Enhancements to VTK enabling Scientific Visualization in Immersive Environments

Category: Systems

Figure 1: HTC Vive town data.

ABSTRACT

Modern scientific, engineering and medical computational simulations and experimental and observational data sensing/measuring devices produce enormous amounts of data. While statistical analysis is one tool that provides insight into this data, it is scientific visualization that is tactically important for scientific discovery, product design and data analysis. But these benefits are impeded when the scientific visualization algorithms are implementing from scratch — a time consuming and redundant process in immersive application development. This process then can greatly benefit by leveraging the state-of-the-art open source Visualization Toolkit (VTK) and it's community. But, over the past two (almost three) decades, integrating VTK with a virtual reality environment has only been attempted to varying degrees of success. In this paper, we demonstrate two new approaches to simplify this amalgamation of 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 StylesInput Devices and Strategies

1 INTRODUCTION

There is a growing body of evidence demonstrating measurable benefits attained when exploring scientific data using immersive interfaces. Molecular research at UNC [**?**], genetics at NCSA [**?**], oil well placement at the University of Colorado [**?**], and confocal microscopy data at Brown University [**?**], to name but a few. We know too that while not scientifically verifiable, any time a scientist expresses a case where they "discovered" some relationship in their data while immersed in a virtual reality 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 the cost. And 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 often 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 a programming-level API that provides quick access to an expanse of scientific visualization rendering algorithms, as well as components for displaying and interacting with the results on a desktop display. However, while the concept of combining VTK with VR was 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 always rendering into windows with graphics contexts created by VTK itself, there is now the option to "externally" render into contexts provide by a collaborating system, or even integrate a VR system directly into VTK.

Immersive visualization efforts are often associated with research facilities that provide large-scale VR systems such as CAVEs^{TM}and other large-screen walk-in displays. 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, and while technically this is possi-

ble, the consumer systems offer a simpler approach that will entice many devlopers to follow that path. Thus we offer two approaches, one that addresses the simplier solution of integrating directly with OpenVR, and other that allows integration into any full-fledged VR integration library capable of interfacing with CAVE-style and HMD displays.

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

VR Toolkit embedding. The OpenVR VTK module supports several immersive environments directly without the issues faced by previous work, and is 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 These enhancements include:

- A 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 an 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 support our goals for enhancing scientific visualization through immersive environments.

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 created from scratch using OpenGL directly [**?**, **?**, **?**, **?**]. 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) [**?**].

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 bring VTK into an immersive rendering 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, and, therefore, we review related work for these areas.

Geometry transport An early approach to VTK-VR integration was the `vtkActorToPF` library [**?**]. In this method, the generation of visualization geometry is decoupled from the rendering of the geometry (see Figure 2). VTK generates the geometry in the form of actors that consist of polygons and properties. `vtkActorToPF` transforms these actors into pfGeodes (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 [**?**] or directly into OpenGL [**?**]. Others have used VTK in a preprocessing step to produce geometries or textures eliminating the need for a direct connection to VTK[**?**].

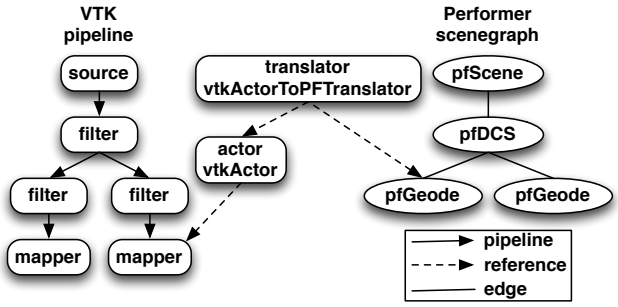

Figure 2: VTK, vtkActorToPF and Performer interaction diagram. (Recreated from Paul Rajlich 2.)

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 benefits 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 [**?**], OpenSceneGraph [**?**] and VTK).

In previous work, by Sherman et al. [**?**] and others, Delta3D and OpenSceneGraph were quickly modified to instead use windows and associated OpenGL contexts of a VR integration library such as Vrui [**?**]. These solutions are generally limited in their integration. Specifically, a rendering library that is unaware of the actual viewing matrix will generally not calculate lighting correctly, and picking input operations do not conform to the shifted rendering.

