Practical Volume Rendering using Programmable Shaders in VTK for Scientific and Medical Computing

Aashish Chaudhary, Sankhesh J. Jhaveri*, Alvaro Sanchez, Lisa S. Avila, Kenneth M. Martin, David Lonie, Marcus D. Hanwell

Kitware, Inc., 28 Corporate Drive, Clifton Park, NY 12065, USA

Abstract

The Visualization Toolkit (VTK) is a popular cross-platform, open source toolkit for scientific and medical data visualization, processing and analysis. It provides a wide variety of data formats, algorithms and rendering of both polygonal and volumetric data. VTK's volume rendering module boasts a comprehensive set of features such as plane clipping, color and opacity transfer functions, lighting and other controls needed for visualization. However, due to VTK's legacy OpenGL backend and its reliance on a deprecated API, volume rendering would fail to take advantage of the latest improvements in graphics hardware and the flexibility of a programmable pipeline. Additionally, this dependence on an antiquated pipeline would pose a restriction on its reliability when running on recent target platforms and limit its scope. We developed a new and improved volume rendering module for VTK's OpenGL2 backend, which also augments the GPU-based implementation with features such as fast volume clipping, gradient magnitude based opacity modulation, render to texture and hardware-based volume picking.

Keywords: Visualization Toolkit, Data analysis, Volume rendering, Scientific Visualization,

1. Introduction

VTK is an open source cross-platform software system used for scientific data processing, analysis and visualization. It was originally developed to provide examples accompanying a book on object oriented computer graphics programming [Schroeder et al., 2006; Geveci and Schroeder, 2012]. Since its inception, VTK has a long history of volume rendering and, unfortunately, that history is evident in the large selection of classes available to render volumes. Each of these methods was state-of-the-art at the time it was introduced, but given VTK's 20+ year history, many of these methods are now quite obsolete. Recently, there has been a major effort [Hanwell et al., 2015] undertaken to re-write the rendering backend from using legacy deprecated OpenGL API to a more modern programmable pipeline based OpenGL API [Shreiner et al., 2013], to support latest technological advances in the graphics hardware industry. One of the goals of this work was to reduce the number of volume mappers to ideally just two: one that supports accelerated rendering on the Graphics Processing Unit (GPU) and another that works in parallel on the Central Processing Unit (CPU). The `vtkSmartVolumeMapper` would help application developers by automatically choosing between these techniques based on the system configuration.

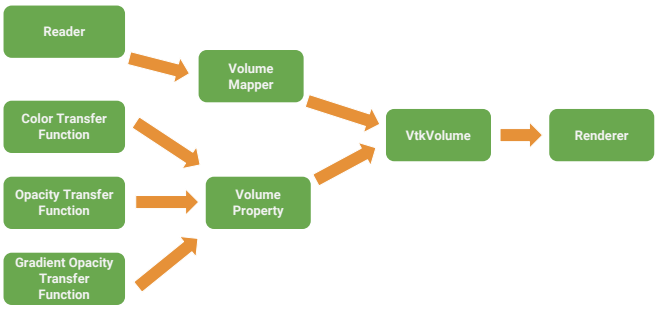


Figure 1: VTK pipeline for volume rendering which is similar to VTK polygonal rendering with differences such as transfer functions are defined on the property object.

The main objective of this effort is to create a cross-platform, multi-functional and high-performance volume renderer that works in both serial and parallel mode (for example in ParaView [Ahrens et al., 2005; Ayachit, 2015]). To achieve this, we have created a replacement for the OpenGL fixed pipeline based vtkGPUVolumeRayCastMapper. The new mapper, which shares the same name but uses the OpenGL programmable pipeline, can be used via `vtkSmartVolumeMapper` or instantiated directly and replaces the old `vtkGPUVolumeRayCastMapper`.

*Corresponding Author

Email addresses: `aashish.chaudhary@kitware.com` (Aashish Chaudhary), `sankhesh.jhaveri@kitware.com` (Sankhesh J. Jhaveri), `alvaro.sanchez@kitware.com` (Alvaro Sanchez), `lisa.avila@kitware.com` (Lisa S. Avila), `ken.martin@kitware.com` (Kenneth M. Martin), `david.lonie@kitware.com` (David Lonie), `marcus.hanwell@kitware.com` (Marcus D. Hanwell)

Availability of the new mapper with new OpenGL-VTK implementation improved the management of textures in the mapper and benefited both forms of rendering (geometry and volume) by sharing common code between them. While volume ray-casting itself is a well-known technique, developing a volume renderer that works with variety of data formats and types, supports many essential features for medical and scientific computing, works on the main commercial platforms (such as Windows, Mac, and Linux) and performs well at interactive frame rates with very large datasets is still a challenging task that requires an in-depth knowledge of the data, graphics pipeline, the VTK framework and the user requirements. In the next section, we will cover technical details of our work that resulted in a sophisticated volume renderer for the VTK community.

2. Approach

The new `vtkGPUVolumeRayCastMapper` uses a ray casting technique [Engel et al., 2006] for volume rendering which is a state-of-the-art for volume rendering on modern graphics platforms. Algorithmically, at a high level, it is similar to the older version of this class (although with a fairly different OpenGL implementation since that original class was first written over a decade ago and used GPU assembly code). One of the main reason we chose to use ray casting due to the flexibility of this technique, which enables us to support all the features of the software ray cast mapper but with the acceleration of the GPU. Ray casting is an image-order rendering technique, with one or more rays cast through the volume per image pixel. VTK is inherently an object-order rendering system, where the GPU renders all graphical primitives (points, lines, triangles, etc.) represented by vtkProp(s) in the scene in one or more passes (with multiple passes needed to support advanced features such as depth peeling for transparency).

