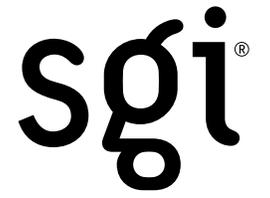


White Paper

OpenGL Volumizer™ 2



Experiments, simulations, and instrumentation devices are continuously producing larger, more complex, and more detailed volumetric data. Along with this apparent increase in information, comes a greater need for more powerful computational tools to visualize such data. Volume visualization provides a way to discern details within the data while potentially revealing complex 3D relationships. This paper presents OpenGL Volumizer 2, a new application programming interface (API) from SGI® for interactive, high-quality, scalable volume visualization.

Volume Visualization

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 geometry with transparency. The 3D-texture approach is a direct data

visualization technique using textured data slices that an API or application combines successively in a specific order using a blending operator [Cabral, 1994; Drebin, 1988]. 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 the voxels is cast at one time, the resulting algorithm is much faster with hardware-accelerated textures than with ray casting. This technique takes advantage of graphics hardware and resources by using OpenGL® 3D texture rendering, which allows applications to reach real-time performance and makes this 3D texture-based approach the method of choice for interactive and immersive volume-visualization applications. The 3D-texture approach is equivalent to ray casting and produces similar results. Unlike ray casting, in which each image pixel is built up ray by ray, this approach takes advantage of spatial coherence. A comparison of these techniques is shown in Figure 1.

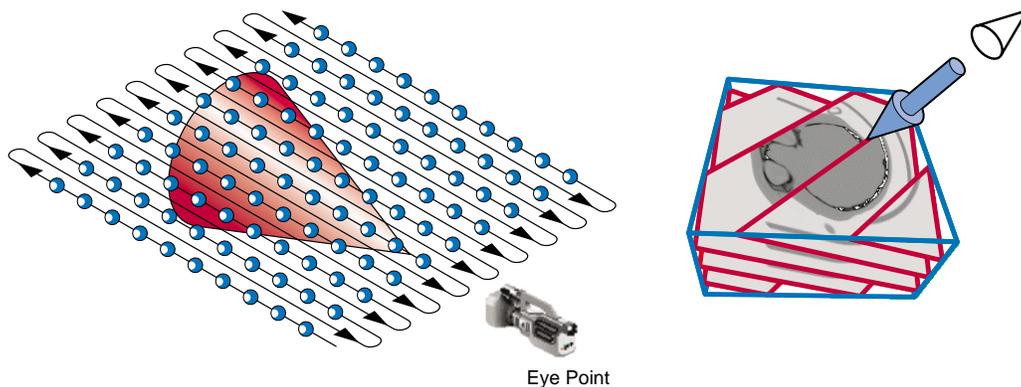


Figure 1 Ray Casting vs. 3D Texture Mapping

To help application programmers develop interactive and immersive volume-visualization methods that exploit hardware-accelerated 3D texturing, SGI designed and implemented OpenGL Volumizer, a revolutionary API providing groundbreaking capabilities for traditional volume-visualization applications and allowing application developers to treat volumetric and surface data equally.

OpenGL Volumizer 2 should be distinguished from its predecessor, OpenGL Volumizer 1. Although it has the same objectives as OpenGL Volumizer 1, OpenGL Volumizer 2 is a separate product with a newly designed API. The new API is a high-level, C++, volume rendering API that supports management and visualization of large volume datasets. This white paper addresses the characteristics and features of OpenGL Volumizer 2, which is hereafter simply referred to as OpenGL Volumizer.

Product Overview

Announced during SIGGRAPH 2001, OpenGL Volumizer considerably simplifies the programming model while offering new capabilities and features. This makes visualization of extremely large volumetric datasets easier on multiple platforms. It provides the following:

- A high-level, extensible, C++ API that segments classes and methods based on the corresponding procedural-versus-descriptive nature of the component members. The core API consists of a volumetric-shape description API and a procedural 3D texture-based render action.
- Thread safety, which allows implementation of multithreaded applications that run on multiple processors and graphics engines in conjunction with APIs like OpenGL Multipipe™ SDK and OpenGL Performer™.
- Integrated shading capabilities to perform volumetric shading which allows techniques like multivolume blending and volumetric

lighting to improve realism and to implement very high quality visualizations.

- Large data management capabilities, including support for 3D clip textures, which allow interactive visualization of extremely large datasets.
- Examples that include a transfer function editor, data loaders, and a volume rendering application for multipipe systems, along with sample integration with existing APIs.
- A container for volume rendering techniques. Developers can integrate their own scene graph parameters and rendering algorithms in the API structure. The ability to incorporate such custom-tailored parameters and renderers gives the flexibility to advanced developers to implement and experiment with new rendering methods.

Figure 2 shows the various modules of the API. At this time, all modules, with the exception of the Shirley-Tuchman renderer, are included with the OpenGL Volumizer distribution.

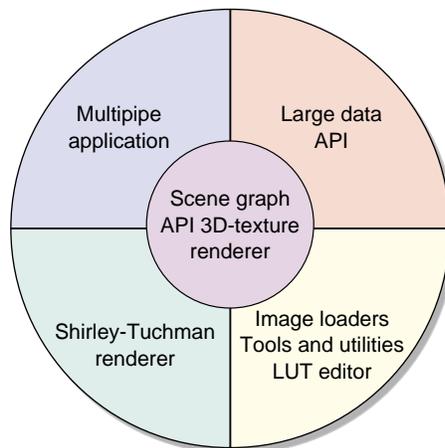


Figure 2 Modular Architecture of OpenGL Volumizer

OpenGL Volumizer API

OpenGL Volumizer supports a hierarchical scene graph structure to retain and organize visualization parameters. The leaf node of the volumetric scene graph is the shape node [vzShape], which is a container for its geometry and appearance. The volume's geometry defines the spatial attributes and a ROI, and the volume's appearance defines the visual attributes, such as

rendering parameters. The appearance itself consists of a list of parameters that are specific to the particular rendering technique being applied to the shape. Appearance parameters act as data containers for the render action. They typically retain the volume data itself as well as other shading parameters, such as light direction or lookup tables [LUTs], if needed. Figure 3 shows a sample shape node with the corresponding geometry and appearance.

