Cross-platform ubiquitous volume rendering using programmable shaders in VTK for scientific and medical visualization

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

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 supports a wide variety of data formats, algorithms, and rendering techniques for both polygonal and volumetric data. In particular VTK's volume rendering module has long provided 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, the system did not take advantage of the latest improvements in graphics hardware or the flexibility of a programmable pipeline. Additionally, this dependence on an antiquated pipeline posed restrictions when running on emerging computing platforms, thereby limited its overall applicability. In response to these limitations, the VTK community developed a new and improved volume rendering module, which not only produced a modern GPU-based implementation, but also augmented its capabilities with new 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 supplement a textbook on object-oriented computer graphics programming [Schroeder et al., 2006; Geveci and Schroeder, 2012]. VTK has a long history of volume rendering and, unfortunately, that history was evident in its large collection of classes used to render volumes. While these methods were state-of-the-art at the time they were implemented, given VTK's 20+ year history, many of these methods are now obsolete. Recently, there has been a major effort [Hanwell et al., 2015] undertaken to re-write VTK's rendering backend, which was based on a now-deprecated legacy OpenGL API, to one based on a modern, programmable-pipeline OpenGL API [Shreiner et al., 2013].

This new rendering subsystem was designed to support the latest technological advances in the graphics hardware industry. We aimed to consolidate the large number of volume mappers into two: one supporting accelerated rendering using the Graphics Processing Unit (GPU) and a second, parallel implementation on the Central Processing Unit (CPU). In addition, a `vtkSmartVolumeMapper` class was added to assist application developers in providing an automatic run-time selection of the appropriate volume rendering techniques based on system configuration (see Figure 1b).

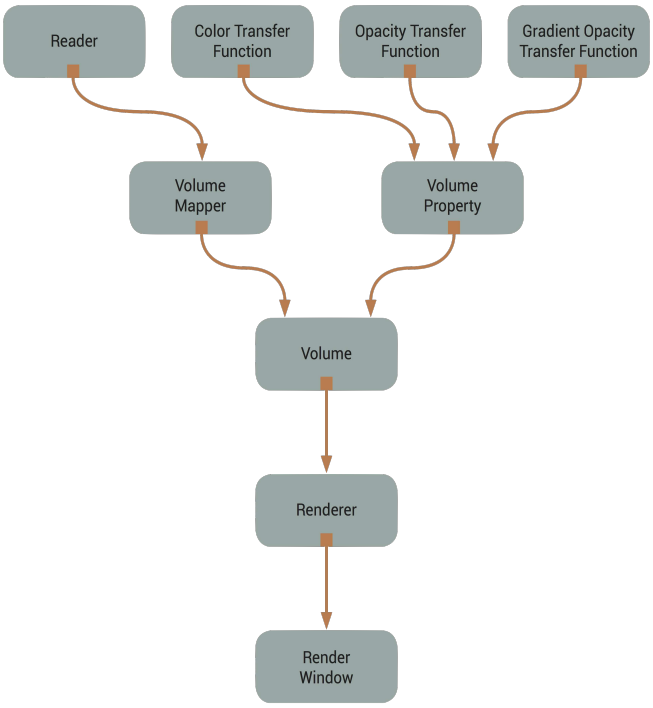
Thus, a primary objective of this OpenGL modernization effort was the subject of this article: to create a cross-platform, multi-functional, high-performance volume renderer supporting both serial and parallel execution modes capable of supporting such complex applications as ParaView [Ahrens et al., 2005; Ayachit, 2015] or 3D Slicer [Fedorov et al., 2012]. While volume rendering using ray-casting approach is currently the state-of-the-art, the strength of our work is the conglomeration of many principles of the chosen technique, which has resulted in a fast-performing, pipeline-based volume render that works for multiple types and formats of the scientific and medical datasets.

The goal of our effort is as follows:

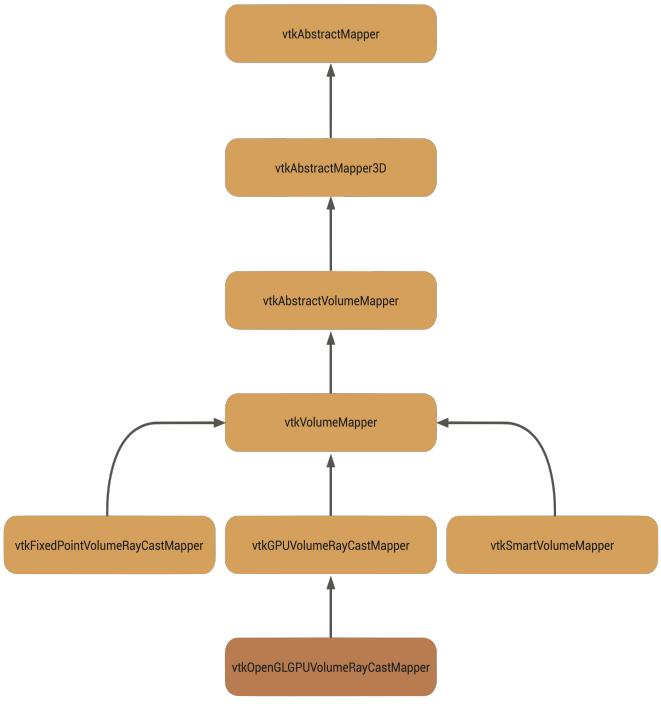
- Enable ubiquitous, cross-platform, high-performance support for volume rendering across all major operating systems and computing environments (desktop, VR, mobile).
- Ensure that the new volume visualization subsystem