The image-order rendering process for vtkVolume is initiated when the front-facing polygons of the volumes bounding box are rendered with a custom fragment program. This fragment program is used to cast a ray through the volume at each pixel, with the fragment location indicating the starting location for that ray. The volume and all the various rendering parameters are transferred to the GPU through the use of textures (3D for the volume, 1D for the various transfer functions) and uniform variables. Steps are taken along the ray until the ray exits the volume, and the resulting computed color and opacity are blended into the current pixel value. Note that volumes are rendered after all opaque geometry in the scene to allow the ray casting process to terminate at the depth value stored in the depth buffer for that pixel (and, hence, correctly intermix with opaque geometry).

In addition to providing supported features of the old mapper, the new mapper added new capabilities such as clipping on GPU, gradient opacity, and volume picking amongst many others. In the next few sections, we will cover each of these features in detail.

2.1. Single Pass

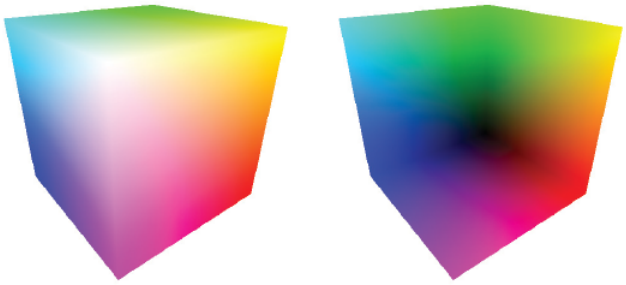


Figure 2: Front and back faces are rendered for start and end position of the ray.

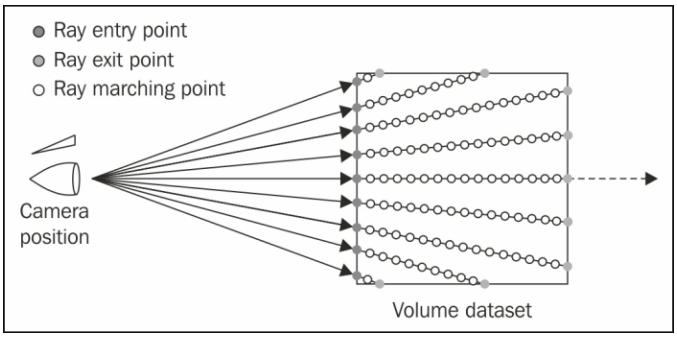


Figure 3: Implementing volume rendering using single-pass GPU ray casting.

In a ray-casting algorithm, the entry and the exit point into the volume is needed to determine when to stop the ray-marching. To determine the entry and the exit point, one approach is to render the geometry of the volume bounding box of the volume twice. In the first pass, the front face of the geometry is rendered and in the second pass the backface is rendered as shown in Figure 2. Using the interpolated vertex position and texture lookup, the start and end positions is computed. Instead of this, in `vtkGPUVolumeRayCastMapper`, entry and exit points are computed based on the fact that the texture extents of the volume is within vec3(1.0), vec3(-1.0) range (as shown in Figure 3). The code below is showing the fragment shader piece that determines whether or not to stop marching the rays depending on the value of stop.

```
bool stop = any(greaterThan(g_dataPos,
                            ip_texMax)) ||
            any(lessThan(g_dataPos,
                         ip_texMin));
```

Listing 1: Ray stop determination

The advantage of such approach is that it requires one less pass and is faster than other approaches since there is no texture generation or lookup happens for determining the termination of the ray.

2.2. Dynamic Shader Generation

In the new `vtkOpenGLGPUVolumeRayCastMapper` all the operations are performed on the GPU. The advantage of this approach was a more streamlined code that is easier to maintain and debug. This approach also provided an opportunity to rework how to support different features without having too many branches in the shader code or having to send all the options to the shader because that would have been detrimental to the performance. In the new `vtkOpenGLGPUVolumeRayCastMapper`, the shader is dynamically composed by the mapper. For this to work, we have introduced tags in a vertex or fragment shader which are then replaced by the `vtkShaderComposer` depending on the options enabled or chosen by the application code. For instance, the skeleton fragment shader defines tags as shown in Listing 2.

```
//VTK::Base::Dec

//VTK::Termination::Dec
```

Listing 2: Fragment shader tags

At runtime the `//VTK::Base::Dec` tag is replaced by shader code shown in Listing 3

```
// Volume dataset
uniform sampler3D in_volume;
uniform int in_noOfComponents;
uniform int in_independentComponents;

uniform sampler2D in_noiseSampler;
#ifndef GL_ES
uniform sampler2D in_depthSampler;
#endif

// Camera position
uniform vec3 in_cameraPos;

// view and model matrices
uniform mat4 in_volumeMatrix;
uniform mat4 in_inverseVolumeMatrix;
uniform mat4 in_projectionMatrix;
uniform mat4 in_inverseProjectionMatrix;
uniform mat4 in_modelViewMatrix;
uniform mat4 in_inverseModelViewMatrix;
uniform mat4 in_textureDatasetMatrix;
uniform mat4 in_inverseTextureDatasetMatrix;
varying mat4 ip_inverseTextureDataAdjusted;
uniform vec3 in_texMin;
uniform vec3 in_texMax;
uniform mat4 in_textureToEye;

// Ray step size
uniform vec3 in_cellStep;
uniform vec2 in_scalarsRange[4];
uniform vec3 in_cellSpacing;

// Sample distance
uniform float in_sampleDistance;

// Scales
uniform vec3 in_cellScale;
uniform vec2 in_windowLowerLeftCorner;
uniform vec2 in_inverseOriginalWindowSize;
uniform vec2 in_inverseWindowSize;
uniform vec3 in_textureExtentsMax;
uniform vec3 in_textureExtentsMin;

// Material and lighting
uniform vec3 in_diffuse[4];
uniform vec3 in_ambient[4];
uniform vec3 in_specular[4];
uniform float in_shininess[4];

// Others
uniform bool in_cellFlag;
uniform bool in_useJittering;
vec3 g_rayJitter = vec3(0.0);
uniform bool in_clampDepthToBackface;

uniform vec2 in_averageIPRange;
```