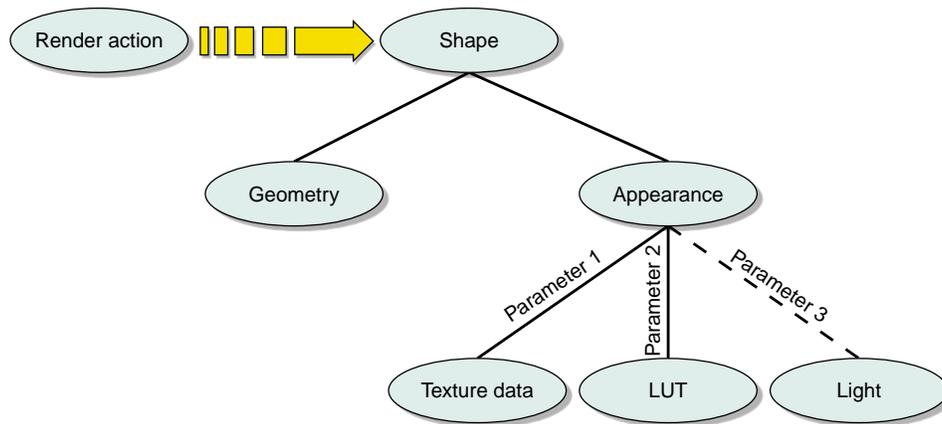


Figure 3 Shape Node and Its Associated Render Action

Render actions are implemented as a separate class derived from vzRenderAction and hold all of the components to render the shape. They primarily implement different visualization algorithms to render shape nodes. The render action is also responsible for managing the resources needed to render the nodes. TMRRenderAction is a 3D texture-based renderer delivered with the API. The render action polygonizes the shape's volumetric geometry by slicing it using viewport-aligned planes. It then applies the other shading parameters, such as 3D textures and a LUT. Separating the shape node's description from the rendering techniques allows the possibility of implementing custom render actions. Adding parameters, defining new shaders, and deriving the right render action will provide a custom rendering method.

The object classes [derived from vzObject] in the API are thread and/or MP safe. This allows them to be shared across multiple threads and/or processes running in parallel to render the shape[s] concurrently on several graphics engines.

OpenGL Volumizer simplifies memory allocation and deallocation of the objects. All objects in the scene graph are reference-counted and automatically deleted when the reference count reaches zero. The vzObject class is derived from the base class vzMemory, which allows the application to specify memory allocation and deletion callbacks. The vzMemory class can be used for allocating memory from shared memory arenas. This is essential for integration with APIs that use a multiprocessed model of execution, such as OpenGL Performer.

Volumetric Geometry

In OpenGL Volumizer, the ROI is represented as the geometry component of the shape node and described apart from the shape's appearance. Using this approach allows the separation of the geometry or the spatial attributes of the shape from the visual attributes. This separation is important since the appearance is specific to the rendering technique being applied to the shape. Figure 4 shows an example of an appearance applied to two shapes with different volumetric geometries.

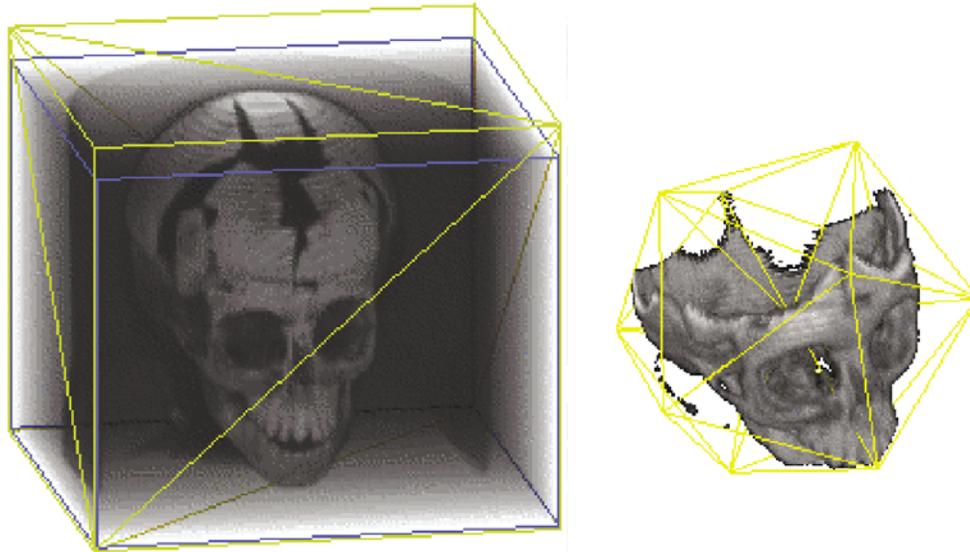


Figure 4 Different Volumetric Geometries for a Volume Dataset

OpenGL Volumizer allows specification of arbitrary volumetric geometry through the use of simple primitives ranging from axis-aligned cubes to arbitrary tetrahedral meshes. Just as triangles are base primitives used to describe polygonal geometry, a tetrahedron is the base primitive used to describe volumetric geometry. Hence, the API uses the tetrahedron as the basic primitive for all its operations by tessellating all other geometric representations into tetrahedral meshes. For example, a cube can be represented with as few as five tetrahedra. This tessellation process is transparent to the application for the built-in geometry classes and allows applications to write their own geometry classes by overriding the appropriate virtual methods in the base class `vzVolumeGeometry`.

Using geometric techniques to render volume datasets gives the following flexibility offered by traditional 3D-render engines:

- Perspective views can now be issued to immerse the observer in the scene. By simply specifying a different camera model, applications can switch between parallel and perspective projections. Perspective transformations are an integral part of 3D graphics languages and are accelerated by the geometry and the texture mapping engines.