*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), `will.schroeder@kitware.com` (Will Schroeder)



(a) VTK pipeline for volume rendering which is similar to the polygonal rendering pipeline in VTK



(b) Graph depicting C++ class hierarchy for the various volume mappers in VTK

Figure 1: Volume rendering works within the constructs of the VTK pipeline mechanism

provides support for VTK-based pipeline and data-flow networks, thereby providing a flexible system addressing a wide variety of scientific and medical data visualization use-cases.

- Provide a variety of useful features at interactive frame rates such as clipping, cropping, gradient opacity, mixed geometry-volume translucent rendering, and on-demand shader composition.

We created a replacement for the OpenGL fixed-pipeline-based `vtkGPUVolumeRayCastMapper`. The new mapper, which shares the same name but uses a modern OpenGL programmable pipeline, can be used via `vtkSmartVolumeMapper` or instantiated directly. The new mapper with its OpenGL-VTK implementation improves the management of textures in the mapper, and benefits both geometry and volume rendering 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 across all major computing platforms, and performs well at interactive frame rates with very large datasets was a challenging task. Our team met this challenge with an in-depth knowledge of the data, graphics pipeline, the VTK framework, and user requirements.

In the next section, we describe the technical details behind the effort to produce the resulting modern, cross-platform volume renderer delivered to the open source VTK community.

2. Background

Volume visualization provides a two-dimensional view of three-dimensional uniformly sampled data by sampling and projecting the data onto a 2D projection plane. While many techniques exist for volume rendering, e.g. marching cubes [Lorensen and Cline, 1987], image splatting [Westover, 1990], texture slicing [Rezk-Salama et al., 2000; Engel et al., 2001], and ray casting [Hsu, 1993; Ma, 1995; Ma and Crockett, 1997; Heng and Gu, 2005], ray casting has become pervasive on modern graphics hardware. Ray casting techniques yield higher quality rendering, and provide a variety of options to optimize the rendering performance, such as early ray termination and space leaping [Yagel and Shi, 1993].

While ray casting is a well-known technique with many implementations, challenges still exist in providing high-quality volume visualization for datasets of adequate resolution, supporting a variety of spacing and formats, with or without intermixed geometry rendering, on multiple platforms (desktop, Cave, Head Mounted Displays (HMD)) and across operating systems (Apple, Linux, Window), at interactive speeds.

A few open source software systems exist for volume visualization, such as Voreen [Meyer-Spradow et al., 2009] which began at the Department of Computer Science at the University of Münster, Germany, in 2004. It provides features such as isosurface rendering, maximum intensity projection (MIP) [Wallis et al., 1989], support for 1D and 2D transfer functions, and support for different illumination models (Phong, tone, ambient occlusion). Both Voreen and VTK use data-flow networks. However, Voreen is a volume visualization library whereas VTK is a scientific visualization library which provides better support for geometry and gridded datasets. While Voreen provides a rapid prototyping environment, VTK volume visualization aims for production quality, performance, and ubiquitousness. Additionally, Voreen only supports high-end desktop devices whereas VTK supports multiple platforms natively, bridging the gap between academic research and practical application. Other volume visualization capable tools such as ParaView, Slicer, and MayaVi [Ramachandran and Varoquaux, 2011] all use VTK as the underlying visualization library, and will benefit from our work. ParaView has already been switched to use our work, and work on updating Slicer to use our mapper is in progress.

