Enhancements to VTK enabling Scientific Visualization in Immersive Environments

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

Figure 1: HTC Vive town data.

Abstract

Modern scientific, engineering and medical simulations and/or experimental and observational data sensing/measuring devices produce enormous amounts of data. While statistical analysis provides insight into this data, it is scientific visualization that is tactically important for scientific discovery, product design and data analysis. Implementing scientific visualization algorithms from scratch can be a time consuming and redundant process in immersive application development. The open source Visualization Toolkit (VTK) community delivers state-of-the-art software to speedup application development. But, over the past two (almost three) decades, integrating VTK with a virtual reality environment has been completed to varying degrees of success. In this paper, we demonstrate two new approaches to simplify this amalgamation pushed into 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 StylesInput Devices and Strategies

1 Introduction

Bill, the following should end the introduction and spells out our contributions.

Our two approaches for the amalgamation of VTK and an immersive environment through a VR Toolkit, makes several contributions to immersive scientific visualization.

OpenGL context sharing. Our vtkRenderingExternal VTK module provides a complete integration including lights, interaction, picking and access to the entire VTK pipeline. This, in turn, enables simple utilization for application developers with any OpenGL-based VR Toolkit.

VR Toolkit embedding. The OpenVR VTK module supports several immersive environments now without the issues faced by previous work, and is a complete template for embedding other VR Toolkits within VTK in the future.

*e-mail: patrick.oleary@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: james.money@inl.gov

Enhanced performance. enhancements to VTK that significantly impact immersive environment application development. These enhancements include:

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

Finally, we have exposed the framework of our image-based approach to the scientist through an advanced selection interface that allows them to make sophisticated (time, storage, analysis, ...) decisions for the production of *in situ* visualization and analysis output.

In the sections that follow, we illustrate how our amalgamation of VTK and VR Toolkits supports our goals for enhance scientific visualization in immersive environments.

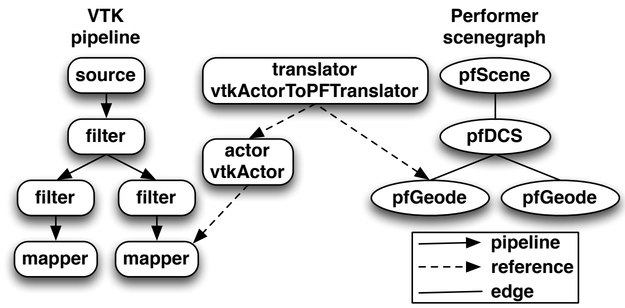

Figure 2: VTK, vtkActorToPF and Performer interaction diagram.

2 Related Work

The use of scientific visualization in immersive environments is simply natural, like bread and butter, while tactically important for scientific discovery, product design and data analysis. There are several high quality scientific visualization virtual reality applications that implement the applications from scratch using OpenGL directly [4, 19, 29, 26]. These are certainly exemplary and valuable applications. However, when building immersive applications, much like desktop applications, with scientific requirements, it is often more efficient to leverage the open source visualization toolkit (VTK) [28] for these development activities.

Throughout the past two plus decades, a number of research teams and developers have integrated VTK with an immersive environment to vary degrees of success. There exist three fundamental approaches to the proposed amalgamation:

- Geometry transport;
- OpenGL context sharing; and
- VR toolkit embedding.

At least one additional method exists, ***OpenGL intercept*** [12, 13, 38, 32, 23], or inserting middleware at runtime between the application and the graphics card, that enables virtual reality technologies for closed-source or proprietary applications without coding. These techniques are extremely valuable and allow scientist, engineers or medical researchers to enable their desktop applications in immersive environments with limited, but striking, interaction capabilities. Basic OpenGL, whatever that is, is sure to work using interception. But intercept methods will require more coding and tagging, if they work at all, for 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. Rather

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

Geometry transport A historic approach to integration was the `vtkActorToPF` library [20]. In this method, generation of visualization geometry is decoupled from rendering of the geometry (see Figure 2, which was recreated from Paul J. Rajlich).
VTK generates the geometry in the form of actors that consist of geometry and properties. `vtkActorToPF` transforms these actors into pf-Geodes (nodes) that are included in a Performer (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 [7] or directly into OpenGL [24]. Others have used VTK in a preprocessing step to produce geometries or textures eliminating the need for a direct connection to VTK[5].