Listing 3: Base declaration fragment code

To define a structure, we have chosen a strategy that separates the tags in the following four categories:

1. **Declaration (::Dec)** The tags belonging to this group are meant to declare variables or function outside the main execution of the shader code. The variables defined are uniform, varying, and user-defined global variables. The functions defined are typically perform operations that are repetitive in nature such as computing color of a fragment.
2. **Initialization (::Init)** The tags belonging to this group are meant to initialize variables inside the main execution function of the shader but before the ray-casting loop in the fragment shader. An example of such code includes computation of the initial position of ray and direction of ray traversal.
3. **Implementation (::Impl)** The tags belonging to this group are the variables or functions or the combination of both that perform the actual operation of clipping, cropping, shading, etc. on one, two, or four component volume data. The implementation code used local and global variables and optimized for performance reasons as they are executed as long as the ray is traversing inside the volume and didn't run into a termination condition which is checked every time.
4. **Exit (::Exit)** The tags belonging to this group perform final computation such as the final color of the fragment. These tags are placed outside the ray-casting loop and typically contain numeric assignments.

2.3. Lighting / Shading

The old mapper supported only one light (due to limitations in OpenGL at the time the class was written). The `vtkFixedPointVolumeRayCastMapper` supports multiple lights, but only with an approximate lighting model, since gradients are precomputed and quantized, and shading is performed for each potential gradient direction regardless of fragment location. The new mapper accurately implemented the VTK lighting model to produce high quality images for publication. To support this, depending on the light type (point, directional, and positional), the lighting parameters are sent to the shader which then performs the per pixel lighting calculations. The number of lights is limited to six mostly for performance reasons as the interactive frame rate goes down significantly with each light added to the scene. The Phong shading lighting model is used for volume rendering lighting. Phong lighting requires normals which have to be computed for each fragment. The normal calculation is done by first computing

the gradient and then scaling the gradient by the spacing between the cells. The gradient is computed by reading the scalar values form the neighboring cdells using the offset vector that stores the step size based on the bounds of the volume.

2.4. Volume Picking

Picking, in the context of volume rendering, is the action of resolving the set of voxels a user has clicked on with the pointer. This provides the user with means to interact with the objects in a 3D scene. VTK legacy volume mappers support picking through an instance external to the mapper itself called `vtkVolumePicker`. This class casts a ray into the volume and returns the point where the ray intersects an isosurface of a user specified opacity. This technique has certain limitations given that the picking class does not have enough information to correctly account for clipping, transfer functions and other parameters defining how the mapper actually renders, thus reducing its reliability on the actual objects being picked.

VTK supports hardware-accelerated picking of goemetric data through the class `vtkHardwareSelector`, which uses a multiple-render-pass approach in order to resolve the various objects in a scene (actors in VTK parlance) and their corresponding primitives. Each pass renders separately a different selection abstraction (processes, actors, composite blocks and primitives), painting with a different color each of the various components in the scene. The images rendered by each pass are downloaded from the GPU and subsequently analyzed in the class thereby resolving the correspondence of each pixel to its particular actor, primitive, etc.

The inherent flexibility of the revamped shader implementation of this mapper permits a seamless integration with `vtkHardwareSelector`'s interface by rendering the appropriate colors for each of its passes, hence granting consistency in the object selection regardless of whether it is geometric or volumetric data. Providing selection support directly within the fragment shader ensures high selection accuracy even in situations where a volume intermixes with geometry in seemingly cumbersome ways or other advanced features (e.g. clipping) are enabled (what you see is what you pick). Given the readily available picking styles supported by `vtkHardwareSelector` (e.g. `vtkAreaPicker`), it is possible to make a selection of a specific set of visible voxels.

2.5. Volume Texture Streaming

An intrinsic limitation of volume rendering is that the 3D volume data to be rendered does not always fit into the graphics memory of a system. Such limitation becomes increasingly important to address now that the new mapper provides support for mobile architectures. A relatively simple method when dealing with a large volume is the volume streaming approach also commonly known as bricking [Engel et al., 2006] in which the volume is split into several blocks so that a single sub-block (brick) fits completely into GPU memory. Each sub-block is stored in main memory and streamed into GPU memory for a rendering pass one at a time (in a back-to-front manner for correct composition). The sub-blocks are rendered using the standard shader programs and alpha-blended with each other by OpenGL. Streaming the volume as separate texture bricks certainly, imposes a performance trade-off but acts as a graphics memory expansion scheme for devices that would not be able to render a higher quality volume otherwise.