Another volume rendering-capable tool, Amira [Stalling et al., 2005], provides multiple modules for volume visualization namely: voltex, volren, and volume rendering module. In the context of our work, the volume rendering module of Amira is most relevant. It provides support for GPU rendering, iso-surface, MIP, and multi-volume support. In comparison to our work, it provides limited support for data larger than main memory as it needs conversion to LDA format, lack of other volume rendering modes such as MinIP, off-screen rendering, and gradient-opacity and is closed-source. OpenWalnut [Eichelbaum et al., 2010] is another open-source tools that for visualization of medical and bio-signal data but provides very basic volume visualization support, but does offer off-screen rendering support. Another example of an open source volume rendering software system is ImageVis3D [CIBC, 2016], developed by the researchers at the University of Utah. ImageVis3D is an application as opposed to a library and provides support for large volume data, desktop as well as mobile devices, MIP, 1D and 2D transfer functions amongst many others. It does not support Virtual Reality environments and the flexibility of data-flow networks. Other data type specific volume rendering libraries such as Voxx [Clendenon et al., 2002] and ClearVolume [Royer et al., 2015] provide visualization of biological and light-sheet microscopy. PyMOL [Schrödinger, LLC, 2015] provides volume visualization capabilities for molecular datasets only. Finally, hardware architecture specific volume visualization libraries such as OSPRay [Wald et al., 2017] and NVIDIA® Optix™ [Parker et al., 2010] provide fast large volume data visualization capabilities on Intel CPU and Nvidia GPU's but lack support for non-native hardware.

Many volume rendering APIs exist apart from Voreen and ImageVis3D. However, except for Voreen, advanced volume visualization research performed by the industry and academic communities are not publicly available. By providing an open source volume rendering engine that supports multiple platforms natively, we hope to bridge the gap between academic research and industrial applications in the field of volume visualization.

3. Implementation Details

The new `vtkGPUVolumeRayCastMapper` uses a ray casting technique [Engel et al., 2006] for volume rendering which is state-of-the-art on modern graphics platforms. At a high level, it is similar to the previous version of this class, with a different OpenGL implementation reflecting recent advances in graphics systems. One of the main reasons we chose to use ray casting is due to the flexibility of this technique, which supports the many features of the previous 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 instances of vtkProp in a scene using one or more rendering 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 rendering opaque geometry, 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 adds new capabilities such as GPU-based clipping, gradient opacity, and volume picking amongst many others. In the next few sections, we will cover each of these features in detail.

3.1. Single Pass

In a ray-casting algorithm, the entry and the exit point into the volume is needed to determine when to stop ray stepping process. 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 back face is rendered. Using the interpolated vertex position and texture lookup, the start and end positions can be computed. Instead of this, in `vtkGPUVolumeRayCastMapper`, entry and exit points are computed based on the fact that the texture extents of the volume are within vec3(1.0), vec3(-1.0) range.The code below shows the fragment shader pseudo code that determines whether or not to stop ray stepping through the volume.

```
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 required to determine the termination of the ray.

3.2. Dynamic Shader Generation

All operations are performed on the GPU in the new `vtkOpenGLGPUVolumeRayCastMapper`. The advantage of this approach is more streamlined code that is easier to maintain and debug. This approach also provides 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 this improved mapper, 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 `vtkVolumeShaderComposer` depending on the options enabled or chosen by the application code. These tags correspond to each setup in a ray casting shader: 1) Declaration of variables 2) Initialization of ray position and direction 3) Iterative Ray traversal inside the volume and check for ray termination conditions 4) Final color and opacity computation after ray termination. For instance, the skeleton fragment shader defines tags as shown in Listing 2.

```
int main()
{
  // The tag below will be replaced by a
  // a concatenated string at run-time that
  // declares the appropriate core
  // shader variables which will be used regardless
  // of the operation (MIP, MinIP) performed.
  // The replacement is shown in ~\Autoref{lst:
      basedeclshader}.
  //VTK::Base::Dec

  // The tag below will be replaced to declare
  // variables that are responsible for termination
  // of the ray-casting. Essentially this enables
  // users to partially replace the shader and
      customize
  // it as necessary.
  //VTK::Termination::Dec

  // Advance ray and accumulate color and opacity
  while(not exit)
  {
    // This tag will be replaced by a concatenated
    // string that performs color and opacity
        aggregation
    // along the ray.
    //VTK::Base::Impl

    // Other tags implementing Cropping, Clipping
        etc.

    // Advance ray here

    // This tag will be replaced by a concatenated
    // string that will check if the ray is still
    // inside the volume and other checks.
    //VTK::Terminate::Impl
  }