- Polygonal surfaces can be embedded in the volume by rendering them first. The Z buffer, hardware ensures that they correctly appear to lie within the volume. For example, a corona-prosthesis model can be easily inserted in MRI- or CT-scanned data from a patient.
- 3D texture mapped polygonal surfaces can show correlation between different data attributes.
- Multiple volumes can be blended together to provide better insight into the data set.

3D Texture Mapping Render Action

The 3D texture-based renderer [TMRenderAction] delivered with OpenGL Volumizer implements a semitransparent plane-rendering technique. The underlying method is composed of two parts. First, the volume geometry is sliced with planes parallel to the viewport and stacked perpendicular to the direction of view. These planes will be rendered as polygons clipped to the geometry primitives' boundaries. During each frame, this polygonization phase generates a set of polygons normal to the viewing direction. This method of slicing is shown in Figure 5.

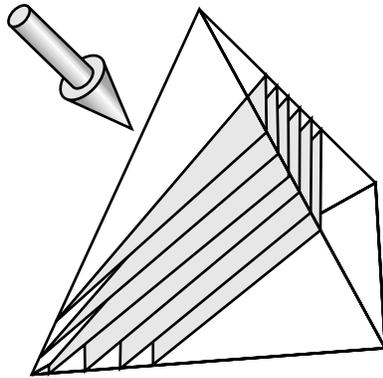


Figure 5 Tetrahedral Slicing

These clipped polygons are textured with the volume data they intersect, and the resulting images are alpha-blended together from back to front toward the viewer's position. Each polygon's pixels are successively drawn and blended into the framebuffer to provide the appropriate transparency or color effect. The polygonization phase can be executed in parallel on the next frame while the current frame is rendered.

To improve image quality while taking into account rendering performance, the application must specify an appropriate sampling rate. The sampling rate controls the distance between the adjacent slices of the polygonized geometry. The

number of slices to be used depends on the scene complexity and the pixel-fill performance of the hardware. This paper elaborates the tradeoff between image quality and performance in the section titled "Understanding the Texture Mapping Render Action".

Slicing with planes, as shown in Figure 5, is common, but artifacts can appear when the observer is very close to the model. As an implementation alternative, spherical slicing provides more accurate visualization in perspective projection [McReynolds, 1998]. This principle is illustrated in Figure 6.

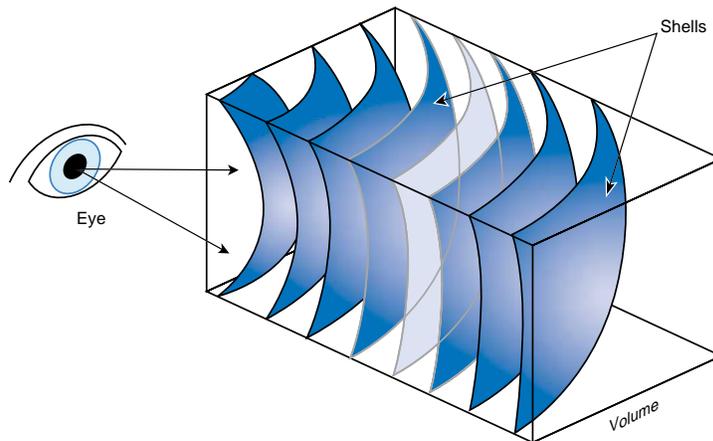


Figure 6 Spherical Slicing

In this case, the polygonization process might become the performance bottleneck. Using a parallel algorithm to perform the polygonization on multiple processors will help maintain a good level of performance.

The advantages of TMRRenderAction include the following:

- Immediate-mode execution to prevent the overhead of storing transient geometry from polygonization.

- Optimized texture management for improved texture download performance. [This includes the case of texture memory oversubscription.]
- Support for custom volumetric shading techniques along with built-in shaders for volumetric lighting and tagging.
- Transparent bricking and interleaving of texture data.
- Support for applications using multi-resolution and volume roaming techniques.

Volumetric Shaders

OpenGL Volumizer introduces the concept of volumetric shaders to apply specific rendering techniques to generate desired visual effects using the same rendering algorithm described earlier.

Each shader implements a particular technique by setting the appropriate OpenGL state and using multiple rendering passes if necessary.

TMRenderAction supports multiple built-in shaders that accept parameters for the particular technique being applied. Additionally, applications can implement custom multipass shaders using the vzTMSHader class. Figure 7 shows the results generated from three different shaders applied to the same medical dataset. The first frame on the left shows the original dataset rendered with 3D texture mapping. The middle frame shows the use of a sine-wave-shaped 3D-stencil buffer to mask out volume data, and in the third frame volumetric lighting was used to provide better depth information and improve visual realism.

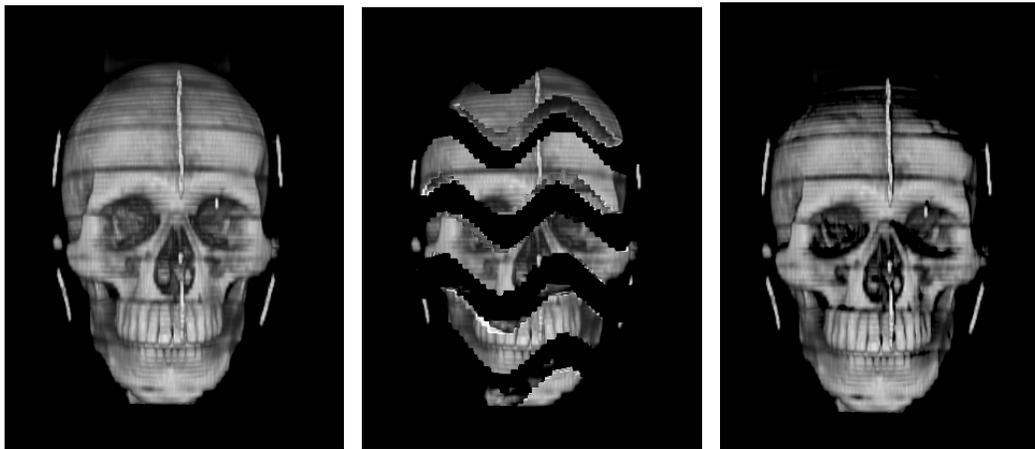


Figure 7 Volumetric Shading Examples

Transfer Functions

For effective visualization of datasets, the data values often need to be mapped to different color and opacity values [Levoy, 1990]. This mapping is specified using transfer functions implemented as LUTs supported in the graphics pipeline. Different alpha values in volumetric data often correspond to different materials in the volume that is being rendered. A nonlinear transfer function can be applied to the texels to help analyze the volume data, highlighting particular classes of volume data. By graphically thresholding values, users can visually extract surfaces in real time. OpenGL Volumizer