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 a VR toolkit creates more obstacles than benefit from an application developers perspective, and is 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 their immersive application. In VTK, the renderer and render window classes are responsible for rendering scenes. VTK creates it's own window and associates an OpenGL context with that window to be used by the renderer. An OpenGL context represents: all of the state; the default framebuffer; and everything affiliated with OpenGL with respect to the renderer, window and application. The application developer would simply like to share the OpenGL context from the VR Toolkit with a third party rendering software (e.g. Delta3D [21], OpenSceneGraph [37] and VTK).

In previous work, by Sherman et al [31] and others, Delta3D and OpenSceneGraph were hacked to use the VR Toolkit's, like Vrui [17], window and associated OpenGL context. These solutions were limited in their integration. The lights, interaction and picking were not available to the application developer.

Our `vtkRenderingExternal` VTK module formalizes this integration providing lights, interaction and picking connectivity lacking in other implementations, while allowing 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 [35, 10, 30, 2]. To render in an immersive environment, derived classes of the `Renderer` and `RenderWindow` are created, which depends on fundamental calls to the VR toolkit. Thus, VTK-based applications can simply exchanged these two items to run on the desktop or immersive environment. VTK has evolved significantly over the years, as have the diversity of virtual reality products. vtkCave [33], for the CAVELib [22], followed by vjVTK [6] and VR JuggLua [25], for VRJuggler [3], created third party software essentially deriving RenderWindow and Renderer classes, but, from outside of VTK, lighting and interaction were not shared and/or troublesome.

In this work, we've created a new VTK module based on OpenVR [34]. OpenVR is a application programming interface (API) developed by Valve for supporting SteamVR (HTC Vive) and other virtual reality hardware [27]. The OpenVR module supports several immersive environments now without the issues faced by previous work, and provides a template for embedding other VR Toolkits within VTK in the future.

Enhanced performance Near real-time update of scientific visualization metaphors is crucial in these immersive environments. The field has scene several proposed solutions from decoupling rendering from processing to parallel visualization [8, 36]. But here is where using VTK for scientific visualization in immersive environments really stands apart. Instead of proposed solutions in a paper and existing only in a forgotten demo, the open source, community driven VTK provides state-of-the-art implementations that are only an API call away. VTK has been around since 1993 with over one hundred thousand repository commits from over two-hundred and fifty contributors. Having the latest algorithm implementation requires using the existing implementation in VTK or contributing the it to VTK.

We present enhancements to VTK that significantly impact immersive environment application development. These enhancements include the new default OpenGL 3.2+ pipeline, dual depth peeling for transparency, and symmetric multiprocessing (SMP) tools and algorithms.

3 APPROACHES

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

3.1 OpenGL context sharing

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

3.1.1 Architecture

Integrating VTK in external rendering systems required overriding some of the behavior of the `vtkRenderWindow`, `vtkRenderer`, and `vtkCamera` classes. A `Renderer` is attached to a `RenderWindow`, a `Mapper` to an `Actor`, and a `Camera` to a `Renderer`. In a typical VTK application the `RenderWindow` class is responsible for creating a rendering context, and defining width and height of the visualization viewport. The `Renderer` class is responsible for rendering one-or-more `Actors`. The `Actor` class is a drawable entity, which uses a `Mapper` to render specific data within a `Renderer`. Each of these components has their corresponding derived classes that implements the API using OpenGL, VTK's underlying graphics API. Using OpenGL provides VTK with the ability to use hardware acceleration that ultimately leads to better visualizations and near real-time performance as required by many interactive applications including the ones that are designed for immersive environments. In this design of VTK, each component participates in a specific way and communicates with other components via the public API provided by each. For instance, the `RenderWindow` typically creates the context in which `Renderer` draws drawable entities, `Actors`. A `RenderWindow` can have one or more `Renderers`. Each `Renderer` can make a decision on whether or not it should reset the buffers such as color or depth to its initial state while rendering one-or-more actors.

Since `vtkRenderWindow` typically creates the context, and `vtkRenderer` controls objects of a scene in a given viewport, the rendering pipeline is constructed with properties and other attributes set specifically to support this most general use case. However, in the case of external environments, the context is created outside of VTK, and non-VTK graphical elements (such as the 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`.

The `vtkExternalOpenGLRenderWindow` class is an extension to the `vtkGenericOpenGLRenderWindow`, 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. The `vtkExternalOpenGLRenderer` derives from `vtkOpenGLRenderer` 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 in the scene, of which VTK is unaware.