2.6. Dual Depth-Peeling

VTK has long supported the use of depth-peeling for order-independent rendering of translucent geometry. Since translucent fragments must be blended in a specific order to obtain the correct pixel color, techniques like depth-peeling are necessary for correctly shading a complex scene.

In a multipass depth-peeling rendering, 'slices' of fragments are pulled from the scene in depth order; first the nearest fragments per pixel are collected in the first pass, and then the fragments just behind those are written in the next pass. After each pass, the current set of fragments is blended into an accumulation buffer, ultimately producing a correctly colored scene in which all fragments are blended from front-to-back.

The standard depth-peeling algorithm, which collects a single layer of fragments per geometry pass in front-to-back order, was recently updated to use a "dual depth-peeling" technique [Bavoil and Myers, 2008], in which two layers of fragments are peeled in a single geometry pass: One layer from the front and a second from the back. These are blended into two separate accumulation buffers and eventually the front and back peels will meet towards the middle of the geometry. This allows depth-peeling to be carried out in roughly half as many geometry passes.

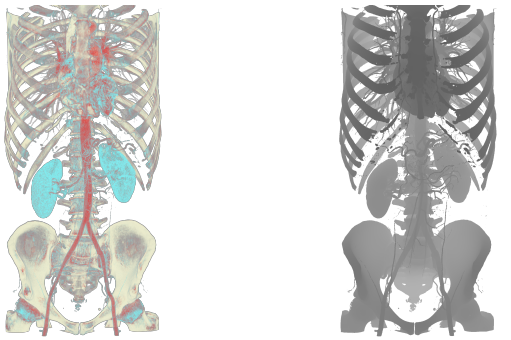
An interesting detail of the new depth-peeling implementation is that there are four depth values available per-pixel during a typical peeling pass. Two depth values each are associated with the front and back peels; these are the depth of the currently peeled fragment, and the depth of the next fragment that will be peeled. These depth values can be reinterpreted as two sets of "inner" and "outer" boundaries for a view-ray traversing a volume.

By adapting the volume mapper's fragment shader to use these depth values for computing two "slices" of a volume, we are able to integrate volume rendering into our depth-peeling pass. This enables volumetric data to be mixed with translucent geometry while efficiently yielding a correctly colored result.

2.7. Render to texture

Typically, the mapper uses a single pass rendering approach in which the 3D volume data is rendered on screen i.e. the default framebuffer that comprises of a color and a

depth buffer. In games and other graphics applications, a multi-pass rendering approach called "Render to Texture" is used to support advanced visual rendering techniques such as deferred shading, texture baking, post processing, etc. This technique uses the rendered output from one pass to modify/enhance the output of another rendering passes.



(a) Color image (b) Depth image

Figure 4: Render to texture

When the `RenderToImage` flag is enabled, the volume mapper switches rendering to an OpenGL FrameBuffer-Object (FBO) and allows the user to obtain the rendered pixel data via simple image retrieval calls. The mapper provides methods to grab color and depth information either as individual textures or as VTK's image data structure - `vtkImageData`. The color data is simply the pixel representation of the rendered volume whereas depth data consists of a grayscale image depicting how deep each voxel is in the scene. When an application enables render to texture mode, a framebuffer object is created, the vertex and fragment shader are generated, color and depth buffers are cleared, and the color is set to white with a value of zero alpha. The value of zero alpha is used so that applications can safely ignore the transparent pixels and the color buffer is set to white color because white represents the maximum depth, that is the depth of the far plane. In the render to texture pass, the fragment shader writes to color and depth target and to write the depth information it uses the depth of first non-transparent voxel.

2.8. Mobile Support

The decision to use OpenGL 3 or higher enabled volume rendering to support mobile devices (iOS and Android devices) as OpenGL ES 3.0 supports 3D textures. However the OpenGL ES does not support all of the texture format types, and therefore, new texture formats are added with compile time switch to enable or disable them depending on whether the platform is desktop or mobile. One such example is capturing of depth buffer since that is not yet supported on the mobile platform. Since typically interactions on mobile devices require touch interactions, the new rendering system added support for multiple touch events such as using two fingers to translate, rotate, and zoom the camera. With minor feature-set exceptions, the new volume mapper works on mobile devices enabling developers to build sophisticated applications for the scientific community.

2.9. Optimizations and Edge-Cases

The new `vtkOpenGLGPUVolumeRayCastMapper` is more than just a ray cast mapper implementation. It is designed to work on multiple platforms and developed to perform volume rendering at interactive frame rates. To achieve interactive frame rates and to handle edge cases, we have implemented following optimizations in the new mapper.

- *Clipping plane optimization* In VTK a user can place multiple planes at desired angles to clip the volume. This technique is essential for many medical use-cases. Since only one side of clip planes needs to be traversed, performing any sort of ray cast on the clipped side is wasteful. Hence a simple optimization is to move the starting point of the ray on the plane by projecting the ray onto the plane in the view direction.

- *Camera Inside the Bounding Box* Rendering with camera views within the volume is handled by clipping the proxy geometry with the camera near plane (using `vtkClipConvexPolyData`), thus ensuring that all bounding box fragments fall within range. Plane-Axis-Aligned-Bounding-Box intersection is used to determine whether geometry clipping is necessary.