implements a lookup table [LUT] parameter, mapping color and opacity values after texture interpolation. To edit transfer functions, a simple LUT editor is delivered with the product.

Understanding the Texture Mapping Render Action

This section explains the details of the render action and mentions a few techniques that can be employed by application writers. Figure 8 shows the pipeline used by a typical volume rendering application using TMRenderAction.



Figure 8 Pipeline Used by a Volume Rendering Application Using TMRendAction

The application first computes the number of shapes it needs to keep resident in texture memory for the given frame. The list of shapes might be the outcome of visibility culling in an immersive application, the current frame index of a time-varying simulation, and so on. Once the application is done managing and unmanaging the shapes for the current frame, it is ready to draw them.

TMRendAction does not perform any visibility sorting of the rendered shapes; hence, it is the application's responsibility to sort them in the correct order. After the sort, the application sets the appropriate OpenGL state (such as enabling blending and setting the appropriate blending functions) for performing volume rendering. TMRendAction renders the polygonal geometry in a back-to-front sorted order. The blending function for the most common volume rendering application is the over operator `glBlendFunc [GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA]` [McReynolds, 1998].

The flexibility in choosing the blending function allows the application writer to implement other techniques by setting the appropriate blending equations. For example, maximum intensity projection can be implemented by using `glBlendEquation[GL_MAX]` [McReynolds, 1998].

At this point, the application notifies the render action that it is ready to start drawing the shapes by calling the `beginDraw` method. The `beginDraw` method marks the end of the texture management phase and the beginning of the rendering phase. Inside the method, the render action does the following:

- Computes the total resources required for the list of managed shapes
- Performs the OpenGL state management (pushes OpenGL state applications, stores transformation matrices, and so on.)

- Performs the OpenGL resource management (creates and downloads texture objects, LUTs, and so on.)

Next, the application draws all of the shapes in the visibility sorted order that was computed earlier. Inside each `draw` method, the render action does the following:

- Invokes the shader's initialization routine, which sets the appropriate OpenGL state (bind texture objects, enable LUTs, and so on.)
- Polygonizes the volumetric geometry using the transformation matrices
- Draws the polygonized geometry in a back-to-front order

Note that the polygonized geometry is always parallel to the viewport unless the application has set slicing planes on the volumetric geometry. The transformation matrices are queried directly from OpenGL in the `beginDraw` method. These matrices are stored and used for all the subsequent draws before the next `endDraw` is called. Finally, in the `endDraw`, the render action restores the OpenGL state that it modified, including texture-related settings, LUTs, and pixel store.

In addition to rendering volumetric geometry, the TMRendAction can render arbitrary polygonal geometry with the shape's volume texture applied to it. This is accomplished by using the `vzPolyGeometry` class, derived from `vzGeometry`. The class provides a virtual draw method, which the derived class can override. This draw method is invoked by the render action after enabling the OpenGL state for the shape's appearance, which the application can then use to render arbitrary polygonal geometry. This method can be used, for example, to implement the spherical sampling technique described earlier.

Understanding the texture management can help you improve the performance of the rendering by the render action in many common cases. `TMRenderAction` computes the total amount of resources required to render the given set of managed shapes in the `beginDraw` method and compares it to the amount available on the

graphics pipe. Depending on the outcome of the comparison, the render action uses different texture management schemes. One optimization common to all the schemes is that the render action tries to reuse OpenGL texture objects whenever possible. Consider the sequence of frames in Figure 9.

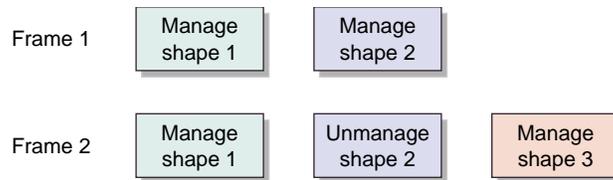


Figure 9 Shapes Managed and Unmanaged in a Sequence of Two Frames

In the first frame, the render action would allocate OpenGL texture objects for shape 1 and shape 2. In the second frame, even though shape 2 is not managed, the render action does not delete the texture objects for it. Instead, it reuses the texture objects for downloading and binding the textures in shape 3. This scheme has two advantages. First, reusing texture objects prevents fragmentation of texture memory, since not all texture managers do garbage collection immediately after the texture object has been deleted. Also, for downloading the textures in shape 3, the render action uses `glTexSubImage3D` calls, which are considerably faster than the corresponding `glTexImage3D` calls.

The preceding discussion assumes that the textures in the shapes fit in texture memory and have the same data region of interest [ROI] and internal texture formats. Hence, if your application uses multiple shapes and needs to constantly manage and unmanage them in order to improve the download performance of your application, you should try to divide the whole scene into multiple shapes such that the textures in the shapes are all of equal sizes. Typical examples of such applications are volume roaming, multi-resolution volume rendering, and time-varying volumes.

