OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets

Praveen Bhaniramka, Yves Demange*
SGI

Abstract

We present the OpenGL Volumizer API for interactive, high-quality, scalable visualization of large volumetric data sets. Volumizer provides a high-level interface to OpenGL hardware to allow application writers and researchers to visualize multiple gigabytes of volumetric data. Use of multiple graphics pipes scales rendering performance and system resources including pixel-fill rate and texture memory size. Volume roaming and multi-resolution volume rendering provide alternatives for interactive visualization of volume data. We combine the concepts of roaming and multi-resolution to introduce 3D Clip-Textures, an efficient technique to visualize arbitrarily large volume data by judiciously organizing and paging them to local graphics resources from storage peripherals. Volumetric shaders provide an interface for high quality volume rendering along with implementing new visualization techniques. This paper gives an overview of the API along with a discussion of large data visualization techniques used by Volumizer.

Keywords: volume visualization, large data, texture mapping, clip-textures.

1 Introduction

Experiments, simulations, and instrumentation devices produce ever large, complex, and detailed volumetric data, which exceed the capabilities of current visualization platforms. This increase in information generates a need for powerful computational tools to visualize such data. Volume visualization provides a way to discern details and reveal complex multi-dimensional relationships within the data.

There are a number of approaches for visualization of volume data. Many of them use data analysis techniques to find the contour surfaces inside the volume of interest and then render the resulting polygonal geometry [11]. Others generate 2D images from 3D volume data using direct volume rendering/ray marching [3, 6, 9, 14]. The 3D-texture approach is a direct data visualization technique, using textured data slices combined successively in a specific order using a blending operator [2]. In this model, a 3D texture becomes a voxel cache, and the graphics hardware processes all rays simultaneously, one 2D slice at a time. Since an entire 2D slice of voxels is cast at one time, the resulting algorithm is fast and efficient. While special purpose hardware can be used to accelerate ray casting [22, 23], the texture-based approach takes advantage of traditional graphics hardware and resources by using OpenGL 3D-texture rendering. This makes it the method of choice for many interactive and immersive volume-visualization applications.

*{praveenb, deum}@sgi.com

Interactive visualization of large volume data is an active field of research. Parker et al. [19] use multi-processor systems to achieve interactive ray tracing of large volume data resident in main memory, while Bajaj et al. [1] use out-of-core techniques for computing isosurfaces. We introduce 3D clip-textures, which combine the concepts of volume roaming and multi-resolution volume rendering to interactively visualize huge amounts of volume data resident in main memory or on disks.

This paper presents the latest release of OpenGL Volumizer, which has a new API to support development of interactive, large data volume-visualization applications. The API provides a high level interface to describe volumetric data and exploits hardware-accelerated 3D texturing by implementing intelligent system specific optimizations. It provides capabilities for visualizing extremely large volumetric data, including support for 3D clip-textures.

With 3D clip-textures, we extend the concept of 2D clip-maps [26] to 3 dimensions. We also extend the concept of multi-pass shading techniques [20] to support volumetric shaders for high quality rendering of volume data sets. In section 2, we give a brief overview of the Volumizer API. In section 3, we discuss multiple techniques for large data visualization. In section 4, we present the concepts behind 3D clip-textures along with some implementation details. We conclude with some results and a discussion on future directions of research in this field.

2 OpenGL Volumizer API

We first discuss the main design goals behind Volumizer's new API. The API provides:

1. A high-level and extensible interface for volume rendering.
2. Integrated shading capabilities, to improve visual realism and implement new visualization techniques.
3. Large data management, through the use of data paging, memory management and graphics-resource control.
4. Thread safety, to allow implementation of multi-threaded applications that run on multiple processors and graphics engines to scale performance.
5. A viewer application, with a plug-in architecture, to provide an infrastructure for further application development.
6. Inter-operability with other toolkits like OpenGL Performer™ [24], Open Inventor™ and the Visualization Toolkit (VTK) [25].

The Volumizer API provides a modular architecture consisting of two main components. This approach is used to separate volume data description from specific data rendering/visualization techniques. Figure 1 shows the API structure, using a simple example. The following sub-sections give an overview of these components.

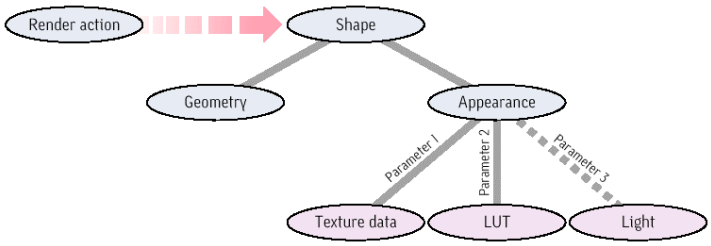


Figure 1. Volumizer API components: The Shape node and the Render Action

2.1 Data Description

OpenGL Volumizer supports a hierarchical structure to represent volumetric data and organize visualization parameters. Scene graphs have been used in the past to describe volumetric data [15]. Volumizer plugs in as the leaf node of such a scene graph. This leaf node is the *shape* node, a container for the volume's *geometry* and *appearance* (Figure 1). Geometry defines the spatial attributes and region of interest, while appearance defines the visual attributes like rendering parameters and shading description. This provides a logical separation between the 3-dimensional structure (defined by *primitives* like triangles or tetrahedra), and corresponding appearance (defined by *properties* like textures, LUTs, per-vertex colors, lighting, etc.). This separation is important since the appearance description is specific to the particular rendering technique being applied while the geometry holds more generic components of volumetric data.