- *Support for Double and Long Long Data Type* The current OpenGL API does not support 64-bit data types. In order to be able to render volumes of these types, the mapper loads the data array slice by slice casting the data values to floating-point. This, despite the obvious precision loss, is provided to the user as a convenience feature

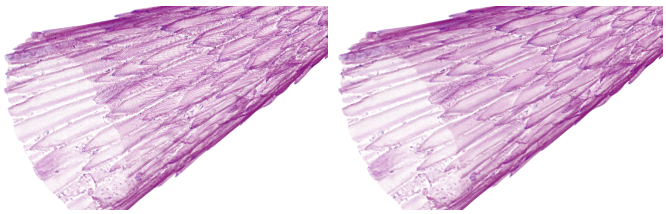
3. Results

By using the approach described in the previous section, the new mapper grew capable of providing critical features for medical and scientific computing. This section describes each of these features in sufficient detail in the following text.

3.1. Stochastic Jittering

Because ray-casting effectively samples a discrete signal (voxel values along the ray trajectory), the distance between those sampling points heavily influences how accurately the volume data is represented. Due to well established limitations described by the sampling theorem, low sampling rates may result in aliasing effects often referred to as wood-grain artifacts as shown in Figure 5a in the context of volume rendering [Engel et al., 2006]. Reducing the distance between samples neutralizes these artifacts but takes an important toll on performance.

The mapper supports stochastic jittering, which is an alternative technique to counteract wood-grain artifacts by adding a random offset to the rays in the viewing direction thereby breaking the coherence between neighboring fragments which causes the aliased patterns become apparent as shown in Figure 5b. Jittering is implemented by creating a random noise texture (using `vtkPerlinNoise`) and applying the offset to the ray's starting point, thus has a much lower performance penalty than reducing the sampling distance.



(a) Wood-grain artifacts. (b) With jittering enabled.

Figure 5: Stochastic jittering.

3.2. Clipping

A set of infinite planes can be defined to clip the volume to reveal inner detail, as shown in Figure 4. The visibility of each sample along the ray is determined by computing the sample's distance to each plane and testing for it being in front or behind.

The current implementation iterates through each of the planes before entering the ray marching loop in order to early-discard rays which fall in any of the following criteria:

1. Entering the volume on the clipped side and exiting before ever intersecting the plane (the ray only traverses clipped space).
2. Intersecting overlapping geometry before the plane (z-buffer compositing).

3.3. Cropping

Cropping refers to 27 regions that defined by two planes along each coordinate axis of the volume and can be independently turned on (visible) or off (invisible) to produce a variety of different cropping effects, as shown in Figure 6. Cropping is implemented by determining the cropping region of each sample location along the ray and including only those samples that fall within a visible region.

3.4. Wide Support of Data Types

The mapper supports most signed and unsigned data types such as short, int, float and double as well as the two most common data abstractions in VTK, cell and point data. Bias and scale factors are pre-computed and passed into the fragment shader to normalize the data values for correct look-up table mapping.

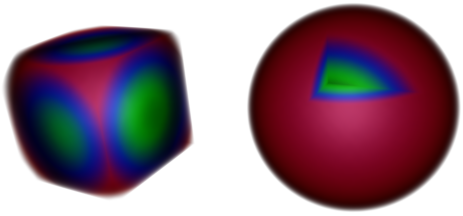
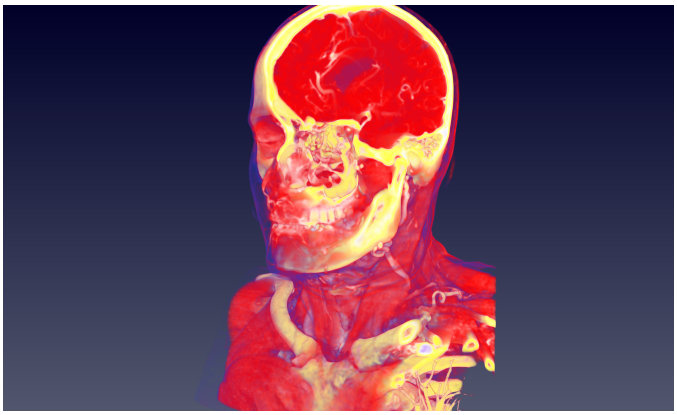


Figure 6: A sphere is cropped using two different configurations of cropping regions.



(a) Clipping using an oblique clipping plane

(b) Without shading (c) With shading

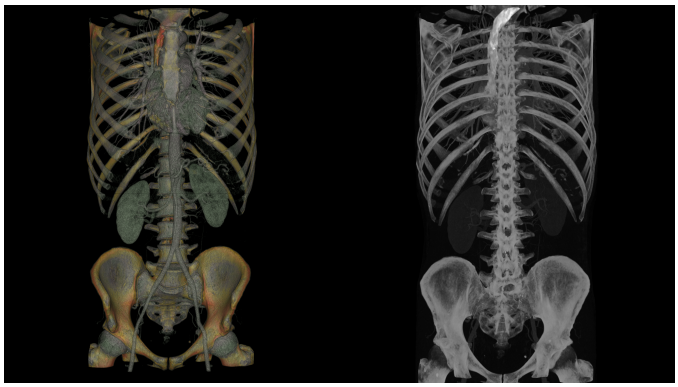
Figure 7: Clipping planes with `vtkGPUVolumeRayCastMapper`

3.5. Blending Modes

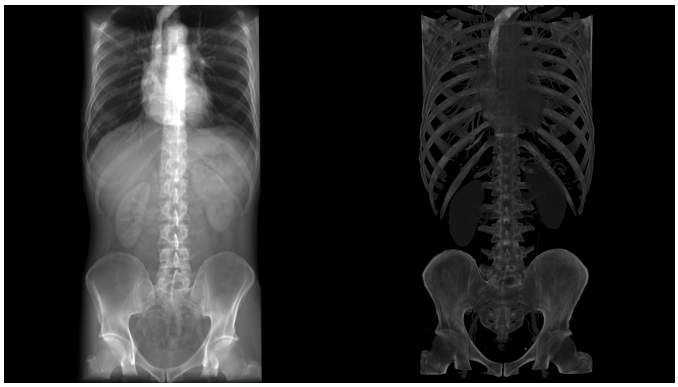
The mapper supports composite blending, minimum intensity projection, maximum intensity projection, additive intensity and average intensity blending. Each of these blending modes are useful for a particular use-case in medical computing. The most common one which is also the default is the composite blending mode. See Figure 8 for an example of the different blend modes on the same data.