The sampling rate used to polygonize the volumetric geometry controls the number of slices that are used to render the shape. Theoretically, the minimum data-slice spacing is computed by

finding the longest ray cast through the volume in the view direction, then finding the highest frequency component of the texel values and using twice that number for the minimum number of data slices for that view direction. Practically, the rendering process tends to be pixel-fill limited and, in many cases, choosing the number of data slices to be equal to the volume's dimensions, measured in texels, works well. An application can differentiate itself by trading off performance and image quality.

Integration with Other Toolkits

OpenGL Volumizer is an API designed to handle the volume rendering aspect of an application. You can use other toolkits, such as OpenGL Performer and Open Inventor™, to structure the other elements of your application. The API allows seamless integration with other scene-graph-based APIs, because the shape node can be used as the leaf nodes of such a scene graph. Figure 10 illustrates a hypothetical scene graph that contains polygonal data mixed with volumetric data. In this case, the `vzShape` nodes are used to represent the volumetric components of the scene, whereas the other `PolyNode` is used to represent polygonal geometry.

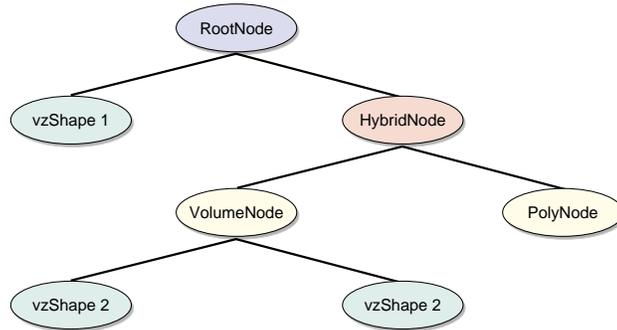


Figure 10 A Complex Scene Graph

Mixing geometric objects with volume-rendered data is a useful technique for many applications. For opaque objects, the geometry is rendered first using depth buffering, and then the volume data is rendered with depth testing enabled. When using APIs like OpenGL Performer or Open Inventor, the scene-graph traversal should be done in the appropriate order to ensure correct alpha compositing. The application can ensure this by

marking the volumetric nodes as transparent so that the scene traverser renders it after the opaque geometry. In the case of OpenGL Performer, this can be accomplished by creating the appropriate pfGeoState and attaching it to the volume node. Figure 11 shows a volumetric dataset rendered along with opaque geometry using this technique.

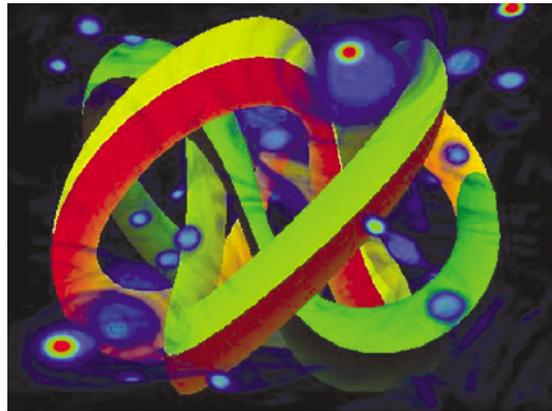


Figure 11 Volume and Opaque Geometry Integrated in a Single Scene

Using Multiple Graphics Pipes

OpenGL Volumizer objects are thread-safe. This allows applications to scale the graphics performance and other available resources by

sharing the volume data among multiple rendering threads/processes. Figure 12 shows *n* pipes rendering the same scene using one thread/process per pipe.

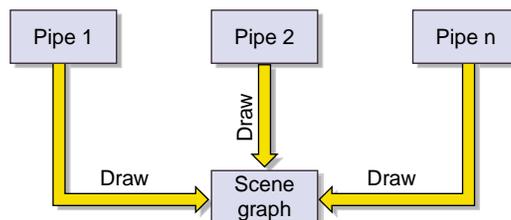


Figure 12 Multipipe Architecture

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

- Screen space [2D] decomposition, which scales the fill rate trivially and the geometry rate using view-frustum culling
- Database [DB] decomposition, which scales the fill rate, texture memory size, and geometry rate
- Time slice [DPLEX] decomposition, which linearly scales the frame rate during interaction by introducing latency
- Stereo [EYE] decomposition, which scales the frame rate while in stereo mode

- Multi-level decomposition, which mixes the above decomposition schemes using a hierarchical composition network

OpenGL Multipipe SDK provides runtime configurability and scalability to an application. Figure 13 shows an example of database decomposition across four graphics pipes. The 70 MB head dataset 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.

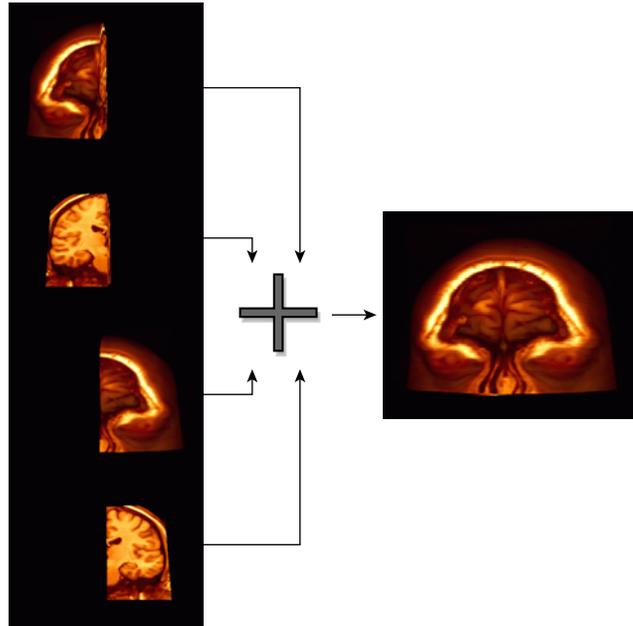


Figure 13 Database Decomposition of Volume Data

Visualizing Large Data

As the power of computing platforms or acquisition-device capabilities increases, applications using numeric simulations or data-acquisition techniques give more and more data. Some examples of these applications are in the scientific and energy domain. In this case, “large data” refers to data larger than what the

local resources can handle. This data-resource constraint means that the data to be visualized will reside on slower and larger storage peripherals, such as main memory, disks, or others instead of on local graphics resources. This data will have to migrate from one peripheral to others within the frame rate constraint. From this point of view, data migration becomes the main bottleneck for visualization.