2.1.1 Geometry

The region of interest is represented as the geometry component of the shape node. This geometry can either be volumetric, consisting of polyhedral primitives, or polygonal. The former needs to be *polygonized* before rendering (see section 2.2 for more details). Using geometrical techniques to render volume data sets gives the flexibility offered by traditional 3D-rendering engines:

1. Perspective views can be issued to immerse the observer in the scene by specifying a different camera model.
2. Polygonal surfaces can be embedded in the volume by rendering them first using the Z-buffer hardware (Figure 11a, 12).
3. 3D texture mapped polygonal surfaces can show correlation between different data attributes (Figure 11b).
4. Volume-warping can be implemented by specifying per-vertex texture coordinates in an irregularly sampled grid.
5. Multiple-volumes can be blended together to provide better insight into the data set (Figure 12).

Volumizer uses the tetrahedron as the basic primitive for internal operations on volumetric geometry types [17, 18]. Applications can implement new geometry classes by overriding virtual methods in the base classes.

2.1.2 Appearance

The shape's appearance defines the visual attributes of the shape. The appearance contains a list of *parameters*. They typically include properties like texture data, transfer functions, gradient volumes and other visual attributes. The appearance also contains a *shader*, which determines how to combine the parameters in order to generate the desired visualization effect.

2.1.3 Volumetric Shaders

We use the concept of volumetric shaders to generate desired visual effects from the given parameters. A shader defines multiple rendering passes per primitive, each pass with different OpenGL state, similar to the surface shading by Peercy et al. [20]. In our case, shading is applied to the polygonized geometry used to render the volume data. This is a powerful concept, which allows applications to implement very high quality visualizations using the graphics hardware. Engel et al. [5] use pixel shaders to implement pre-integrated volume rendering. Shaders can be used to implement the group nodes defined by Nadeau [15] to composite different volumes together. With our approach, applications can switch between visualization techniques by changing the shader for the shape node, rather than having to re-compute the volume data for every such change. We use shaders to implement visualization techniques like volumetric tagging and multi-volume blending. Figure 2a shows a 256 MB size medical data set rendered using a single pass, while 2b shows a two pass volumetric tagging shader applied to the data set. This shader is used to mask out arbitrary regions from the volume using a 3D-stencil buffer, in this case, a set of sine waves. 2c shows two pass volumetric lighting [21] applied to the same dataset. Figure 12 shows an example of multi-volume blending used to fuse a T1-weighted MRI with a T2-weighted MRI of a human head using two passes.

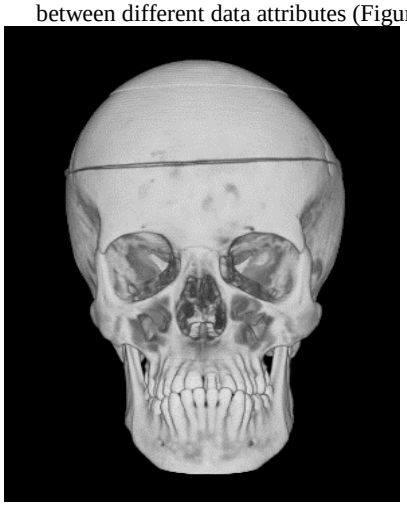


Figure 2a. Original data set rendered with 3D texture mapping

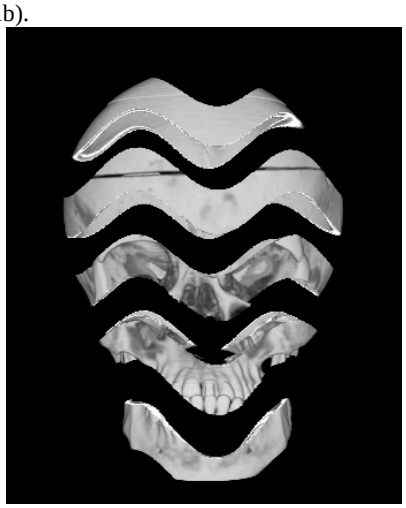


Figure 2b. A sine wave shaped 3D-stencil buffer to mask out volume data

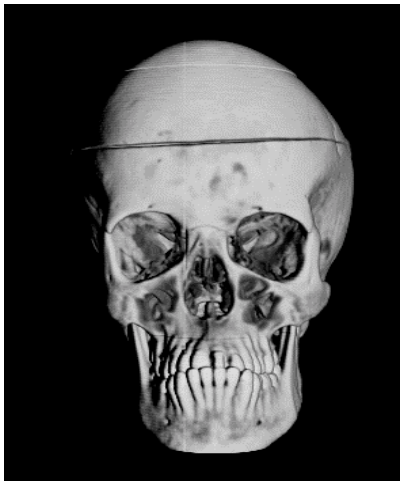


Figure 2c. Volumetric lighting to provide better depth information.

2.2 Render Actions

Multiple rendering engines are implemented through separate procedural components called *render actions*. Render actions hold all of the components to render volumetric shapes and implement different visualization algorithms to render shape nodes. Developers can integrate their own scene parameters and rendering algorithms in the API structure. This provides the ability to incorporate custom-tailored parameters and renderers and gives advanced developers the flexibility to implement and experiment with new rendering methods.

Render actions are responsible for *polygonization* of volumetric primitives to generate polygonal primitives. Polygonization is an integral part of the volume visualization process since it allows direct rendering of volume data using conventional graphics hardware. Render actions are also responsible for managing system resources depending on the visualization technique being implemented.

2.2.1 3D Texture-Mapping Render Action

The texture mapping render action (TMRenderAction) is an optimized 3D-texture-based renderer delivered with the API. It polygonizes the shape's volumetric geometry by slicing it using viewport-aligned planes. This polygonized geometry is then passed to the volumetric shader for rendering and shading. The TMRenderAction provides:

1. Support for arbitrary volumetric and polygonal geometry.
2. Optimized texture management for improved texture download and rendering performance.
3. Multi-pass shading interface for building custom shaders, along with built-in shaders for techniques like volumetric lighting and tagging.
4. Transparent bricking and interleaving of texture data, including texture-memory management for specific graphics hardware.
5. Support for multi-resolution and volume roaming techniques.