3.1.2 Implementation

One of the most important prerequisites of this work was seamless stereo rendering and user interaction with the two rendering systems.

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, we utilized the OpenGL context, set by the VR toolkit, to determine the the type of stereo (Quad Buffer, Side-by Side, or Top-Bottom stereo) and simply render using the OpenGL context, which sets active buffer, stenciling, etc. This is set only once immediately after the context has been created and maintained by the VR toolkit over time.

2D and 3D Interface Widgets It is unlikely that the VTK elements will be the only rendering in a scene. There will probably at least 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, leaving that to the display integration library or application. The depth buffer can then act to allow OpenGL elements to be mixed (composited) in three-space with closer elements occluding farther ones.

User Interactions Generally, in the case of a VR toolkit, interaction such as navigation in the scene space, grab, and rotation of various scene objects are handled by the system (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 was added. This class empowers the external application to manage camera interaction for VTK objects. We added a GL query in the external renderer, which uses the GL state system to get the projection and modelview transformation matrices. These two matrices determine the location and orientation of the object in the scene. The `vtkExternalOpenGLRenderer` sets these matrices on the `vtkExternalOpenGLCamera`. Setting these matrices directly on the camera leaves the camera parameters such as position, focal point, and view up direction to incorrect values, Therefore, we compute appropriate viewing coordinates for the camera by multiplying the modelview martix with the camera initial default position that is in OpenGL coordinate system. Once, everything is set on the camera, the navigation and lighting works as expected by the user.

The application itself handles the secondary kind of interactions such as interactive slicing and clipping of the scientific datasets. VTK provides classes (filters) to perform thresholding, clipping, slicing, etc. These filters take inputs such as thresholding value, slicing position and normal, and clip position. In our implementation, the application receives the tracker data (6 DOF), and, based on the mode the application is in, uses these information to set appropriate values on a specific filter. This integration is straightforward as our module makes the coordinate system consistent between the two rendering system.

3.1.3 Enabling `vtkRenderingExternal`

The `ExternalVTKWidget` class provides a one-stop solution to use all the new classes, described above, in an external application. The overarching application need just instantiate this class to use VTK's rendering capabilities. It creates a new external render window or uses one provided to it.

This work has been merged into the VTK as of release 7.0 available at www.vtk.org. To enable this module, set Module `vtkRenderingExternal` to `ON` (default is `OFF`).

3.2 VR Toolkit embedding (OpenVR)

To make it possible to use OpenVR-compatible devices with VTK, we embedded OpenVR into VTK within a module, called `vtkOpenVR`. Our goal is to allow VTK programs to use the OpenVR library with few changes, if any. If you link your executable to the `vtkOpenVR` module, the object factory mechanism should replace the core rendering classes (e.g., `vtkRenderWindow` and `vtkRender`) with the OpenVR-specialized versions in VTK.

3.2.1 Implementation

The `vtkOpenVR` module contains the following classes as drop-in replacements in VTK.

`vtkOpenVRRenderWindow` - This is a derived classes of the RenderWindow class. The current implementation creates one renderer that covers the entire window. As described in the Related work section, this class (and `vtkOpenVRRenderer`) is the location for embedding the VR toolkit, and handles the bulk of interfacing to OpenVR.

`vtkOpenVRRenderer` - This is a derived classes of the Render class. The `vtkOpenVRRenderer` class computes a reasonable scale and translation, and sets the results on `OpenVRCamera`. It also sets an appropriate default clipping range expansion. Again, this class (and `vtkOpenVRRenderWindow`) is the location for embedding the VR toolkit.

`vtkOpenVRCamera` - This is a derived classes of the Camera class.`vtkOpenVRCamera` gets the matrices from OpenVR to use for rendering. It contains a scale and translation that are designed to map world coordinates into the head-mounted display (HMD) space. Accordingly, the application developer can keep world coordinates in the units that are best suited to his/her problem domain, and the camera will shift and scale the coordinates into the units that make sense for the HMD.

`vtkOpenVRRenderWindowIneractor` - VTK is designed to pick and interact based on two-degrees of freedom, desktop X and Y mouse/window coordinates. In contrast, OpenVR provides X, Y and Z world coordinates and W, X, Y and Z orientations. The `vtkOpenVRRenderWindowInteractor` 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 can use them. `vtkOpenVRRenderWindowInteractor` supports multiple controllers through the standard PointerIndex approach that VTK uses for MultiTouch.