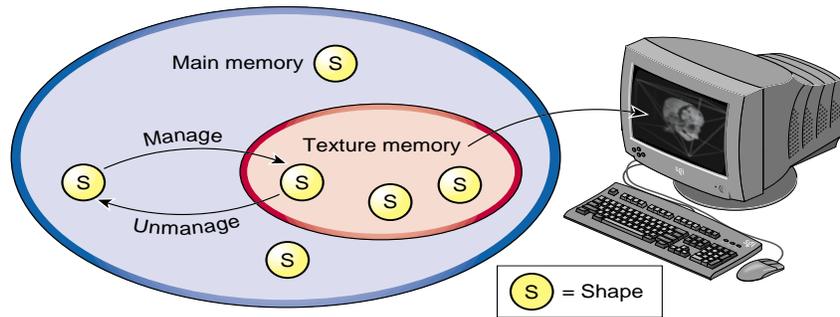


Figure 14 Large Data and Resource Management Across Multiple Devices

To handle these issues, OpenGL Volumizer benefits from the SGI® Onyx® 3000 series architecture by exploiting the high bandwidths and low latencies of such systems. The data transfer process is supported by dividing the whole volumetric data into smaller components called bricks. In this context, a brick represents one volume shape. The application controls the frame rate by moving the data bricks to the local texture memory from the various storage devices. This control gives applications the capability to visualize huge data located in memory or on high-performance disks by paging them into texture memory using intelligent schemes. In addition, TMRendAction automatically bricks textures that are too big to fit in texture memory, allowing them to be rendered using OpenGL. That is, TMRendAction handles all texture-memory-management processes by hiding all hardware-specific details, and therefore making this task transparent to the application. The following sections briefly describe techniques that can implement large data visualization applications for interactive rendering of data. The last section of this paper describes the 3D clip texture API, which is now built into OpenGL Volumizer.

Time-Varying Volume Rendering

Most computer simulations in the field of computational sciences produce time-varying datasets. OpenGL Volumizer renders those datasets by allowing applications to control the set of textures that can 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. The time-varying volume rendering example that ships with OpenGL Volumizer demonstrates how to render a large, time-varying dataset using asynchronous disk paging. Several techniques have been proposed to improve the visualization of such datasets. Examples of these techniques include Time Space Partitioning trees and texture compression.

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 dataset using a viewing window and enables them to concentrate on a specific section of the whole dataset.

Figure 15 illustrates the concept of volume roaming. The figure on the left shows the concept of volume roaming [figure courtesy TotalFinaElf]. The figure on the right is a snapshot of this technology as applied to a seismic data set. The key components of the technique are texture bricking, intelligent texture memory management, intelligent 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 subdivided 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 TMRendAction efficiently pages in the new data into texture memory from main memory.

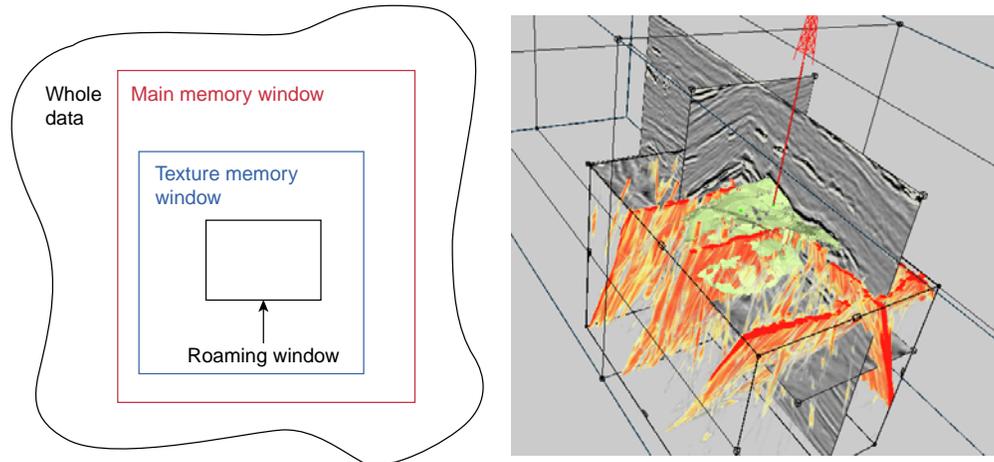


Figure 15 Volume Roaming with a 3D probe

Roaming allows an application to overcome fill rate, texture memory, and main memory size constraints, with the limitations of rendering only a subsection of the whole volume data at a time, and not providing constant frame rates during fast user motion.

data by assigning varying levels-of-detail [LOD], thus making a tradeoff between performance and image quality. Volume data is processed to compute different levels-of-resolution of the dataset by filtering and subsampling 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 the diagram on the left in Figure 16. The figure on the right shows an example of brain data rendered using the multi-resolution technique.

Multi-resolution Volume Rendering

Multi-resolution volume rendering allows applications to interactively render huge volume