3.6. Masking

Both binary and label masks are supported. With binary masks, the value in the masking volume indicates visibility of the voxel in the data volume. When a label map is in use, the value in the label map is used to select different rendering parameters for that sample. See Figure 5 for an example of label data masks.



(a) Composite (b) Maximum Intensity



(c) Additive (d) Average Intensity

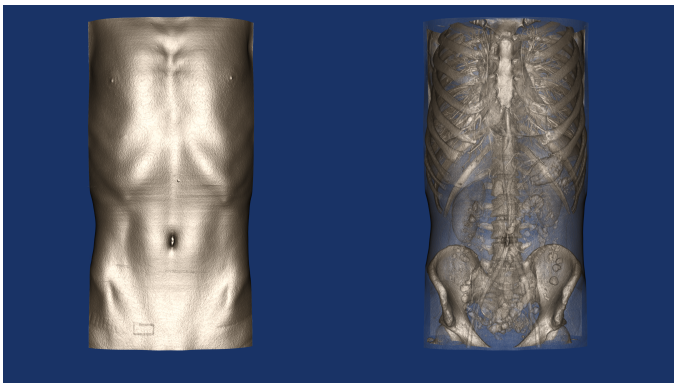
Figure 8: Blend modes supported by `vtkGPUVolumeRayCastMapper`

3.7. Opacity Modulated by Gradient Magnitude

While the `vtkGPUVolumeRayCastMapper` supports direct accumulation of color and opacity values along the ray, it also allows for modulating the opacity accumulation calculation based on relative values of each voxel in the dataset. Prior efforts [Marchesin et al., 2010] have demonstrated the usefulness of such a technique for feature enhancement in the rendered image. A transfer function mapping the magnitude of the gradient to an opacity modulation value can be used to essentially perform edge detection (de-emphasize homogenous regions) during rendering. See Figure 9 for an example of rendering with and without the use of a gradient opacity transfer function.

3.8. Performance benchmarks

As part of the modernization effort, we obtained several performance metrics from sample systems with differing specifications of hardware graphics; ranging from onboard graphic cards to dedicated GPUs. The code used to benchmark is located within the VTK code repository under `Utilities/Benchmarks/`. The test runs for a specified number of iterations wherein each iteration increases data size by an order. For each iteration, the test creates a mock dataset using the `vtkRTAnalyticSource` and renders it for 80 frames, rotating the dataset each frame and recording the time taken to render to screen. Each test was run twice on each system; once with and once without gradient computations required by shading. The results of the benchmark are shown in Figure 14. As observed, surface shaded rendering degrades performance but the mapper maintains interactive frame rates even for large datasets.



(a) Without gradient opacity function (b) With gradient opacity function

Figure 9: Gradient magnitude based opacity modulation

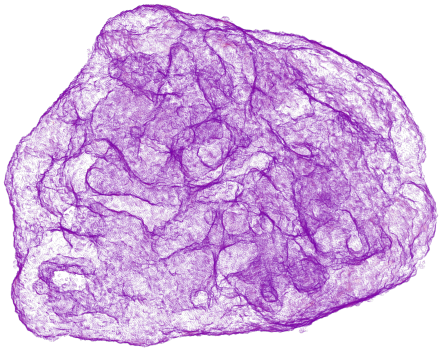


Figure 10: Platinum-Copper (PtCu) nanoparticle [Scott et al., 2012; Miao et al., 2016] with gradient opacity modulation enabled

4. Future Work

At this point, we have a replacement class for vtkGPURayCastMapper that is more widely supported, faster, more easily extensible, and supports majority of the features of the old class. In the near future, our goal is to ensure that this mapper works as promised by integrating it into existing VTK applications such as ParaView and Slicer. Once these tasks are complete, we have some ideas on new features we would like to add to this mapper (outlined below). We would also like to solicit feedback from folks using the VTK volume mappers. What features do you need? Drop us a line at kitware@kitware.com, and let us know.

4.1. 2D Transfer Functions

Currently, volume rendering in VTK uses two 1D transfer functions, mapping scalar value to opacity and gradient

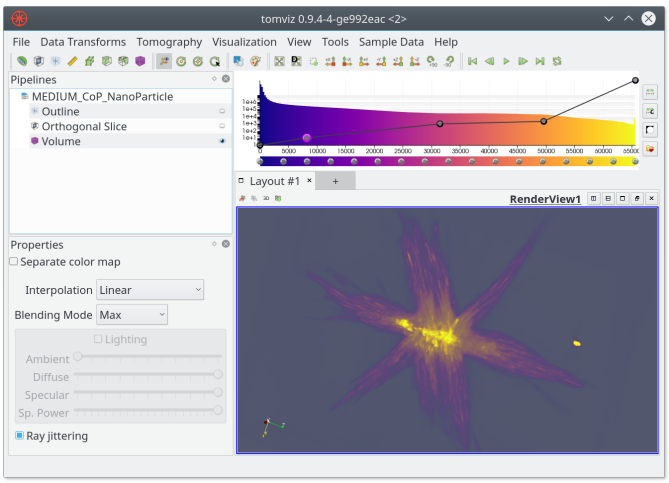


Figure 11: Tomviz [Hanwell et al., 2014] rendering a Cobalt-Phosphorous (Co_2P) nanoparticle [Levin et al., 2016]

