics at 2016 IEEE Virtual Reality (VR)*, March 2016.

[30] P. McDowell, R. Darken, J. Sullivan, and E. Johnson. Delta3D: A Complete Open Source Game and Simulation Engine for Building Military Training Systems Perry McDowell. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 2(3), 2006.

[31] Mechdyne. CAVELib. Online at *https://www.mechdyne.com/software.aspx?name=CAVELib*, 2016.

[32] Mechdyne. Conduit. Online at *https://www.mechdyne.com/software.aspx?name=Conduit*, 2016.

[33] K. Moreland, C. Sewell, W. Usher, L. ta Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs, M. Larsen, C.-M. Chen, R. Maynard, and B. Geveci. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications*, 36(3):48–58, 2016.

[34] N. Ohno, A. Kageyama, and K. Kusano. Virtual reality visualization by CAVE with VFIVE and VTK. *Journal of Plasma Physics*, 72(6):1069–1072, 12 2006.

[35] P. O'Leary, W. Sherman, A. Murray, C. Riesenfeld, and V. Peng. Enabling scientific workflows using immersive microbiology. In *Workshop on Scientific Workflow with Immersive Interfaces for Visualization at EEE Visualization 2008 Conference*, page DVD, Columbus, OH, USA, October 2008. IEEE Press.

[36] OpenMP.org. OpenMP. Online at *http://www.openmp.org/*, 2016.

[37] R. A. Pavlik and J. M. Vance. VR JuggLua: A framework for VR applications combining Lua, OpenSceneGraph, and VR Juggler. In *SEARIS*, pages 29–35. IEEE, 2012.

[38] Prabhat, A. Forsberg, M. Katzourin, K. Wharton, and M. Slater. "A Comparative Study of Desktop, Fishtank, and Cave Systems for the Exploration of Volume Rendered Confocal Data Sets". *IEEE Transactions on Visualization and Computer Graphics*, 14(3):551–563, 2008.

[39] D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Woessner. COVISE in the CUBE: an environment for analyzing large and complex simulation data. In *Proceedings of the 2nd Workshop on Immersive Projection Technology*, 1998.

[40] K. Reda, A. Febretti, A. Knoll, J. Aurisano, J. Leigh, A. Johnson, M. E. Papka, and M. Hereld. Visualizing large, heterogeneous data in hybrid-reality environments. *IEEE Computer Graphics and Applications*, 33(4):38–48, 2013.

[41] K. Reda, A. Knoll, K. i. Nomura, M. E. Papka, A. E. Johnson, and J. Leigh. Visualizing large-scale atomistic simulations in ultra-resolution immersive environments. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 59–65, Oct 2013.

[42] Road to VR. Making Valve's OpenVR Truly Inclusive for VR Headsets. Online at *http://www.roadtovr.com/making-valves-openvr-truly-inclusive-for-vr-headsets*, 2015.

[43] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An object oriented approach to 3d graphics*. Kitware, Inc., 4th edition, 2004.

[44] J. P. Schulze, R. Niemeier, and U. Lang. The perspective shear-warp algorithm in a virtual environment. In *Proceedings of the Conference on Visualization '01*, VIS '01, pages 207–214, Washington, DC, USA, 2001. IEEE Computer Society.

[45] D. P. Shamonin. Vtkcave. Online at *http://staff.science.uva.nl/dshamoni/myprojects/VtkCave.html* (defunct), 2002.

[46] W. R. Sherman, P. O'Leary, E. T. Whiting, S. Grover, and E. A. Wernert. IQ-Station: A Low Cost Portable Immersive Environment. In G. Bebis, R. D. Boyle, B. Parvin, D. Koracin, R. Chung, R. I. Hammoud, M. Hussain, K.-H. Tan, R. Crawfis, D. Thalmann, D. Kao, and L. Avila, editors, *ISVC (2)*, volume 6454 of *Lecture Notes in Computer Science*, pages 361–372. Springer, 2010.

[47] TechViz. Techviz. Online at *http://www.techviz.net*, 2016.

[48] H. M. Tufo, P. F. Fischer, M. E. Papka, and K. Blom. Numerical simulation and immersive visualization of hairpin vortices. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[49] Valve. OpenVR. Online at *https://github.com/ValveSoftware/openvr*, 2016.

[50] T. van Reimersdahl, T. Kuhlen, A. Gerndt, J. Henrichs, and C. Bischof. Vista: a multimodal, platform-independent vr-toolkit

based on wtk, vtk, and mpi. In *Proceedings of the 4th International Immersive Projection Technology Workshop (IPT)*, 2000.

[51] R. Wang and X. Qian. *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, 2010.

[52] D. Zielinski, R. McMahan, S. Shokur, E. Morya, and R. Kopper. Enabling closed-source applications for virtual reality via opengl intercept techniques. In *SEARIS 2014 Workshop. Co-located with the IEEE VR 2014 conference*, page 8, 2014.