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 [**?**, **?**, **?**, **?**]. To render in an immersive environment, derived classes of the `vtkRenderer` and `vtkRenderWindow` are created, which depends on fundamental calls to the VR toolkit. Thus, VTK-based applications can simply exchange 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 [**?**], for the CAVELib [**?**], followed by vjVTK [**?**] and VR JuggLua [**?**], for VRJuggler [**?**], created third party software essentially deriving `vtkRenderWindow` and `vtkRenderer` classes, but, from outside of VTK; lighting and interaction were not shared and resulted in troublesome behavior.

In this work, we've created a new VTK module based on OpenVR [**?**]. OpenVR is an application programming interface (API) developed by Valve for supporting their SteamVR ecosystem, compatible with the HTC Vive and other virtual reality hardware [**?**]. 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.

OpenGL intercept A fourth possible means for melding VTK into a virtual environment system is the *OpenGL intercept* method

[**?**, **?**, **?**, **?**, **?**]. Here, at runtime, middleware is inserted between the application and the graphics card. With this technique, closed-source applications can be rendered with the head-tracked perspective rendering overriding the internal view matrix to provide the virtual reality 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 allow scientists, engineers or medical researchers to interact with their desktop tools in a whole new way. However, many of these methods lack full functionality in the immersive environments, which limits the usefulness to end users. Pure OpenGL, without any modifications or additions, is sure to work using interception. The difficulty of using intercept methods is that they require more coding and tagging, and are not guaranteed to work at all, and this is becoming more difficult with OpenGL 3.0+, especially when using a core OpenGL profile.

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 [**?**, **?**]. This effort stands apart from all these previous efforts, which valiant as they were, ultimately have been 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 hasn't been available for techniques that perform functions outside the bounds of the API design, often accessing internal features that do not have the assurance of stability. As a commercially supported open-source tool, VTK's rendering performance is continually being advanced. 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 algorithm to VTK.

We present internal 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 the existing architecture. VTK provides a well-defined rendering pipeline through the `RenderWindow`, `Renderer`, `Camera`, `Actor`, and `Mapper` classes. This precise pipeline definition and 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 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 interface-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 into 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` and managing the viewport within the `RenderWindow`. The `Actor` class is a drawable entity, which uses a `Mapper` to render specific data within a `Renderer`. Figure 3 shows the classes and interactions between them.

Each of these components has its corresponding derived classes that implement 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 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, the `RenderWindow` typically creates the context in which `Renderer` draws drawable entities, i.e. `Actors`.

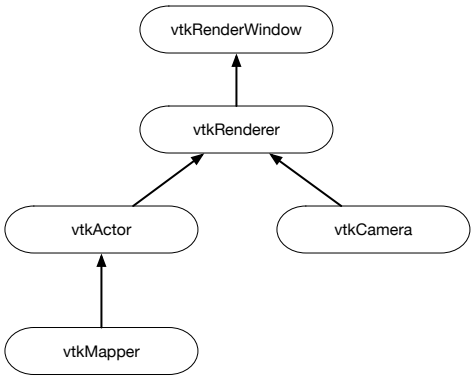

Figure 3: vtkRenderWindow, vtkRenderer, vtkCamera, vtkActor, and vtkMapper interaction diagram.

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`.

`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 in the scene, of which VTK is unaware.

vtkExternalOpenGLCamera - This class inherits `vtkCamera` and provides the ability to set the projection and modelview matrices on the camera. This allows the external rendering framework to 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 VTK's external rendering capabilities. The `ExtenalVTKWidget` creates a new external render window or uses one provided to it from the external library / application.

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 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 is 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 in addition to the VTK elements. 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 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 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 user's eye (camera) 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 matrix with the camera initial default position in the 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 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, 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 the VTK as of release 7.0 available at `http://www.vtk.org`. To enable this module when compiling VTK with `CMake`, set `Module_vtkRenderingExternal` to `ON` (default is `OFF`).

3.2 VR Toolkit Embedding (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 Valve's SteamVR. 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-off — for developers who don't already have expertise in a full-fledged VR integration library — is avoiding the programming of the alternative VR integration library, and immediately gaining access to HMDs compatible with OpenVR, but not to other VR display systems.

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 will replace the core rendering classes (e.g., `vtkRenderWindow` and `vtkRenderer`) 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 class 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 class 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 class 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 HMD space. Accordingly, the application developer can keep world coordinates in the units best suited to their problem domain, and the camera will shift and scale into 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 3D world coordinates and 3D orientations. The `vtkOpenVRRenderWindowInteractor` class catches controller events and converts them to mouse/window events. In addition, this class also stores the world coordinate positions for the styles or pickers that need 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 to use 3D world coordinates to adjust `Actor`s. This class provides a common grab-and-move style of interaction that is common to OpenVR and other VR toolkits.

vtkOpenVRPropPicker - Finally, the derived `vtkOpenVRPropPicker` class determines what `Actor`s or `Prop`s VTK picks. Note that `Prop` is an abstract superclass for any object 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 3D 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 seamless access to cameras, lighting, interaction and the complete VTK pipeline.