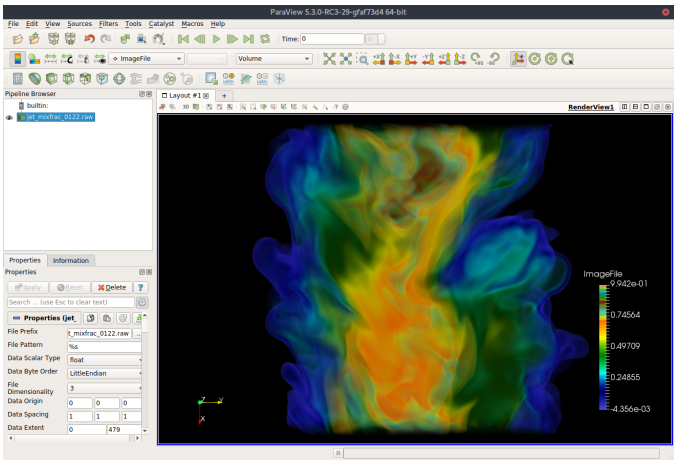


Figure 13: ParaView [Ahrens et al., 2005; Ayachit, 2015; Ayachit et al., 2015] rendering a single time step of a simulation of temporally-evolving plane jet flames [Hiroshi Akiba et al., 2007]

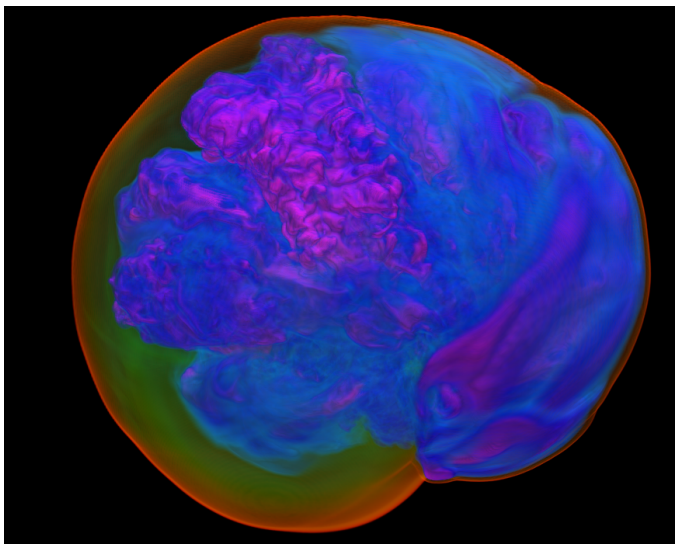


Figure 12: Rendering a single timestep output of the supernova modeling simulation [Blondin and Mezzacappa, 2007]

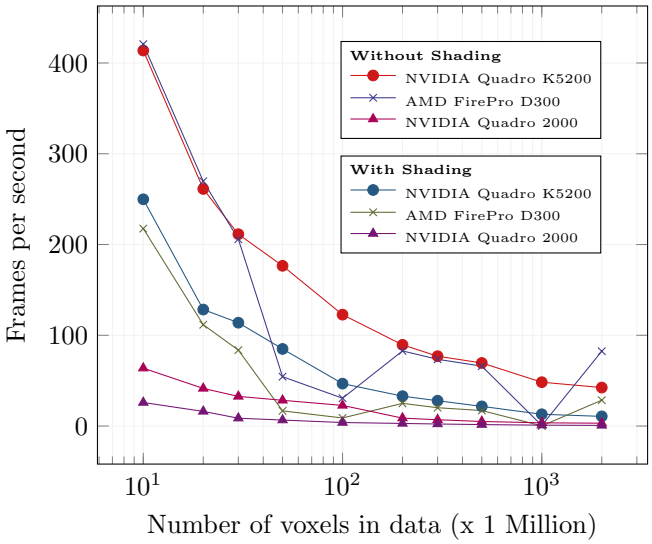


Figure 14: Results of the benchmarking tests performed on three different graphics cards using the `vtkGPUVolumeRayCastMapper` with and without shading (gradient computations)

magnitude to opacity. For some application areas, better rendering results can be obtained by using a 2D table that maps these two parameters into an opacity value. Part of the challenge in adding a new feature such as this to volume rendering in VTK is simply the number of volume mappers that have to be updated to handle it (either correctly rendering according to these new parameters or at least gracefully implementing an approximation). Once we have reduced the number of volume mappers in VTK, then adding new features such as this will become more manageable.

4.2. Support for Depth Peeling

VTK allows intermixing of volumes with opaque geometry. For translucent geometry, you can obtain a correct image only if all translucent props can be sorted in depth order. Therefore, no translucent geometry can be inside a volume, as it would be, for example, when depicting a cut plane location with a 3D widget that represents the cutting plane as a translucent polygon. Nor can a volume be contained within a translucent geometric object, as it would be if, for example, the outer skin of a CT data set was rendered as a polygonal isosurface with volume mappers used to render individual organs contained within the skin surface. We hope to extend the new vtkGPURayCastMapper to support the multipass depth peeling process, allowing for correctly rendered images with intersecting translucent objects.