Applications specify the sampling rate to control the distance between adjacent slices of the polygonized geometry. The number of slices depends on the scene complexity and pixel-fill performance of the hardware. Choosing the number of slices to be equal to the volume's data dimensions usually works well.

Applications can control the OpenGL state for the volume rendering process. The blending function commonly used in volume-rendering applications is the *over* operator. It can be modified to implement other techniques like maximum intensity projection by using the *max* operator [13].

2.3 Integration with Other Toolkits

OpenGL Volumizer is designed to handle only the volume-rendering aspect of an application. Other toolkits, such as Open Inventor™ and Visualization Toolkit (VTK), can still be used to structure the other elements of the application. Figure 3a shows a Volumizer based shape node in an Open Inventor scene graph. Figure 3b shows a Volumizer based *mapper* integrated inside a VTK application. In this example, VTK is used to render an isosurface inside the medical dataset while, Volumizer is used for volume rendering using 3D texture mapping.

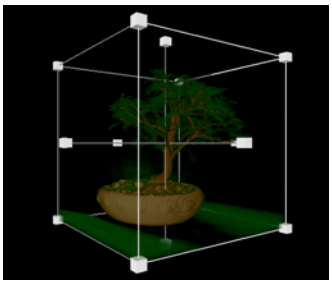


Figure 3a. Volumizer node in an Open Inventor scene graph

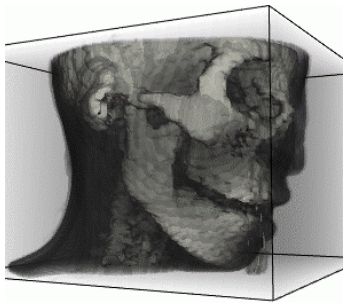


Figure 3b. Volumizer mapper inside a VTK application

For maximum compatibility with other toolkits, Volumizer uses memory management callbacks for allocation and de-allocation of memory. These callbacks can be used to allocate memory from shared memory arenas, allowing integration with APIs using a multi-processed model of execution, such as OpenGL Performer.

3 Volume Rendering Large Datasets

As the power of computing platforms and acquisition-device capabilities increase, applications using numeric simulations or data-acquisition techniques generate larger data sets. Effective visualization of such data sets is one of the most important challenges that we face today. In this section, we discuss the techniques OpenGL Volumizer uses for visualization of large volumetric data.

3.1 Resource Constraints

Before discussing the specific techniques, we should identify the resource constraints that prevent us from addressing such large data sets. In this context, by large data we mean any data larger than what local resources can handle. Also, we use the term *graphics pipe* to refer to the graphics sub-system of the visualization platform. Amongst the limiting factors for interactive and efficient volume visualization with three-dimensional textures, we consider:

- Fill-rate limitation of the graphics pipe,
- Size of texture memory on the graphics pipe,
- Size of physical memory on the computer system,
- System data bandwidth, especially between various peripherals and the graphics pipe.

When using 3D textures the application performance is dependent on the capabilities of the available graphics hardware. Our first approach uses additional graphics hardware to overcome the resource limitations mentioned above whereas the later ones manage the available resources efficiently.

3.2 Volume Bricking

A critical component in any large data visualization system is the organization of the data and optimization of its transportation between various peripherals. A common solution is to divide the volume data into smaller pieces or *bricks* [27]. The size of bricks plays an important role in the overall efficiency. Short data transfers ask for frequent interrupts in the data flow and affect performances, while long data transfers optimize the overall bandwidth but remain non-interruptible. All of the following techniques use bricking as a base for exchanging data from main memory to texture memory or from disks to main memory.

3.3 Scalable Volume Rendering

Volumizer objects are thread-safe allowing applications to scale graphics performance and resources by sharing the volume data among multiple rendering threads/processes. Figure 4 shows '*n*' pipes rendering the same scene using one thread/process per pipe.

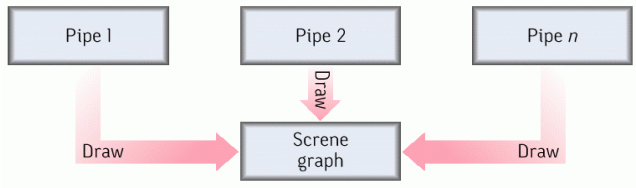


Figure 4. Scalable volume rendering

Rendering performance can be scaled using one of several hardware or software compositing schemes:

- Screen space (2D) decomposition – Scales fill rate trivially and geometry rate using view-frustum culling.
- Database (DB) decomposition – Scales fill rate, texture memory size and geometry rate.
- Time slice (DPLEX) decomposition – Linearly scales frame rate during interaction by introducing latency.
- Stereo (EYE) decomposition – Scales frame rate while in stereo mode.
- Multi-level decomposition – Mixes the above decomposition schemes using a hierarchical composition network.

We use OpenGL Multipipe SDK [16], which provides runtime configurability and scalability to our volume viewer application *volview*. Figure 5 shows an example of Database decomposition across four graphics pipes. The 70 MB head data set is decomposed into four bricks by creating four shapes. Each shape is rendered on one pipe using one render action per pipe to generate partial images. These partial images are composited in back-to-front visibility sorted order to generate the final image.

Splitting the rendering process into this hierarchy of compositions can introduce errors during image composition when using the commonly used *over* operator, (α_{src}, $1-\alpha_{src}$) for the blending function. Our approach is to pre-modulate intensity values of the volume data by corresponding alpha values and then use a different blend function, (1, $1-\alpha_{src}$) for the blending.