  // This tag will be replaced by a concatenated
  // string that performs final color, depth, and
      opacity
  // computation.
  //VTK::Shading::Exit
}
```

Listing 2: Fragment shader tags

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 ray initial position and direction of 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 uses local and global variables and is optimized for performance reasons as it is 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.

3.3. Lighting / Shading

The old mapper supported only one light (due to limitations in OpenGL at the time the class was written). Similarly 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. However the new mapper accurately implements 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 must 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 cells using the offset vector that stores the step size based on the bounds of the volume.

3.4. 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 the means to interact with 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 renders, thus reducing its reliability on the objects being picked.

Now VTK supports hardware-accelerated picking of geometric data through the class `vtkHardwareSelector`, which uses a multiple-render-pass approach in order to resolve the various objects in a scene (actors or props 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.

3.5. Volume Texture Streaming

A common limitation of volume rendering is that the 3D volume data does not always fit into the graphics memory of a system. This limitation has become increasingly important to address as 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 are not able to render a large volume otherwise.

3.6. 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.

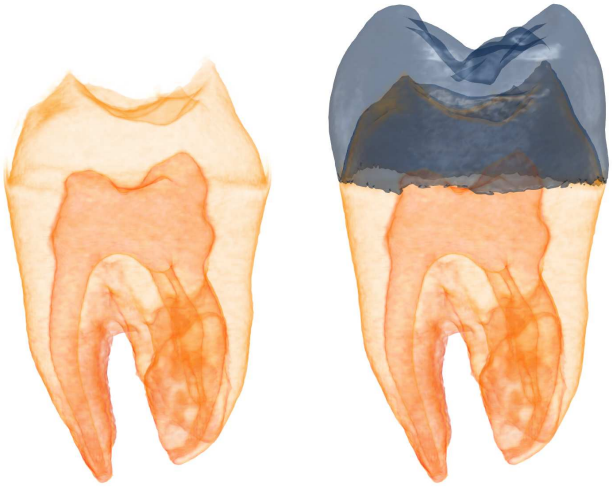
In a multipass depth-peeling rendering, 'slices' of fragments are pulled from the scene in depth order; the nearest fragments per pixel are collected in the first pass, and then the fragments just behind those are written in the next pass. After each pass, the current set of fragments is blended into an accumulation buffer, ultimately producing a correctly colored scene in which all fragments are blended 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 meet towards the middle of the geometry. This allows depth-peeling to be carried out in roughly half as many geometry passes. For this purpose, accumulation buffers have been implemented as RGBA8 texture attachments of a framebuffer to which the rendering results are output on each intermediate peeling/blending pass.

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 Figure 2.



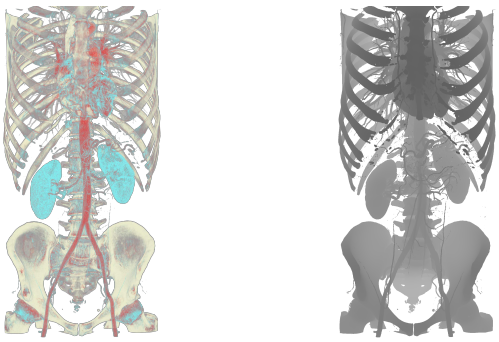
(a) Volume only (b) Intermixed

(c) Volume only (rotated) (d) Intermixed (rotated)

Figure 2: Industrial CT scan of a human tooth [Pfister et al., 2001]. The dentin and pulp of the tooth are rendered as a volume (a) while an iso-contour of the enamel is rendered as translucent polygonal geometry (b). (c) and (d) show a different viewpoint (object rotated on the horizontal axis) from which it is easier to observe the correct ordering of the rendered entities.

3.7. 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 other rendering passes.



(a) Color image (b) Depth image

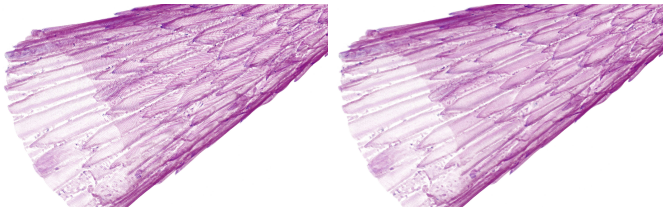
Figure 3: When `RenderToImage` is enabled, the color and depth data of the final rendering is written to a texture object which can be used for further processing

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 (as depicted in Figure 3). 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: the depth of the far plane. In the render-to-texture pass, the fragment shader writes to color and depth target. To write the depth information it uses the depth of the first non-transparent voxel.

