White Paper

OpenGL Volumizer™ 2.x

Experiments, simulations, and instrumentation devices continuously produce 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.x, a new application-programming interface (API) from SGI for interactive, high-quality, and scalable volume visualization.

Volume Visualization

There are a number of approaches for the 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, allowing applications to reach real-time performance and making this 3D texture-based approach the method of choice for interactive and immersive volume-visualization applications. The 3D-texture approach described here is equivalent to ray casting [Hall, 1991] 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.

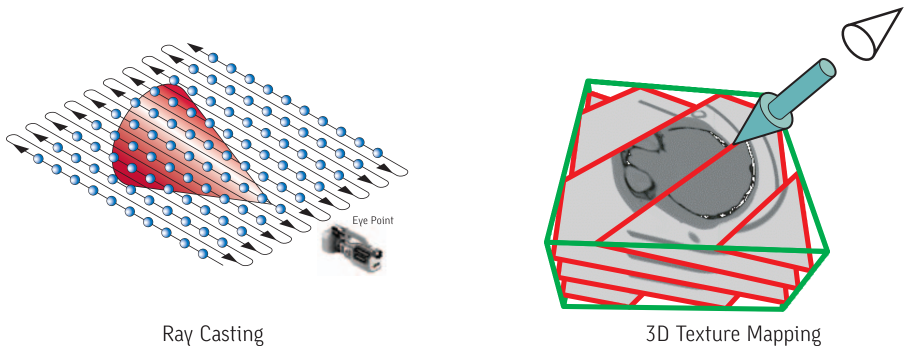


Fig. 1. Ray Casting vs. 3D Texture Mapping

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.x should be distinguished from its predecessor, OpenGL Volumizer 1.x. With the same objectives as OpenGL Volumizer 1.x, OpenGL Volumizer 2.x 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 data sets. In this white paper, we address the characteristics and features of OpenGL Volumizer 2.x, which we refer to simply as OpenGL Volumizer.

Product Overview

Announced during SIGGRAPH 2001, OpenGL Volumizer considerably simplifies the programming model while offering new capabilities and features, making visualization development of extremely large volumetric data sets on multipipe platforms easier. It provides:

- 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 (see page 2).
- Thread safety, which allows implementation of multi-threaded applications that run on multiple processors and graphics engines in conjunction with APIs like OpenGL Multipipe™ SDK.

- Integrated shading capabilities to perform volumetric lighting to improve realism and provide depth cues. The API provides support for gradient-based and gradientless rendering algorithms.
- Large data-management capabilities, including data-management mechanisms for data paging and graphics-resource control.
- Examples that include a transfer-function editor, data loaders, and a volume-rendering application for multi-pipe 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 architecture of the API.

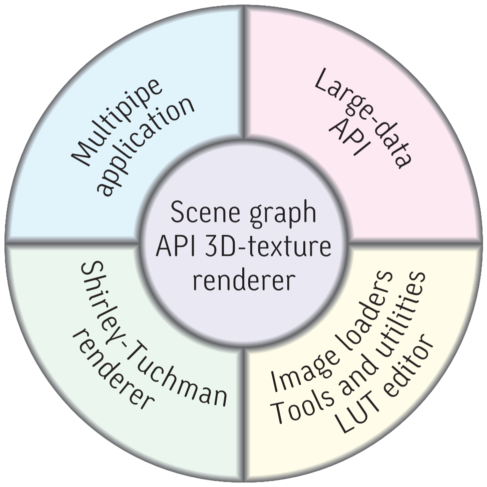

Multipipe application
Large-data API
Scene graph API 3D-texture renderer
Shirley-Tuchman renderer
Image loaders Tools and utilities LUT editor

Fig. 2. Layered architecture of OpenGL Volumizer; all modules with the exception of the large-data API and the Shirley-Tuchman renderer are included with the OpenGL Volumizer 2.0 distribution

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 that is a container for its geometry and appearance. The volume's geometry defines the spatial attributes and a region of interest, while the volume's appearance defines the visual attributes like 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 like light direction or lookup tables, if needed. Figure 3 shows a sample shape node with the corresponding geometry and appearance.

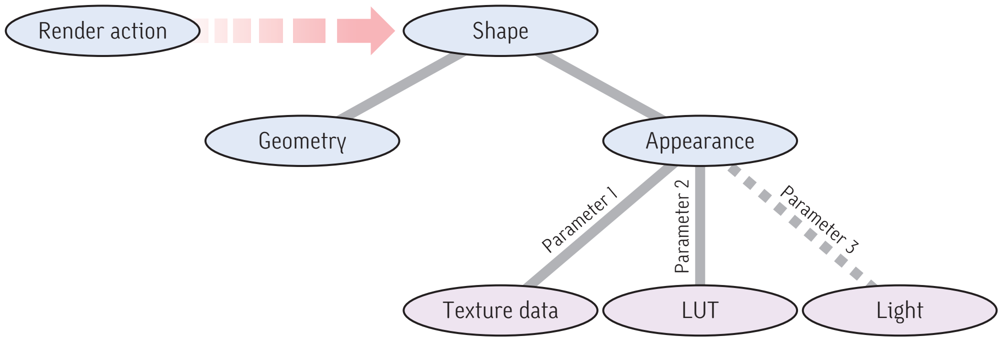


Fig. 3. Shape node and its associated render action

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 shape nodes. The texture mapping render action 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 postinterpolation lookup table (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/MP safe. This allows them to be shared across multiple threads/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. This can be used for allocating memory from shared memory arenas, which is essential for integration with APIs using a multiprocessed model of execution, such as OpenGL Performer™.