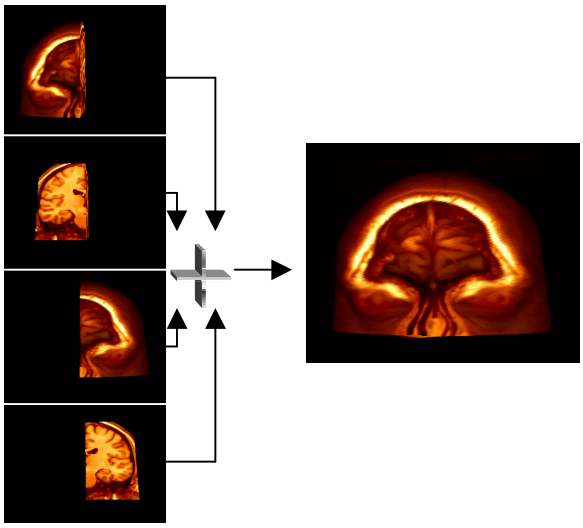


Figure 5. Database decomposition of volume data.

Lombeyda et al. [10] use a similar approach to composite images generated by multiple VolumePro boards [22].

It can be shown that the original algorithm, using the *over* operator demonstrates errors in the final composited image, while the modified technique gives the correct results. Since directly modulating the texel values would require computing and downloading the texture data for every change to the transfer function, our implementation uses a texture lookup table and modulates the transfer function itself to achieve the same effect. In this case, we only need to compute and download the lookup table when the transfer function changes.

3.4 Time-Varying Volume Rendering

Most computer simulations in the field of computational sciences produce time-varying data sets. Volumizer allows rendering such data sets by allowing applications to control the set of textures to be resident in texture memory. This allows users to run a volume movie of the simulation to visualize animated fluid dynamics or crash analysis data. Using simple out-of-core rendering, we get approximately 5 frames/sec on an SGI Onyx system using a viewport size of 1280x1024, for the quasi-geostrophic turbulent flow data in [12]. The data set has 1492 steps, each time step being a 3D texture of 16 MB. Several other techniques can be used to improve the visualization of such data sets like Time Space Partitioning trees [4] and texture compression [12].

3.5 Volume Roaming

Volume roaming allows the user to explore large volumetric data by interactively moving a volumetric probe inside the volume. The probe allows users to navigate the data set using a viewing window and helps them concentrate on a specific section of the whole data set.

Figure 6 illustrates the concept of volume roaming. The key components of the technique are texture bricking, intelligent texture and main memory management, and asynchronous disk paging of volume data. The application maintains a hierarchy of windows, which are smaller subsets of the total volume data, updated during user motion. Each window is sub-divided into multiple shapes, one for each brick. As the window moves, the bricks are updated with new texture data. All the window management and data transfer between the various peripherals is controlled by the application in this case. The TMRenderAction efficiently pages in the new data into texture memory from main memory.

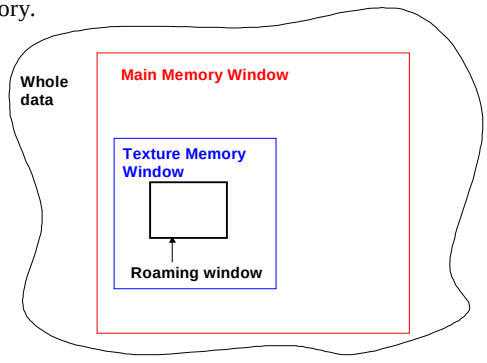


Figure 6. Volume roaming a large data set

Roaming allows an application to overcome fill rate, texture memory and main memory size constraints, with the limitations

of rendering only a sub-section of the whole volume data at a time and not providing constant frame rates during fast user motion. Figures 11a and 11b show volumetric probes used to roam a seismic data set using different geometries.

3.6 Multi-resolution Volume Rendering

Multi-resolution volume rendering allows applications to interactively render large volume data by assigning varying levels of detail (LOD), thus making a trade-off between performance and image quality. Volume data is processed to compute different levels-of-resolution of the data set by filtering and sub-sampling the original data. Many researchers have worked on multi-resolution techniques for interactive volume rendering, typically using an octree decomposition of the whole volume as in Figure 7 [7, 8, 28].

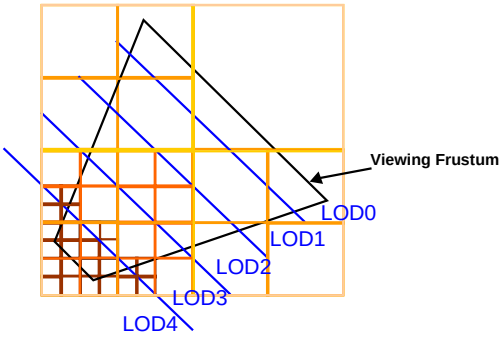


Figure 7. Multi-resolution volume rendering

The key components of this technique are texture bricking, intelligent texture memory management, and proper computation of LOD levels. In this case, a shape is used to represent each node in the octree. The TMRenderAction manages the texture data and multiple LUTs used to compensate for the different opacities at the LOD levels [7, 28]. Applications can improve the performance by rendering low-resolution data at non-leaf nodes, during user interaction and then improving the image quality as the interaction stops. This is similar to the approach used by Laur et al. [8] for hierarchical splatting. Low resolutions help improve rendering performance by limiting texture memory and fill-rate consumption of the application. One of the primary limitations of this technique is that the volume data along with the LOD levels needs to be resident in main memory. This limits the total size of the data set that can be rendered using this technique. Figure 8 shows a 300 MB size brain data set rendered using multi-resolution techniques on InfiniteReality2 graphics (64 MB texture memory).