3.8. Stochastic Jittering

Because ray-casting effectively samples a discrete signal (voxel values along the ray trajectory), the distance between 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 4a 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 less apparent as shown in Figure 4b. Jittering is implemented by creating a random noise texture (using `vtkPerlinNoise`) and applying the offset to the ray's starting point, resulting in a much lower performance penalty than reducing the sampling distance.



(a) Wood-grain artifacts. (b) Jittering enabled.

Figure 4: Cactus sample scanned at Beamline 8.3.2, Advanced Light Source, Lawrence Berkeley National Laboratory. A coarse ray-sampling distance causes wood-grain artifacts (a) to appear. Same data and sampling distance with stochastic jittering enabled (b) to mitigate the artifacts.

3.9. 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 meet any of the following criteria:

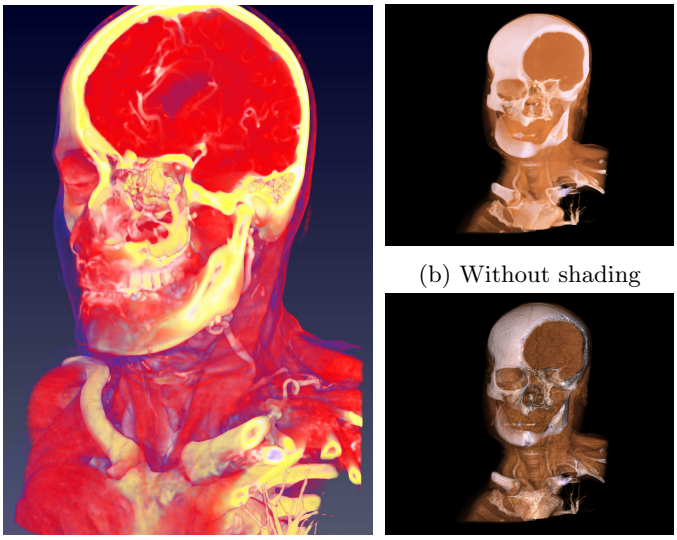
1. Entering the volume on the clipped side and exiting before ever intersecting the plane (the ray only traverses clipped space).
2. Intersecting overlapping geometry before the plane (z-buffer compositing).

3.10. Cropping

Cropping refers to the 27 regions defined by pairs of planes along each volume coordinate axis and can be independently turned on (visible) or off (invisible) to produce a variety of different cropping effectsCropping 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.

3.11. Broad 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.



(a) Oblique clipping plane (b) Without shading (c) With shading

Figure 5: Clipping planes with `vtkGPUVolumeRayCastMapper`. Example of an oblique (a) i.e. off-axis clipping plane through the volume. A clipping plane through the volume without (b) and with (c) surface shading.

3.12. Blending Modes

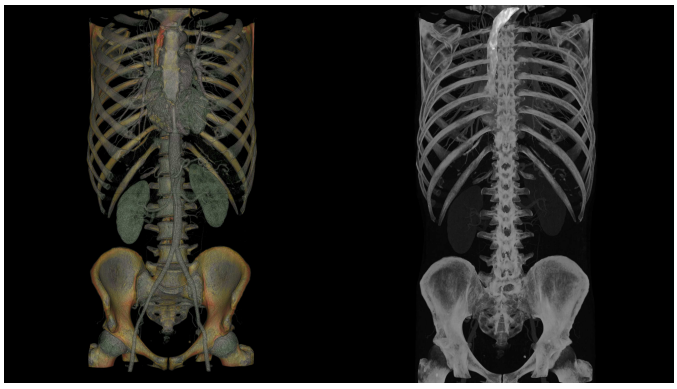
The mapper supports composite blending, minimum intensity projection, maximum intensity projection, additive intensity and average intensity blending. These blending modes are useful for the variety of use case found in medical computing. The most common one, which is also the default, is the composite blending mode. See Figure 6 for an example of the different blend modes on the same data.

3.13. Masking

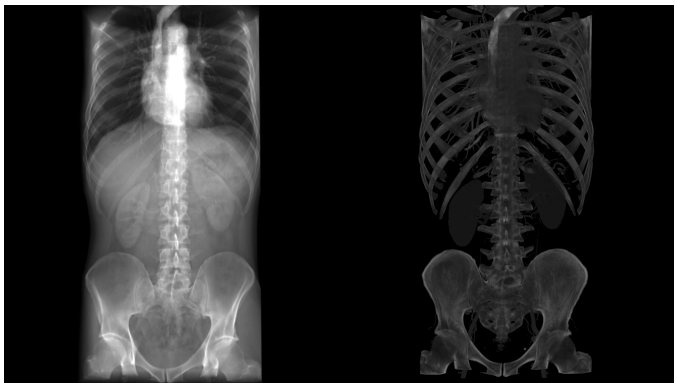
Both binary and label masks are supported. With binary masks, the value in the masking volume indicates visibility of the voxel in the data volume. When a label map is in use, the value in the label map is used to select different rendering parameters for that sample. See Figure 3for an example of label data masks.

3.14. 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 7 for an example of rendering with and without the use of a gradient opacity transfer function.



(a) Composite (b) Maximum Intensity



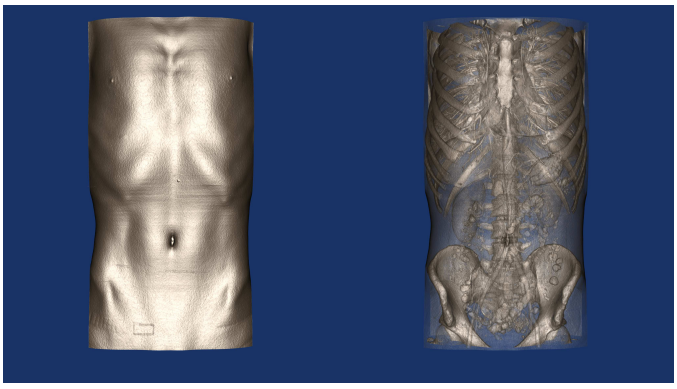
(c) Additive (d) Average Intensity

Figure 6: Blend modes supported by `vtkGPUVolumeRayCastMapper`

3.15. 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. The 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.



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

Figure 7: Gradient magnitude based opacity modulation

4. Performance benchmarks