`vtkInteractorStyleOpenVR` - In concert with the `vtkOpenVRRenderWindowInteractor` class, we derived the `vtkInteractorStyleOpenVR` class. The `vtkInteractorStyleOpenVR` class uses X, Y and Z world coordinate positions and W, X, Y and Z orientations to adjust `Actors`. This class provides a nice grab-and-move style of interaction that is common to OpenVR and other VR toolkits.

`vtkOpenVRPropPicker` - Finally, the derived `vtkOpenVRPropPicker` class determines what `Actors` or `Props` VTK picks. Note that `Prop` is an abstract superclass for any objects that can exist in a rendered scene (either 2D or 3D), and defines the API for picking, LOD manipulation, and common instance variables that control visibility, picking, and dragging. The `vtkOpenVRPropPicker` class uses the X, Y and Z world coordinate as the picking value as opposed to an intersecting a ray, which is slower.

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

3.2.2 Enabling `vtkOpenVR`

To use `vtkOpenVR`, first download the master branch of VTK from the VTK repository on GitHub (see www.vtk.org). The remote module for `vtkOpenVR` can be found at https://goo.gl/0jem0V. Place this file into the Remote folder of your VTK source tree. `vtkOpenVR` also requires that you download two external libraries: Simple DirectMedia Layer 2 (SDL2) and OpenVR. To enable this module, set Module `vtkOpenVR` to `ON` (default is `OFF`).Make sure you build an optimized version of VTK to maximize performance.

3.2.3 Future Developments

The `vtkOpenVR` module is currently in the alpha phase and currently tested on the HTC Vive virtual reality system. Moving forward, we look to add support for the OpenVR overlay, which is great for displaying a user interface. We also aim to make the module faster and include more event interactions.

3.3 Performance enhancements

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

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 [11].

The results have been nothing short of spectacular. For polygon rendering demonstrates a ten times speedup for first frame rendering followed by a two-hundred times speed up for subsequent frames for one to thirty million triangles. The previous volume rendering was also graphics processing unit (GPU) aware, and, thus, the improvement is a modest but substantial two times speedup.

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

3.3.2 Dual-Depth Peeling

As we developed several example programs leveraging the `vtkRenderingExternal` module, we found that the rendering performance slowed as transparency was introduced into the scene.

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 by OpenGL 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** - The fragments are processed (blended using just alpha) in random order. It is very fast, but 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 provides generally consistent results with some artifacts where primitives overlap.
- **Depth Peeling** - Extract and blend fragments in a multi-pass render, and, therefore, requires multiple geometry render passes.

VTK by default uses depth peeling. To enhance rendering performance with transparancy we implemented `vtkDualDepthPeelingPass`, which was originally proposed by nVidia in 2008 [1]. Dual-depth peeling extends traditional depth peeling by extracting two layers of fragments per-pass: from the front and back simultaneously. Uses a two-component depth buffer to track of peel information and three types of geometry passes:

- **InitializeDepth** - Initializes buffers using opaque geometry information.
- **Peeling** - Repeated pass that extracts and blends translucent geometry peels. It extracts both near and far peels while blending far peels into accumulation buffer.
- **AlphaBlending** - An optional pass to clean up unpeeled fragments and used with occlusion thresholds.

This work provides a two times speedup 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 which under the hood uses different libraries such as TBB, OpenMP and X-Kaapi. The typical target application is coarse-grained shared-memory computing as provided by mainstream multicore, threaded CPUs such as Intel's i5 and i7 architectures.

For several of the example programs utilizing the `vtkRenderingExternal` module, we leveraged a new contouring algorithm in VTK that is readily parallelizable using `vtkSMPTools` and still incredibly efficient in serial mode, `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 usefulness of the enhancements to VTK enabling scientific visualization, and to test various use cases, we have implemented three kinds of applications for the `vtkRenderWindow` approach: GeometryViewer, VolumeViewer, and MooseViewer using VRUI as the VR toolkit, and one simple example using `vtkOpenVR`. As the name suggests, the GeometryViewer enables end-users to load geometry files from the disk, the VolumeViewer renders a structured dataset using VTK's GPU-based volume rendering technique, and MooseViewer renders a multi-block unstructured dataset as geometry or volume depending on the end-user's interactive selections. Details on each of these applications or examples with screen captures are provided in the next few sub-sections.