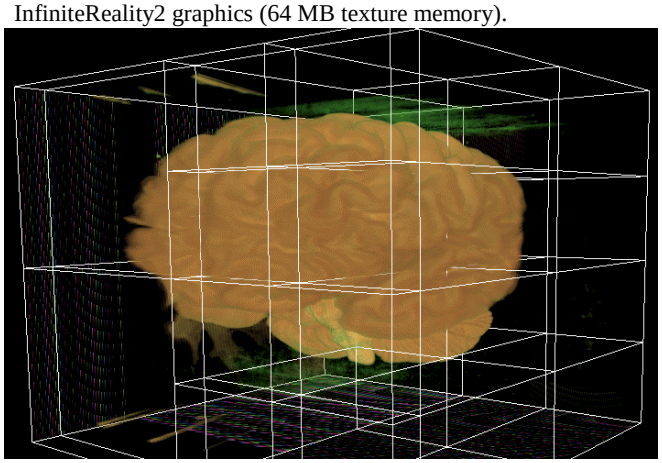


Figure 8. Brain data rendered using multi-resolution

4 3D Clip-Textures

From the above discussion, we realize that even though multi-resolution volume rendering and volume roaming are attractive techniques for rendering volumetric data, they are restricted while dealing with extremely large data sets. 3D clip-textures allow applications to visualize arbitrarily large volumetric data, by merging the advantages of volume roaming and multi-resolution techniques. They provide an intuitive scheme for visualizing multi-gigabyte volumetric data sets. When viewing the whole data set, the application does not need to render the full resolution volume data since each pixel is equal to the screen space projection of multiple voxels. When more detail is required, the user would zoom into or roam the data set and only a smaller subset of the whole data would be visible. 3D clip-textures allow merging these two schemes into one interface.

Figure 9 illustrates the concept of clip-textures in 2D. 2D clip-textures have been used successfully to provide interactive navigation of very large terrain data [26]. Clip-textures are mip-mapped versions of the original texture data except that each mip-map level maintains a roaming window (physical memory window in Figure 9) to limit the amount of texture data resident in main memory. These 'clipped mip-map levels' are called *clip-levels*. The highest level of resolution in the hierarchy corresponds to the original texture data. The remaining levels are computed by filtering and decimating the original data.

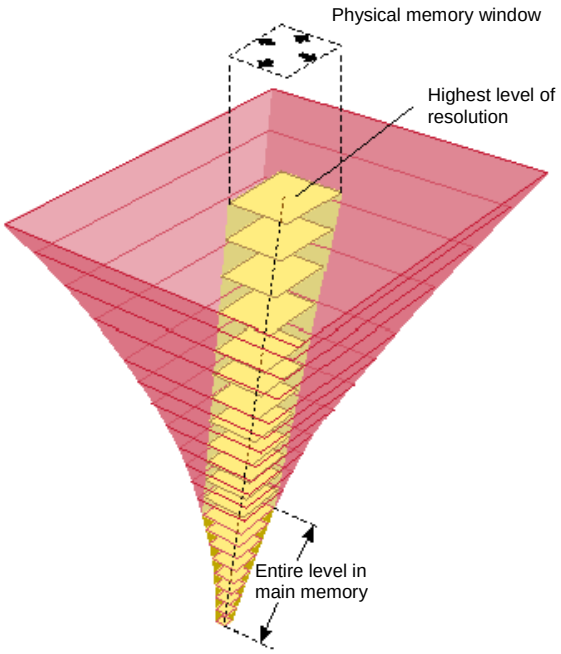


Figure 9. Clip-texture hierarchy in 2-dimensions

The center of the physical memory window is usually the viewer's center of interest. As the viewer moves, the center of interest is updated and the texture data, which is not in the window anymore is replaced by new data from disk. This data is paged into slots vacated by data being paged out of the window. This mapping ensures constant memory usage during user interaction [26]. As shown in Figure 9, lower resolutions of texture data fit completely in main memory. During periods of fast user motion, these low-resolution textures are rendered, while high-resolution data is being paged in. As higher resolution texture data is available, it is rendered to improve the

image quality of the visualization. This mechanism provides the capability to interactively visualize very large texture data resident in main memory or on high-performance disks.

Special OpenGL graphics hardware, like InfiniteReality graphics, has built in support for clip-textures but only in 2D. We implement a software emulation for 3D clip-textures. 3D clip-texture is an extension of the 2D scheme combining volume roaming with multi-resolution volume rendering. In this case, the data transfer process is supported by representing the whole clip-texture hierarchy as a collection of smaller 3-dimensional bricks at each level of resolution. This combines the benefits of bricked volume files, asynchronous disk-paging, multi-resolution and volume-roaming methods to overcome memory and pixel-fill constraints.

The core of Volumizer's large-data API is the abstraction of a 3D clip-texture and its associated render action. The implementation of clip-textures is exposed as a new parameter class, vzParameterClipTexture, and an associated render action, vzClipRenderAction [17, 18]. Texture data is paged into system's main memory from storage devices using asynchronous disk paging, implemented in the clip-texture emulation system. The main memory to texture memory transfer is performed by the clip-renderer, which employs the texture management built into the TMRenderAction.

When using clip-textures, the volumetric data to be visualized is assumed to be too big to fit in main memory of the system. This resource constraint means that the data will reside on slower and larger storage peripherals like disks instead of on local graphics resources. This data will have to migrate from one peripheral to others within the frame time. From this point of view, data migration becomes the main bottleneck for visualization, though less so on high bandwidth and low latency architectures like the SGI Onyx series.

4.1 Clip-Texture implementation

The new vzParameterClipTexture parameter class provides an abstraction for the 3D clip-texture hierarchy. It maintains the set of clip-levels and manages the amount of physical memory used to store the texture data. It handles bricking of texture data and pages these bricks from disk depending on application provided parameters. The following parameters are used to initialize the clip-texture hierarchy:

- Brick dimensions – Used to compute the number of clip-levels and optimize the data transfer on the underlying hardware architecture.
- Physical memory size – Used to limit the amount of physical memory allowed to load texture data. Controls the size of the physical memory windows at each clip-level.
- Data loader callback – Invoked by the clip-texture to load texture data from disk.