As part of the modernization effort, we obtained several performance metrics from sample systems with differing specifications of hardware graphics ranging from on-board graphic cards to dedicated GPUs. The code used for benchmarking is located within the VTK code repository under `Utilities/Benchmarks/`. The test runs for a specified number of iterations with each iteration increasing data size. 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 benchmarking are shown in Figure 8. As observed, surface shaded rendering degrades performance but the mapper maintains interactive frame rates even for large datasets. Note that the tests on some systems were restricted to a maximum voxel count of 300 million voxels owing to graphics memory limitations. Note also that we were able to render large volumes requiring more than the available graphics memory using texture streaming, as described in subsection 3.5, outside the scope of this performance benchmark evaluation.

5. Application Areas

One of the goals of our work is to support volume visualization for various domains (scientific, medical), on multiple devices (workstations, virtual reality, mobile, clusters) and on different operating systems (Linux, Mac, Windows). This is important as VTK is used by a large user base in different setups. In the next few sub-sections, we

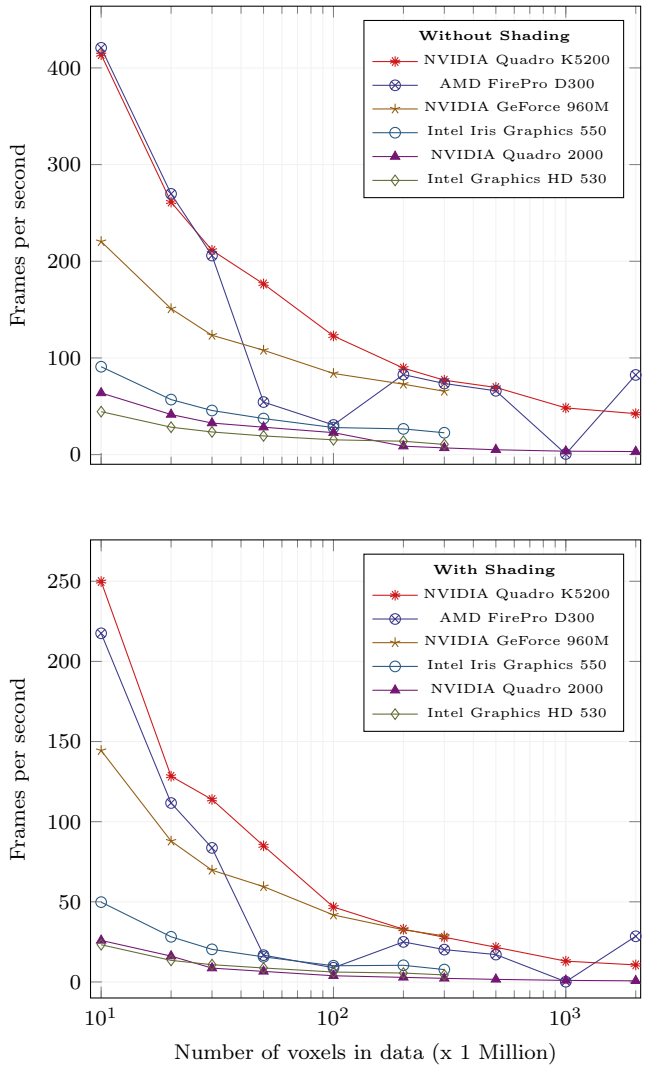
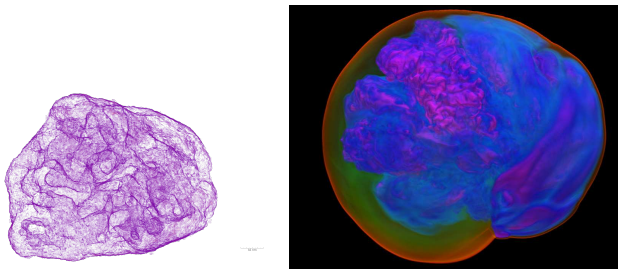


Figure 8: Results of the benchmarking tests performed on six different graphics cards using the `vtkGPUVolumeRayCastMapper` without and with surface shading (i.e. gradient computations)

have presented possible customizations to adapt our mapper for different domains and environments.

5.1. Domains

Owing to the fact that the new mapper is built with the VTK pipeline model (see Figure 1a), it can be integrated into existing VTK applications with little effort. As described in section 3, the volume mapper provides a rich feature-set for rendering a variety of data types. This versatility allows it to be used in diverse scientific and medical domains for volumetric visualization. As shown in Figure 10, it has found its way into real-world scientific applications like ParaView [Ahrens et al., 2005; Ayachit, 2015; Ayachit et al., 2015] and tomviz [Hanwell et al., 2014]. The volume mapper can be used for visualizing data generated from physics simulations (Figure 9b), transmission and scanning electron microscopes (Figure 9a) and medical and dental CT scans (Figures 2, 5 and 6).



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

(b) Rendering a single timestep output of the supernova modeling simulation [Blondin and Mezzacappa, 2007]

Figure 9: Physics and chemistry data visualization using the `vtkGPUVolumeRayCastMapper`

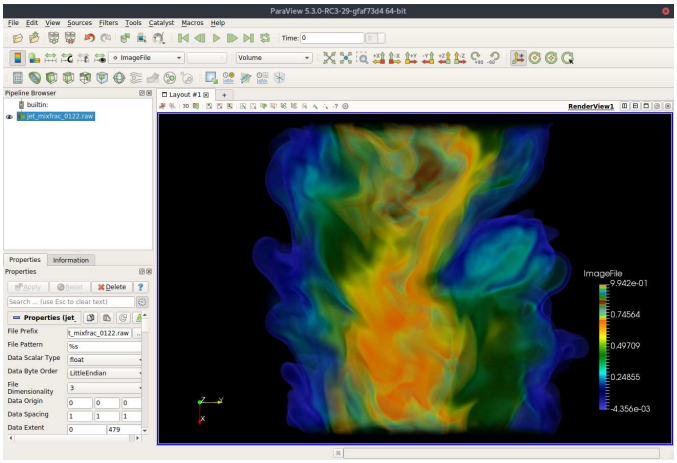


Figure 10: ParaView [Ahrens et al., 2005; Ayachit, 2015; Ayachit et al., 2015] rendering a single time step of a simulation of temporally-evolving plane jet flames [Hiroshi Akiba et al., 2007]

5.2. Devices and Environments

The new volume mapper supports rendering on most common devices including mobile platforms. This poses a challenge as the mapper would have to be versatile enough to handle low computing power requirements of mobile devices and scene composition in a multi-processor cluster environment simultaneously and still perform at *interactive* frame rates.

5.2.1. Mobile Support

The decision to use OpenGL 3 or higher enabled volume rendering to support mobile devices (iOS and Android devices) since OpenGL ES 3.0 supports 3D textures. However OpenGL ES does not support all of the texture format types, therefore, new texture formats are added with compile time switches 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. Furthermore, as interactions on mobile devices require touch interactions, the new rendering system added support for multi-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.

Various artifacts arise when rendering parallely (bricking) on a cluster. Artifacts due to sampling, gradient computation, etc. are seen at the edges of each of the bricks. To address it, our mapper ensures the entry texture coordinate and the limit texture coordinates are correctly adjusted to, and the ray step is scaled accordingly.