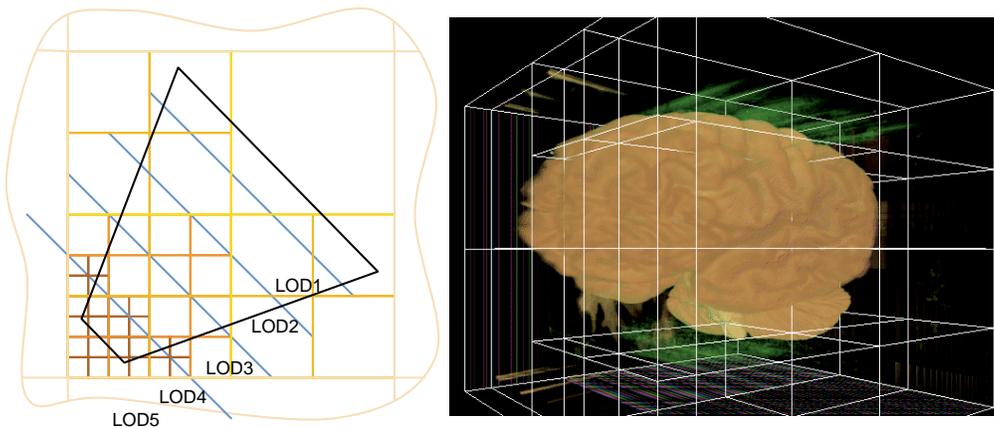


Figure 16 Brain Rendered Using Multi-Resolution Technique

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. TMRenderAction manages the texture data and multiple LUTs used to compensate for the different opacities at the LOD levels [LeMar, 1999; Weiler, 2000]. Applications can improve the performance by rendering low-resolution data at nonleaf nodes during user interaction and then improving the image quality as the interaction stops. 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 dataset that can be rendered using this technique.

3D Clip Textures

OpenGL Volumizer has built-in support for 3D clip textures, which allows applications to visualize arbitrarily large volumetric data by merging the advantages of volume roaming and multi-resolution techniques. 2D clip textures have been used successfully to provide interactive navigation of very large terrain data [Tanner, 1998]. Clip textures are MIPmapped versions of the original texture data with the exception that each MIPmap level maintains a roaming window (a physical memory window, as shown in Figure 17) to limit the amount of texture data resident in main memory. These clipped MIPmap 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.

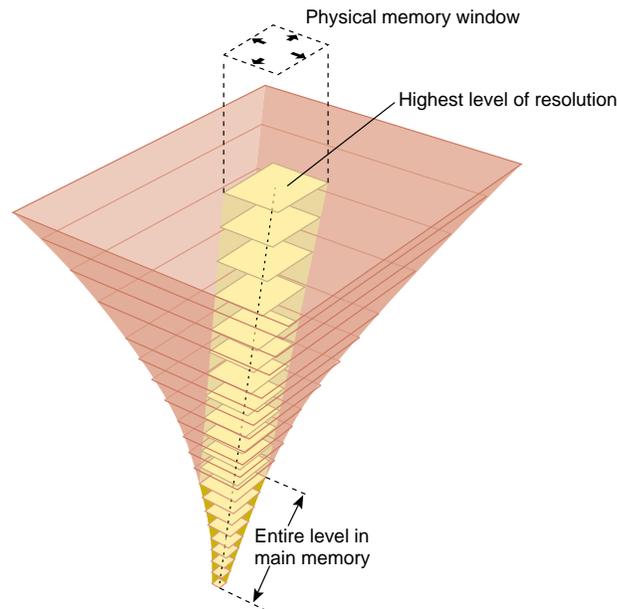


Figure 17 Clip Texture Hierarchy in 2-dimensions

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 no longer in the window, 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. 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 huge amounts of texture data resident in main memory or on high-performance disks.

The core of the OpenGL Volumizer large-data API is the abstraction of a 3D clip texture and its associated render action. Special OpenGL graphics hardware, such as InfiniteReality® graphics, has built-in support for clip textures but only in 2D. The OpenGL Volumizer implements a software emulation for 3D clip textures. 3D clip texture is an extension of the 2D scheme to 3 dimensions. In this case, the data transfer process is supported by representing the whole clip texture hierarchy as a collection of smaller 3D 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 implementation of clip textures is exposed as a new parameter class, `vzParameterClipTexture`, and an associated render action, `vzClipRenderAction`. Texture data is paged into system main memory from storage devices using asynchronous disk paging, which is 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 `TMRRenderAction`.

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, which are used to compute the number of clip levels and optimize the data transfer on the underlying hardware architecture.
- Physical memory size, which is used to limit the amount of physical memory allowed to load texture data. It controls the size of the physical memory windows at each clip level.
- Data loader callback, which is 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.

The application updates the following parameters for the clip texture depending on the user interaction:

- Center-of-interest is used to update the center of the physical memory windows of the clip levels and sort the load queue.
- Roaming window size, which is 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.

The window management mechanism is implemented using a 4D toroidal mapping technique at each level in the hierarchy. As the user moves, the physical memory window is updated, and all the bricks that are no longer in the window are pushed on the load queue to be reloaded by loader threads. This toroidal mapping scheme ensures constant memory usage and exploits frame-to-frame coherence by reusing cached texture data in subsequent frames.

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 multithreading scheme is optimized to get maximum usage of the disk bandwidth that is available on the system.

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 that are closer to the viewpoint are rendered at a higher resolution than those farther away.

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

- Total texture memory, which is 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, which is 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, which is used to control the overhead incurred due to the amount of textures downloaded from main memory to texture memory during user motion.
- LOD threshold value, which is used to trade off image quality with rendering time during user interaction.

The renderer maintains a list of bricks rendered in the current frame. In subsequent frames, as the user moves only the bricks that are no longer rendered 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 Volumizer, using texture subloads 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 ROI in the

volume data and can be moved around to navigate the dataset 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 computing the differential of the current and predicted positions and downloading this difference over a sequence of multiple frames. The clip texture can also be rendered using the single-resolution mode. When using this mode, only textures at the same level of resolution are rendered. This is implemented by traversing the hierarchy and finding the set of bricks of the highest resolution possible, which can be rendered under the given resource constraints.

The clip texture API has been utilized to interactively visualize huge volumetric datasets. Figure 18 and Figure 19 show the segmented and classified version of the visible human dataset rendered interactively using the clip texture renderer on an Onyx system with InfiniteReality3™ graphics [256 MB texture memory]. The total dataset 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 19 shows the whole data rendered with volumetric lighting in multi-resolution mode. The image on the left in Figure 19 shows a zoomed-in view with transparency and color for different organs. Bricks farther away from the viewing point are rendered at a lower resolution than the closer bricks [notice the difference between the left and right hands]. The image on the right in Figure 19 shows the full resolution data rendered using the 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 versus roaming], image resolution, shading technique, and so on.