Depending on the brick dimensions, the clip-texture initializes the various clip-levels. Each clip-level is assigned a maximum physical memory window, the size of which is computed from the total physical memory allowed for the clip-texture.

When the application starts, multiple loader threads are created, which invoke the data loader callbacks to load texture bricks from disk. The set of bricks to be loaded is placed on a queue. This load queue is sorted every time the user moves using a cost function proportional to the distance of the brick from the viewpoint. The loader threads remove the brick in the front of the queue and invoke the data loader callback for the brick. When this callback returns, the 'presence flag' in the brick is updated to reflect the presence of the brick in main memory. The application updates the following parameters for the clip-texture depending on the user interaction:

- Center-of-interest – Used to update the center of the physical memory windows of the clip-levels and sort the load queue.
- Roaming window size – Used to update the size of the physical memory windows. The actual physical memory window size is limited by the total physical memory, which can be used.

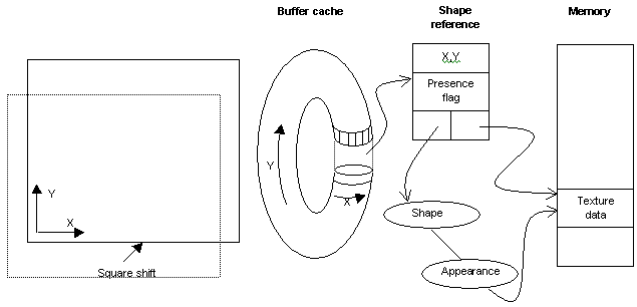


Figure 10. Toroidal mapping to page data from disk

The window management mechanism is implemented using a 4-dimensional toroidal mapping technique at each level in the hierarchy. Figure 10 shows this using a 2D example in which case the mapping scheme represents a 3D torus. As the user moves, the physical memory window is updated, and all the bricks which are not in the window anymore, are pushed on the load queue to be reloaded by loader threads. This toroidal mapping scheme exploits frame-to-frame coherence by reusing cached texture data in subsequent frames and ensures constant memory usage.

Each clip-level maintains a separate toroidal map, which is updated independently. The disk paging mechanism performs predictive loading of textures depending on the user's direction of motion. The multi-threading scheme is optimized to get maximum usage of the disk bandwidth available on the system.

4.2 Clip-Texture Render action

The large data API provides a new render action, which is built as a layer on top of the existing TMRenderAction. This render action implements intelligent texture paging techniques to render the clip-texture hierarchy in a view-dependent fashion using a depth first traversal scheme. The render action performs view-frustum as well as geometry culling, to discard bricks, which are not visible in the current frame. The bricks are rendered in a back-to-front visibility sorted order. In addition, bricks, which are closer to the viewpoint, are rendered at a higher resolution than those further away.

The level of interaction and image quality of the rendering process can be controlled using the following parameters:

- Total texture memory – Used to limit the total texture memory usage. This is done to allow other textures to be resident in texture memory at the same time.

- Number of texels rendered per frame – Used to control the pixel fill overhead, assuming that the sampling rate used is proportional to the data dimensions of the texture bricks.
- Number of texels downloaded per frame – Used to control the overhead incurred due to amount of textures downloaded from main memory to texture memory during user motion.
- LOD threshold value – Used to tradeoff image quality with rendering time during user interaction.

The LOD selection scheme used to decide whether a given brick should be rendered or not is computed by projecting the brick's bounding box to screen space and then comparing it with the application provided threshold value. The following algorithm is used for selecting a brick '*b*' to be rendered where, projection(*b*) is the screen space projection of *b*'s bounding box –

```
if b intersects shape's geometry
  if b intersects the viewing frustum
    if texture data for b is in main memory
      if projection(b) > LOD threshold
        if b fits in texture memory
          draw(b)
```

The renderer maintains a list of bricks rendered in the current frame. In subsequent frames, as the user moves, only the bricks which are not rendered anymore, are reloaded into texture memory. This mechanism is similar to the disk paging scheme used by the clip-texture implementation. In this case, the texture download is efficiently implemented in OpenGL using texture sub-loads to provide better download performance.

The renderer allows applications to roam the clip-texture, by modifying the volumetric geometry for the shape. This geometry provides the region of interest in the volume data and can be moved around and re-sized to navigate the data set interactively. In order to maintain near constant frame-rates during user motion, the render action performs predictive texture downloads to distribute the overhead of the data transfer over multiple frames. This is done using the direction of motion of the probe and then by computing the differential of the current and predicted positions and downloading this difference over a sequence of multiple frames. While in roaming mode, only textures at the same level of resolution are rendered. This special case is implemented by traversing the hierarchy and finding the collection of bricks of the highest resolution possible, each of which satisfy the above criteria.

5 Results

The clip-texture API has been utilized to interactively visualize very large volumetric data sets of sizes up to 20 gigabytes. Note that the technique does not impose any restrictions on the data size and can be used to render arbitrarily large data sets resident on disk or other peripheral devices. The image quality, however, is limited by the amount of texture memory available on the graphics pipe and the data transfer rate from the disk to system memory. Figures 13 and 14 show the segmented and classified version of the visible male data set rendered interactively using the clip-texture renderer on an Onyx system with InfiniteReality3 graphics (256 MB texture memory). The total data set is 6.77 GB in size (1760x1024x1878 unsigned short) and only 1 GB was allowed to be resident in main memory. The clip-levels were computed using a brick size of 4 MB (128x128x128) to generate 5 clip-levels (including the original level) for the hierarchy. Figure 13a shows the whole data rendered with volumetric lighting in multi-resolution mode. 13b shows the wireframe for the rendered bricks to show the different clip-levels. Figure 14b shows a zoomed in view with transparency and color for different organs. Bricks further away from the viewing point are rendered at a lower-resolution than the bricks, which are closer (notice the difference between the left and right hands). Figure 14c shows the full resolution data rendered using roaming mode of clip render action to show the left knee of the visible male. The frame rate for the visualization varies from 2 fps to 30 fps depending on the total amount of texture memory being used, rendering mode (multi-resolution vs roaming), image resolution, shading technique, etc.