Finally, to perform consistently across various devices, our mapper uses two critical pieces of information from VTK. First, the last frame-render time, and second, the desired frame rate for the application. Using these two factors, the mapper distinguishes between an *interactive* versus *still* render and accordingly makes adjustments to the ray sampling computations to achieve the desired frame rate.

6. Future Work

As a result of this work, we have developed a replacement class for `vtkGPUVolumeRayCastMapper` 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 the mapper works as promised with existing VTK applications. Eventually, we plan on adding additional, advanced features as described in the following sub-sections.

6.1. 2D Transfer Functions

Currently, volume rendering in VTK uses three independent 1D transfer functions to map scalar value to color, scalar value to opacity and gradient magnitude to opacity. Increasing the number of parameters in a transfer function can improve discrimination between structures in the volume data given that the combined parametric information allows to disambiguate areas that fall within a given range of those parameters simultaneously. Enhanced structure discrimination is beneficial in medical image visualization where distinct tissues are approximately constant in value and values transition smoothly from one tissue to the next. There is ongoing work to support 2D transfer functions combining scalar value and gradient magnitude (the scalar field variable and its first derivative) under the US Department of Energy (DOE) Office of Science (SC) contract DE-SC0011385 grant.

6.2. Overlapping Volumes

It is currently possible to render overlapping volumes by taking advantage of the up to four independent components supported by the mapper (each component representing a different volume). The limitation of 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 can be cast through proxy geometry bounding the N overlapping volumes to be rendered and separately sampling and compositing their fetched texture values in the fragment shader.

6.3. Improved Rendering of Labeled Data

Currently, VTK supports binary masks and only a couple of specific representations 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. (Users may wish to use linear interpolation for the scalar value to look up opacity, but 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 such data. We then hope to implement more comprehensive labeled data volume rendering for both the CPU and GPU mappers.

7. 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 5) and torso (used in Figures 3, 6 and 7) datasets used in this publication. The Supernova (used for Figure 9b) and Turbulent-Combustion (used for Figure 10) 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 9a) nanoparticle dataset. The cactus sample dataset (used in Figure 4) is courtesy of Michael Holland and Dula Parkinson from Advanced Light Source, Lawrence Berkeley National Laboratory.

8. 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.

CIBC (2016). ImageVis3D: An interactive visualization software system for large-scale volume data. The NIH/NIGMS Center for Integrative Biomedical Computing. http://www.imagevis3d.org.

Clendenon, J. L., Phillips, C. L., Sandoval, R. M., Fang, S., and Dunn, K. W. (2002). Voxx: A PC-based, near real-time volume rendering system for biological microscopy. *American Journal of Physiology. Cell Physiology*, 282(1):C213–218.

Eichelbaum, S., Hlawitschka, M., Wiebel, A., and Scheuermann, G. (2010). Openwalnut–an open-source visualization system. In *Proceedings of the 6th High-End Visualization Workshop*, pages 67–78.