3.2.2 Enabling `vtkOpenVR`

To use VTK with OpenVR, first download the master branch of VTK from the VTK repository on GitHub (see `http://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. You must also install two external libraries: Simple DirectMedia Layer 2 (SDL2) and OpenVR. To enable this module, use `CMake` to set `Module_vtkOpenVR` to `ON` (default is `OFF`). Ensure you build an optimized version of VTK to maximize performance while using these new capabilities.

3.2.3 Future Developments

The `vtkOpenVR` module is currently in the alpha phase and has been tested on the HTC Vive HMD. Moving forward, we look to add support for the OpenVR overlay, which provides support for user interface components. We also expect to make the module faster and include more event interactions.

3.3 Performance enhancements

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. VTK provides state-of-the-art implementations accessible via an API call. The benefit in 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.

To allow VTK to function at levels needed for head-tracked rendering, many other enhancements have been added 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 [**?**].

The results have been nothing short of spectacular. Polygon rendering demonstrates a ten fold speedup for first frame rendering followed by a two-hundred fold 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 fold speedup.

To realize these performance enhancements, VTK now uses 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 currently provide a suitable OpenGL context. But, 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 leveraging 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** - 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 transparency we implemented `vtkDualDepthPeelingPass`, which was originally proposed by nVidia in 2008 [**?**]. 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 algorithm provides a two fold 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, 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 of our VTK enhancements provide scientific visualization efforts, and 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 applications: GeometryViewer, VolumeViewer, and MooseViewer. As the name suggests, the GeometryViewer enables end-users to load geometry files from a file, 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.

4.1 Immersive Environments

A variety of immersive environment display styles exist from head-mounted displays (HMD) to low cost IQ-station [**?**] and even four or six sided CAVEs. Immersive applications need 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
- an HTC VIVE.

In the first two cases, the `vtkRenderingExternal` 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 supports 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. 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 this effort. 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 [**?**] reads and renders a Wavefront (.obj) file that defines a geometry. The file is read using the standard `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 (e.g. `glClipPlane`). This shows that native OpenGL operations can also be interactively performed when using the VTK rendering pipeline.

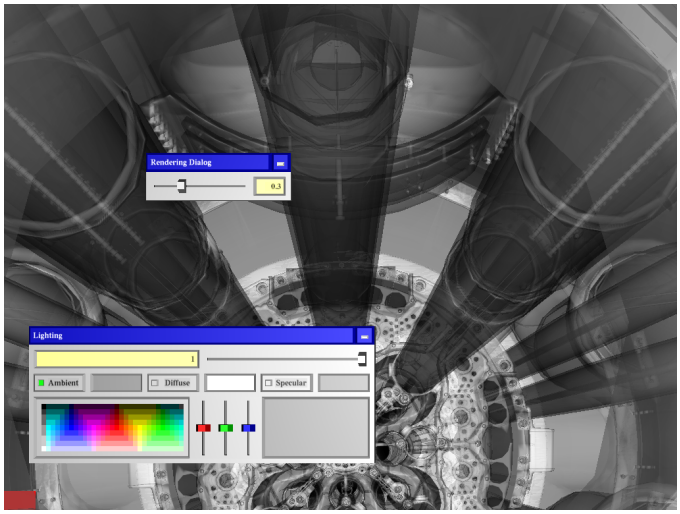

Figure 4: The geometry represents Idaho' Nation Laboratory's Advanced Test Reactor (ATR) reactor core, and is used to virtually understand maintenance processes in this extreme environment

In Figure 4, we show Vrui's user interface (UI) showing rendering options dialog allowing us to adjust the transparency of the ATR reactor core. In addition, we used Vrui's UI to build an interface for the lighting color.

4.2.2 VolumeViewer

VolumeViewer [**?**] 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 predefined *Color Maps* help change mapping of scalar values to colors. A *Transfer Function Editor* allows changes to the color and opacity of the rendered volume.

Figure 5 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 while blending the results in the volume.

Widgets such as 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 full dataset, and then revert back to the full size when ready to visualize the complete dataset.