6 Conclusion

We have given a brief overview of the Volumizer API and presented the techniques it employs to address large volume visualization. We have presented 3D clip-textures as a new technique for interactive visualization of very large volumetric data sets. Future work includes further optimizing clip-textures to take advantage of multiple graphics pipes and processors, and data compression techniques. Extending clip-textures to allow rendering multiple overlapping volumes representing different attributes would require more sophisticated resource management. Mixing this technique, with level-of-detail polygonal data would allow interactive rendering of LOD iso-surfaces with volume data. For Volumetric shaders, we want to provide a general-purpose interface like OpenGL Shader™ to describe volumetric shaders using a simple and comprehensive shading language. OpenGL Volumizer is available free via download from www.sgi.com/software/volumizer.

7 Acknowledgements

We are extremely grateful to Marc Olano, who provided important feedback on the clip-texture and shader APIs and helped us review the paper. We would like to thank Yair Kurzion and Alex Chalfin for fruitful discussions on clip-textures. Stefan Eilemann, Claude Knaus and Patrick Bouchaud helped implementing the scalability-related features in *volview*. We also thank the rest of the Volumizer team, Jenny Zhao and Maria Tovar, for helping us write this paper. Colin Holmes provided the head data sets used in Figures 5, 8 and 12. Sebastien Guillon, from TotalFinaElf, provided Figures 11a and 11b.

8 References

[1] Bajaj, C.L., Pascucci, V., Thompson, D., Zhang, X.Y. "Parallel Accelerated Isocontouring for Out-of-Core Visualization". IEEE Parallel Visualization and Graphics Symposium, 1999.

[2] Cabral, B., Cam, N., and Foran, J. "Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware," Symposium on Volume Visualization, 1994.

[3] Drebin, R.A, Carpenter, L., and Hanrahan, P. "Volume rendering". Proceedings of Computer Graphics (SIGGRAPH 1988).

[4] Ellsworth, D., Chiang, L.J., Shen, H.W. "Accelerating Time-Varying Hardware Volume Rendering Using TSP

Trees and Color-Based Error Metrics." Volume visualization and Graphics Symposium, 2000.

[5] Engel, K., Kraus, M, Ertl, T. "High-Quality Pre-Integrated Volume Rendering using Hardware-Accelerated Pixel Shading." SIGGRAPH/Eurographics Workshop on Computer Graphics Hardware, 2001.

[6] Hall, P.M., and Watt, A.H. "Rapid volume rendering using a boundary-fill guided ray cast algorithm". Proceedings of CG International 1991.

[7] LaMar, E, Hamann, B, and Joy, K.I. "Multiresolution Techniques for Interactive Texture-Based Volume Visualization". Proceedings of IEEE Visualization, 1999.

[8] Laur, D., Hanrahan, P. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. Proceedings of SIGGRAPH 1991.

[9] Levoy, M. "Efficient ray tracing of volume data". ACM Transactions on Graphics (TOG), Volume 9 Issue 3, July 1990.

[10] Lombeyda, S., Moll, L., Shand, M., Breen, D., Heirich, A. "Scalable Interactive Volume Rendering Using Off-the-Shelf Components" IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2001.

[11] Lorensen, W., Cline, H. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm" Computer Graphics, 1987.

[12] Lum, E.B., Ma, K.L., Clyne, J. "Texture Hardware Assisted Rendering of Time-Varying Voume Data". Proceedings of IEEE Visualization 2001.

[13] McReynolds, T., and Blythe, D. "Advanced Graphics Programming Techniques Using OpenGL". SIGGRAPH 1998 Course Notes.

[14] Meiβner, M., Huang, J., Bartz, D., Mueller, K., Crawfis, R. "A Practical Comparison of Popular Volume Rendering Algorithms". Volume visualization and Graphics Symposium, 2000.

[15] Nadeau, D.R. "Volume Scene Graphs". Volume visualization and Graphics Symposium, 2000.

[16] OpenGL Multipipe SDK http://www.sgi.com/software/multipipe/sdk/

[17] OpenGL Volumizer Programmer's Guide http://www.sgi.com/software/volumizer/

[18] OpenGL Volumizer Reference Manual http://www.sgi.com/software/volumizer/

[19] Parker, S., Parker, M., Livnat, Y., Sloan, P.P., Hansen, C., Shirley, P. "Interactive Ray Tracing for Volume Visualization". IEEE Transactions on Computer Graphics and Visualization, 1999.

[20] Peercy, M., Olano, M., Airey, J., Ungar, P.J. "Interactive Multi-pass Programmable Shading". Proceedings of SIGGRAPH 2000.

[21] Peercy, M., Airey, J,, Cabral, B. "Efficient Bump Mapping Hardware". Proceedings of SIGGRAPH, 1997.

[22] Pfister, H., Hardenbergh, J., Knittel, J., Lauer, H., Seiler, L. "The VolumePro real-time ray-casting system." Proceedings of SIGGRAPH, 1999.