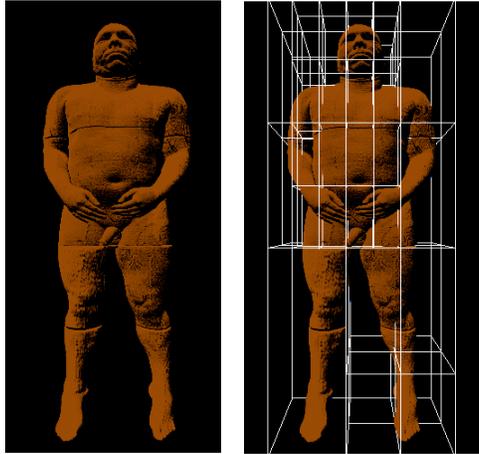


Figure 18 Whole Body with Volumetric Lighting

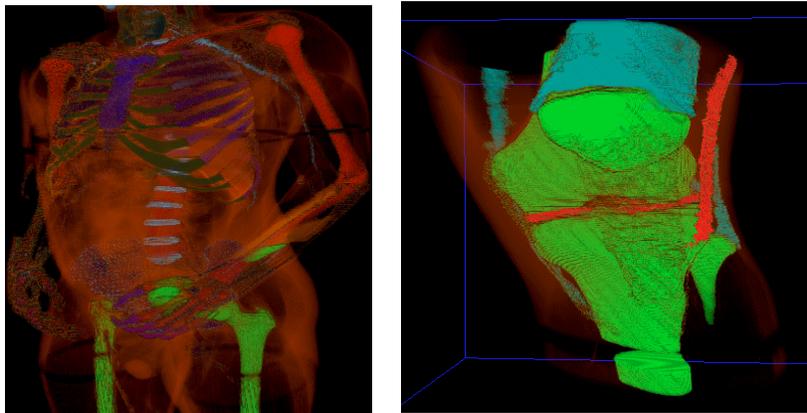


Figure 19 Left: Zoomed in to Show Transparency and Color. Right: Roaming Mode to Visualize the Left Knee at Full Resolution.

A Simple Volume Rendering Example

The following segment of code shows a simple volume rendering example using the OpenGL Volumizer API. This example creates a shape node and renders it using TMRRenderAction.

```
// Create a loader for the volume data.
IFLLoader *loader = IFLLoader::open(fileName);

// Load the volume data
vzParameterVolumeTexture *volume = loader->loadVolume();

// Create a shader for the appearance
vzShader *shader = new vzTMSimpleShader();

// Create the shape's appearance
vzAppearance *appearance = new vzAppearance(shader);

// Add the volume texture as a parameter to the appearance
appearance->setParameter("volume", volume);

// Initialize the geometry
vzGeometry *geometry = new vzBlock();

// Initialize the shape node. Gathering geometry and appearance
shape = new vzShape(geometry, appearance);

// Create a 3D-Texture-based render action
vzTMRRenderAction renderAction = new vzTMRRenderAction(0);

// Manage the shape
renderAction->manage(shape);

// Render the shape node
renderAction->beginDraw(VZ_RESTORE_GL_STATE_BIT);
renderAction->draw(shape);
renderAction->endDraw();

// Unmanage the shape
renderAction->unmanage(shape);

// Delete the render action
delete renderAction;
```

Download and Try It

The latest version of OpenGL Volumizer 2 is available free via download, providing application developers with the necessary tools for implementing interactive, scalable, high-quality, volume-visualization applications. The package can be downloaded from www.sgi.com/software/volumizer. You can find complete documentation and resources on this web page.

References

- Bhaniramka, P.; Demange, Y; OpenGL Volumizer: A Toolkit for High Quality Volume Rendering of Large Data sets. Symposium on Volume Visualization and Graphics 2002 [to be published].
- Cabral, B.; Cam, N.; and Foran, J.; "Accelerated Volume Rendering and Tomographic Reconstruction using Texture Mapping Hardware," Symposium on Volume Visualization, 1994.
- Drebin, R.A; Carpenter, L.; and Hanrahan, P.; Volume rendering. In John Dill, editor, Computer Graphics [SIGGRAPH 88 Proceedings], volume 22, pages 65--74, August 1988.
- LaMar, E; Hamann, B; and Joy, K.I.; Multiresolution Techniques for Interactive Texture-Based Volume Visualization. Proceedings of IEEE Visualization, 1999.
- Levoy, M.; Efficient ray tracing of volume data, ACM Transactions on Graphics [TOG], Volume 9 Issue 3, July 1990.
- McReynolds, T.; and Blythe, D.; Advanced Graphics Programming Techniques Using OpenGL, SIGGRAPH 98 Course Notes, Orlando, FL, 1998.
- Tanner, C.C.; Migdal, C.J.; Jones, M.T. "The Clipmap: A Virtual Mipmap." Proceedings of SIGGRAPH 1998.
- 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.
- OpenGL Volumizer 2 Programmer's Guide*, <http://www.sgi.com/software/volumizer/documents.html>
- OpenGL Volumizer 2 Reference Manual*, <http://www.sgi.com/software/volumizer/documents.html>
- OpenGL Volumizer 2 Release Notes*, <http://www.sgi.com/software/volumizer/documents.html>



Corporate Office
1600 Amphitheatre Pkwy.
Mountain View, CA 94043
[650] 960-1980
www.sgi.com

North America [800] 800-7441
Latin America [650] 933-4637
Europe [44] 118.925.75.00
Japan [81] 3.5488.1811
Asia Pacific [65] 771.0290

©2002, Silicon Graphics, Inc. All rights reserved. Silicon Graphics, SGI, IRIX, and OpenGL are registered trademarks and InfiniteReality3, OpenGL Multipipe, OpenGL Performer, OpenGL Volumizer, and Open Inventor are trademarks of Silicon Graphics, Inc.

007-4557-001