4.2.3 MooseViewer

MooseViewer [**?**] brings the ability of reading and displaying Moose framework [**?**, **?**] ExodusII (.ex2, .e) files to immersive en-

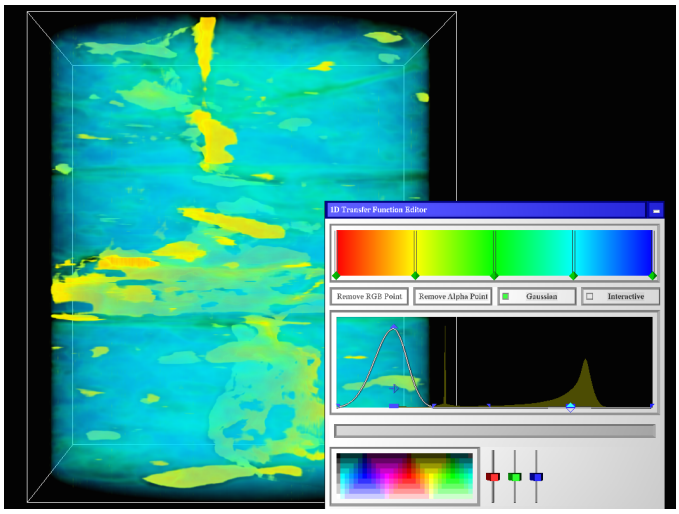

Figure 5: This is a digitized well "rock" core. The yellow isosurfaces isolate the oil trapped within the shale rock.

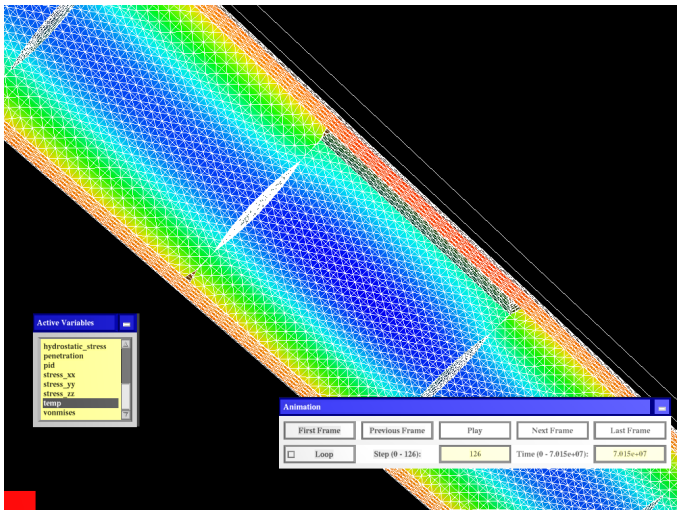

Figure 6: A MOOSE Framework application BISON simulates a nuclear pin with missing cladding on one of the fuel pellets.

vironments. 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 selected "*Color Map*". An interesting feature 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.

We see, in Figure 6, surface geometry colored by the selected temperature attribute animated using the "*Animation Dialog*".

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 the `vtkOpenVR` classes described above. As seen in Figure 7, the rendering classes create a stereo pair from the view and warp it to the camera model of the HTC Vive. The example is available

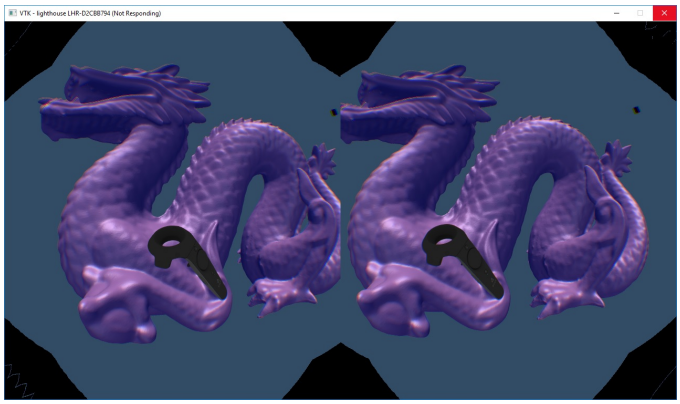

Figure 7: Polygonal rendering of a sample dataset by VTK for OpenVR on HTC Vive.

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 an existing VR framework — in this case `vtkOpenVR`.

5 Conclusion

As the user base for virtual reality flourishes, there will be many new users looking 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-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

Acknowledgements removed for review copy.

References

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

[8] 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.

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

[35] 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.

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

[37] 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.

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

[39] 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.

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

[41] 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.