4.1 Immersive Environments

A variety of immersive environment exists from head-mounted displays (HMD) to low cost IQ station [31] to four or six sided CAVEs. It is important for any immersive application to support a large number of immersive environments as each has their strength and applicability in real world scenarios. We have tested our work in following virtual environments:

- A four-sided CAVE,
- A low cost IQ station, and
- A HTC VIVE.

In the first two cases, the `vtkRenderWindow` module was used with the Vrui VR toolkit provided the configuration necessary to run the application. For the 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. 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 aims to support fully scalable and portable applications that run on a range of immersive environments starting from a laptop with a touchpad, over 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 it runs on. 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 the said work. There is an example within the VTK source tree for `vtkRenderingExternal` module that renders a VTK sphere in a GLUT window. Three advanced applications are also developed 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

GeometryViewer [14] reads in and renders a Wavefront (.obj) file that defines geometry. The file is read using the `vtkOBJFileReader` that creates `vtkPolyData` from the geometry. The `vtkPolyData` is then mapped using VTK's polydata rendering pipeline as a `vtkActor`. The main menu of the application allows the user to center the geometry to the screen as well as 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*, leveraging our work on dual depth-peeling, as well as its representation to either points, wireframe or surface. In addition to VTK level modifications, the application has support for OpenGL level widgets

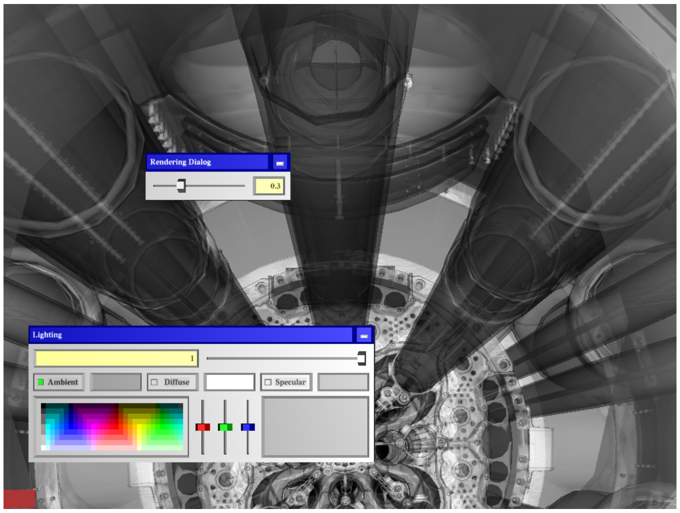

Figure 3: The geometry represents Idaho' Nation Laboratory's advanced test reactor (ATR) reactor core, and is used to virtually understand maintenance processes in this extreme environment

(e.g. `glClipPlane`). This shows that native OpenGL operations can also be interactively performed when using the VTK rendering pipeline.

In Figure 3, we show Vrui's user interface (UI) showing rendering options dialogue allowing us to adjust the transparency of the ATR reactor core. In addition, we used Vrui's UI to build an interface into the lighting color.

4.2.2 VolumeViewer

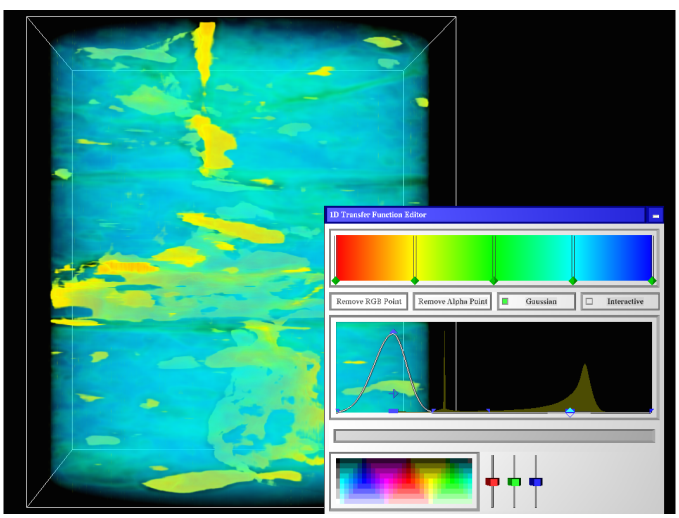

Figure 4: This is a digitized well "rock" core. The yellow isosurfaces isolate the oil trapped within the shale rock.