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

Engel, K., Kraus, M., and Ertl, T. (2001). High-quality Pre-integrated Volume Rendering Using Hardware-accelerated Pixel Shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, pages 9–16, New York, NY, USA. ACM.

Fedorov, A., Beichel, R., Kalpathy-Cramer, J., Finet, J., Fillion-Robin, J.-C., Pujol, S., Bauer, C., Jennings, D., Fennessy, F., Sonka, M., Buatti, J., Aylward, S., Miller, J. V., Pieper, S., and Kikinis, R. (2012). 3D Slicer as an image computing platform for the Quantitative Imaging Network. *Magnetic Resonance Imaging*, 30(9):1323–1341.

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.

Heng, Y. and Gu, L. (2005). GPU-based Volume Rendering for Medical Image Visualization. In *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*, pages 5145–5148.

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.

Hsu, W. M. (1993). Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering*, PRS '93, pages 7–14, New York, NY, USA. ACM.

Lorensen, W. E. and Cline, H. E. (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA. ACM.

Ma, K.-L. (1995). Parallel Volume Ray-casting for Unstructured-grid Data on Distributed-memory Architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, pages 23–30, New York, NY, USA. ACM.

Ma, K.-L. and Crockett, T. W. (1997). A Scalable Parallel Cell-projection Volume Rendering Algorithm for Three-dimensional Unstructured Data. In *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '97, pages 95–ff., New York, NY, USA. ACM.

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.

Meyer-Spradow, J., Ropinski, T., Mensmann, J., and Hinrichs, K. (2009). Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications*, 29(6):6–13.

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/.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA. ACM.

Pfister, H., Lorensen, B., Bajaj, C., Kindlmann, G., Schroeder, W., Avila, L. S., Raghu, K. M., Machiraju, R., and Lee, J. (2001). The transfer function bake-off. *IEEE Computer Graphics and Applications*, 21(3):16–22.

Ramachandran, P. and Varoquaux, G. (2011). Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51.

Rezk-Salama, C., Engel, K., Bauer, M., Greiner, G., and Ertl, T. (2000). Interactive Volume on Standard PC Graphics Hardware Using Multi-textures and Multi-stage Rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00, pages 109–118, New York, NY, USA. ACM.

Royer, L. A., Weigert, M., Günther, U., Maghelli, N., Jug, F., Sbalzarini, I. F., and Myers, E. W. (2015). ClearVolume: Open-source live 3D visualization for light-sheet microscopy. *Nature Methods*, 12(6):480–481.

Schrödinger, LLC (2015). The PyMOL Molecular Graphics System, Version 1.8. https://www.pymol.org/.

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.

Stalling, D., Westerhoff, M., Hege, H.-C., et al. (2005). Amira: A highly interactive system for visual data analysis. *The visualization handbook*, 38:749–67.

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

Wald, I., Johnson, G. P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Gunther, J., and Navratil, P. (2017). OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE transactions on visualization and computer graphics*, 23(1):931–940.

Wallis, J. W., Miller, T. R., Lerner, C. A., and Kleerup, E. C. (1989). Three-dimensional display in nuclear medicine. *IEEE transactions on medical imaging*, 8(4):297–230.

Westover, L. (1990). Footprint Evaluation for Volume Rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 367–376, New York, NY, USA. ACM.

Yagel, R. and Shi, Z. (1993). Accelerating Volume Animation by Space-leaping. In *Proceedings of the 4th Conference on Visualization '93*, VIS '93, pages 62–69, Washington, DC, USA. IEEE Computer Society.