[23] Ray, H, Pfister, H., Silver, D., Cook, T.A. "Ray Casting Architectures for Volume Visualization" IEEE Visualization and Computer Graphics 1999

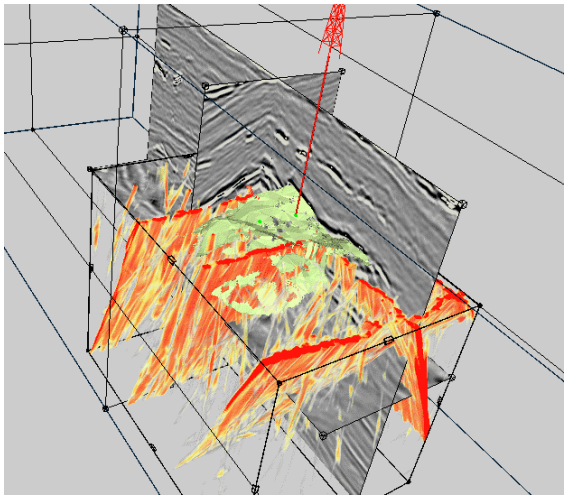
[24] Rohlf, J., Helman, J. "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics". Proceedings of SIGGRAPH 1994.

[25] Schroeder, W.J., Martin, K.M., Lorensen, W.E. "The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization". http://www.kitware.com/VTK/

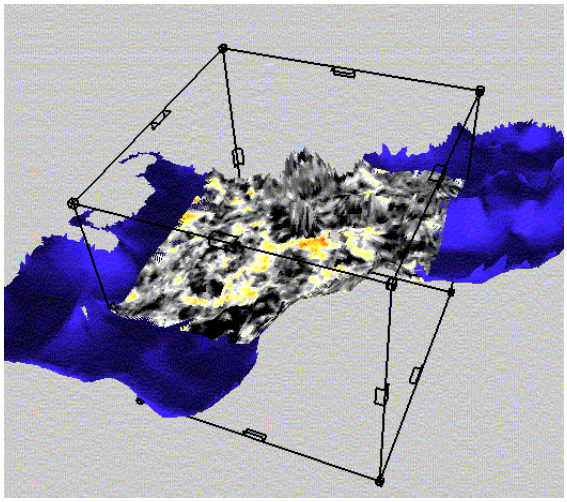
[26] Tanner, C.C., Migdal, C.J., Jones, M.T. "The Clipmap: A Virtual Mipmap." Proceedings of SIGGRAPH 1998.

[27] Volz, W.R. "Gigabyte Volume Viewing using Split Software/Hardware Interpolation". Volume visualization and Graphics Symposium, 2000.

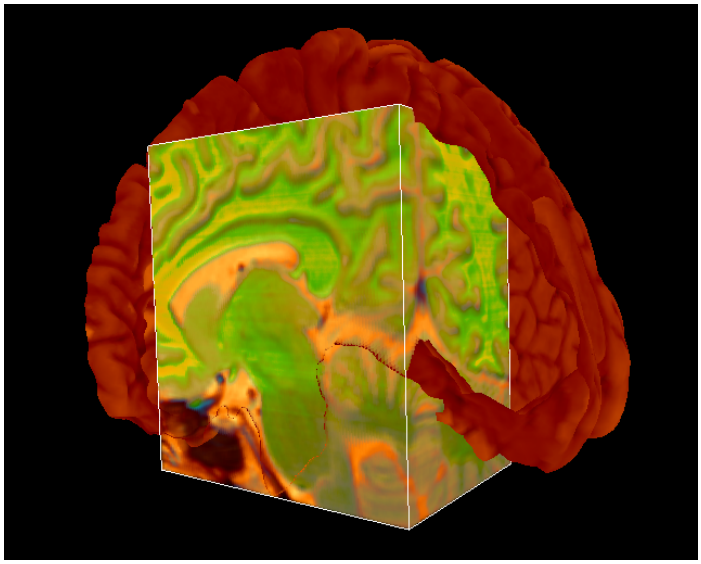
[28] Weiler, M., Westermann, R., Chuck Hansen, C., Zimmermann, K., and Ertl, T. "Level-Of-Detail Volume Rendering via 3D Textures". Volume Visualization & Graphics Symposium, 2000.



11a. Volume rendered data mixed with polygonal data to show a fault.(Courtesy TotalFinaElf)



11b. A 3D texture mapped isosurface, showing co-relation between two different attributes. (Courtesy TotalFinaElf)

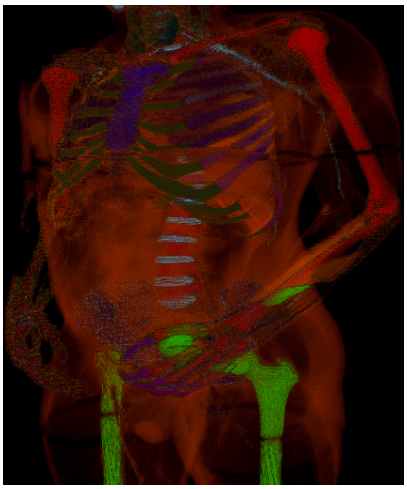


12. Multi-volume blending with iso-surface for the brain.

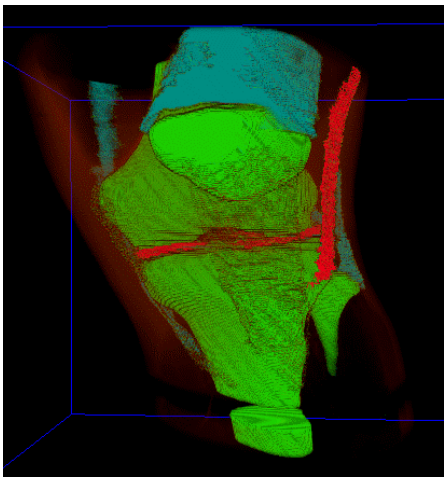
13a.Whole body with volumetric lighting

13b.Wireframe showing the various clip-levels

14a.Whole body with transparency



14b. Zoomed in with transparency and color to show different organs



14c. Roaming mode to visualize the left knee at full resolution