VolumeViewer [16] reads in 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 change mapping of scalar values to colors. A *Transfer Function Editor* allows varying color and opacity of the rendered volume.

Figure 4 depicts our implementation of a transfer function editor using Vrui's UI. In addition, we are leveraging the `vtkFlyingEdges3D` to display the oil isosurfaces in yellow blended in the volume.

Widgets like Isosurfaces, Contours, 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 down samples the dataset. This lets the end-user fulfill actions quickly that would otherwise take more time on the actual dataset, and then revert back to the actual size when done to visualize the output of performed actions at full scale.

4.2.3 MooseViewer

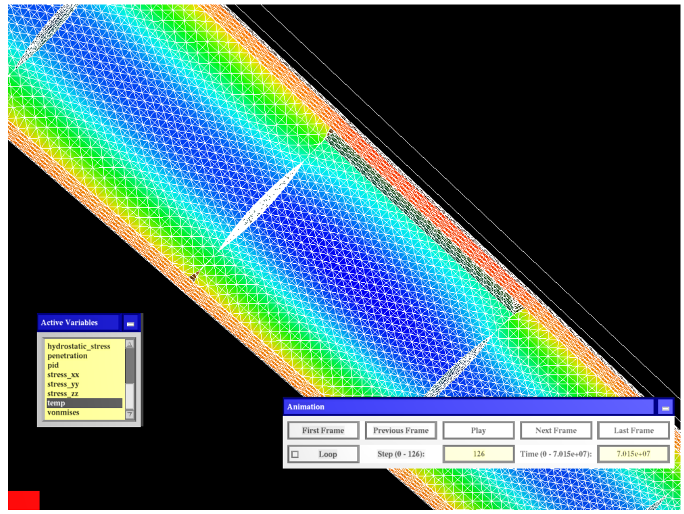

Figure 5: A MOOSE Framework application BISON simulates a nuclear pin with missing cladding on one of the fuel pellets.

MooseViewer [15] brings the ability of reading and displaying Moose framework [9, 18] ExodusII (.ex2, .e) files to 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 that maps the chosen variable scalars to colors using the *Color Map* selected. An interesting feature of the application is animation of the dataset over time. The *Animation Dialog* helps play through the time steps in the data file sequentially along with controls for looping over and stepping through the time steps.

We see, in Figure 5, surface geometry colored by the selected temperature attribute animated using the *Animation Dialog*.

4.3 OpenVR Implementation

4.4 Ply-file Example

5 Conclusion

Bill, the conclusion goes here.

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 from Idaho National Laboratory's Center for Advanced Energy Science. 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] L. Bavoil and K. Myers. Order independent transparency with dual depth peeling. Online, 2008.

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

[3] 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.

[4] M. Billen, O. Kreylos, B. Hamann, M. Jadamec, L. Kellogg, O. Staadt, and D. Sumner. A geoscience perspective on immersive 3d gridded data visualization. *Computers and Geosciences*, 34(9):1056–1072, 2008.

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

[6] 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.

[7] K. M. Bryden, D. McCorkle, and A. Bryden. Ve-suite. Online, 2016.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] S. Jhaveri, D. Lonie, and P. O'Leary. GeometryViewer. Online, 2016.

[15] S. Jhaveri, D. Lonie, and P. O'Leary. MooseViewer. Online, 2016.

[16] S. Jhaveri, D. Lonie, and P. O'Leary. VolumeViewer. Online, 2016.

[17] 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.

[18] I. N. Laboratory. MOOSE Framework. Online, 2016.

[19] 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.

[20] 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.

[21] 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.

[22] Mechdyne. CAVELib. Online, 2016.

[23] Mechdyne. Conduit. Online, 2016.

[24] 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.

[25] 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.

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

[27] Road to VR. Making Valve's OpenVR Truly Inclusive for VR Headsets. Online, 2015.

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

[29] 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.

[30] D. P. Shamonin. Vtkcave. Online (defunct), 2002.

[31] 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.

[32] TechViz. Techviz. Online, 2016.

[33] 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.

[34] Valve. OpenVR. Online, 2016.

[35] 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.

[36] 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 *Fourth International Immersive Projection Technology Workshop (IPT2000)*, 2000.

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

[38] 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.