4.3. Overlapping Volumes

It is currently possible to render overlapping volumes by taking advantage of the up to 4 independent components supported by the mapper (each component representing

a different volume). The limitation with this approach is that each of the overlapping volumes are required to be sampled in the same grid, hence all of the volumes are required to share the same dimensions. Nonetheless, in order to extend the mapper to support overlapping volumes sampled in grids with different dimensions, rays should be casted through proxy geometry bounding the N overlapping volumes to be rendered and separately sampling and compositing their fetched texture values in the fragment shader.

4.4. Improved Rendering of Labeled Data

Currently, VTK supports binary masks and only a couple of very specific versions of label mapping. We know that our community needs more extensive label mapping functionality especially for medical datasets. Labeled data requires careful attention to the interpolation method used for various parameters. (You may wish to use linear interpolation for the scalar value to look up opacity, but, perhaps, select the nearest label to look up the color.) We plan to solicit feedback from the VTK community to understand the sources of labeled data and the application requirements for visualization of this data. We then hope to implement more comprehensive labeled data volume rendering for both the CPU and GPU mappers.

5. Acknowledgements

We would like to recognize the National Institutes of Health for sponsoring this work under the grant NIH R01EB014955 - "Accelerating Community-Driven Medical Innovation with VTK."

We thank the maintainers of the OsiriX DICOM Image Library [OsiriX, 2017] for providing the head (used in Figure 7) and torso (used in Figures 4, 8 and 9) datasets used in this publication. The Supernova (used for Figure 12) and Turbulent-Combustion (used for Figure 13) datasets were obtained from VisFiles [VisFiles, 2007]. The supernova dataset is made available by John Blondin at the North Carolina State University through US Department of Energy's SciDAC Institute for Ultrascale Visualization.The turbulent combustion dataset is made available by Jackqueline Chen at Sandia Laboratories through US Department of Energy's SciDAC Institute for Ultrascale Visualization. We would also like to thank Jianwei (John) Miao from University of California at Los Angeles and Robert Hovden from Cornell University for granting us permission to use the PtCu (used in Figure 10) and Co_2P (used in Figure 11) nanoparticle datasets.

6. References

Ahrens, J., Geveci, B., and Law, C. (2005). *ParaView: An End-User Tool for Large Data Visualization*. Visualization Handbook, Elsevier.

Ayachit, U. (2015). *The ParaView Guide: A Parallel Visualization Application*. Kitware, Inc.

Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., and Mauldin, J. (2015). ParaView Catalyst: Enabling In Situ Data Analysis and Visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, pages 25–29, New York, NY, USA. ACM.

Bavoil, L. and Myers, K. (2008). Order Independent Transparency with Dual Depth Peeling. Technical report, NVIDIA.

Blondin, J. M. and Mezzacappa, A. (2007). Pulsar spins from an instability in the accretion shock of supernovae. *Nature*, 445(7123):58–60.

Engel, K., Hadwiger, M., Kniss, J. M., Rezk-Salama, C., and Weiskopf, D. (2006). *Real-Time Volume Graphics*. A. K. Peters, Ltd., Wellesley, MA, USA, 1st edition.

Geveci, B. and Schroeder, W. (2012). VTK. In Brown, A. and Wilson, G., editors, *The Architecture of Open Source Applications - Elegance, Evolution and a Few Fearless Hacks*, volume 1, pages 385 – 400. lulu.com, 1st edition.

Hanwell, M. D., Ayachit, U., Muller, D., and Hovden, R. (2014). Tomviz for tomographic visualization of nanoscale materials. http://www.tomviz.org/.

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

Hiroshi Akiba, Evatt R. Hawkes, Kwan-Liu Ma, and Jacqueline H. Chen (March/April 2007). Visualizing multivariate volume data from turbulent combustion simulations. *Computing in Science & Engineering*, 9:76–83.

Levin, B. D. A., Padgett, E., Chen, C.-C., Scott, M. C., Xu, R., Theis, W., Jiang, Y., Yang, Y., Ophus, C., Zhang, H., Ha, D.-H., Wang, D., Yu, Y., Abruña, H. D., Robinson, R. D., Ercius, P., Kourkoutis, L. F., Miao, J., Muller, D. A., and Hovden, R. (2016). Nanomaterial datasets to advance tomography in scanning transmission electron microscopy. *Scientific Data*, 3:160041.

Marchesin, S., Dischler, J. M., and Mongenet, C. (2010). Per-Pixel Opacity Modulation for Feature Enhancement in Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 16(4):560–570.

Miao, J., Ercius, P., and Billinge, S. J. L. (2016). Atomic electron tomography: 3D structures without crystals. *Science*, 353(6306):aaf2157.

OsiriX (2017). OsiriX — DICOM Image Library. http://www.osirix-viewer.com/resources/dicom-image-library/.

Schroeder, W., Martin, K., and Bill, L. (2006). *The Visualization Toolkit - An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., 4th edition.

Scott, M. C., Chen, C.-C., Mecklenburg, M., Zhu, C., Xu, R., Ercius, P., Dahmen, U., Regan, B. C., and Miao, J. (2012). Electron tomography at 2.4-angstrom resolution. *Nature*, 483(7390):444–447.

Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, Upper Saddle River, NJ, 8th edition.

VisFiles (2007). VisFiles. http://vis.cs.ucdavis.edu/VisFiles/index.php.