Volumetric Geometry

In OpenGL Volumizer, the region of interest 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.

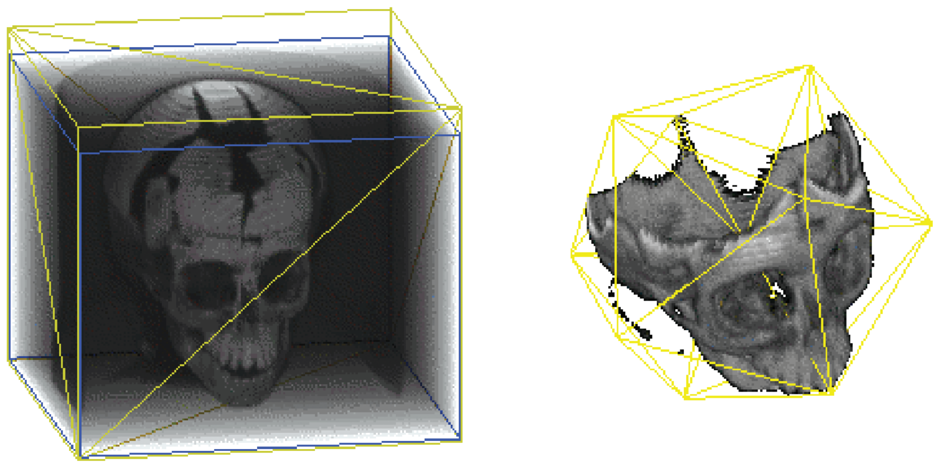
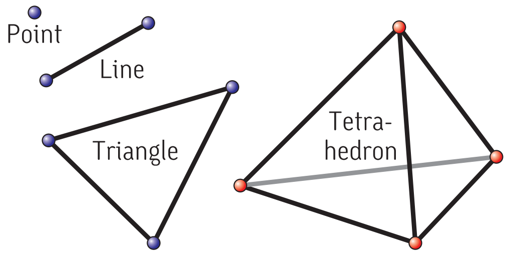
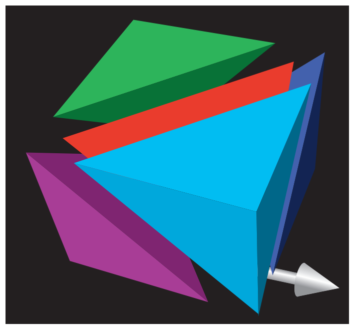


Fig. 4. Different volumetric geometries for a volume data set

OpenGL Volumizer allows specifying arbitrary volumetric geometry using simple primitives ranging from axis-aligned cubes to arbitrary tetrahedral meshes. 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.



Tetrahedron as the basic 3D primitive



Tessellation of a cube into five tetrahedra

Fig. 5. Geometric primitives

Using geometrical techniques to render volume data sets gives the 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-accelerated hidden removal technique will ensure 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-Mapping Render Action

The 3D-texture-based renderer [TMRenderAction] delivered with OpenGL Volumizer implements the semitransparent plane-rendering technique described earlier. 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.

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 pixel is successively drawn and blended into the frame buffer 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 trade-off between image quality and performance in the section titled "Understanding the Texture-Mapping Render Action."

Slicing with planes is common but artifacts can appear when the observer is very close to the model. As an implementation alternative, spherical slicing provides a more accurate visualization in perspective projection [McReynolds, 1998]. The principle is illustrated in figure 7.

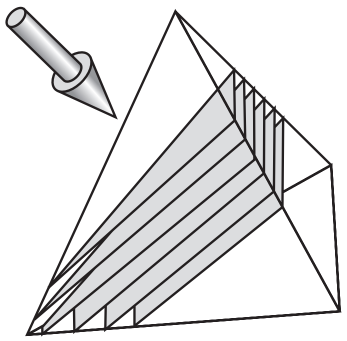


Fig. 6. Tetrahedral slicing

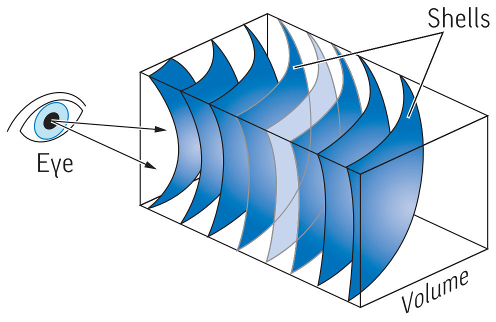


Fig. 7. Spherical slicing

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 the TMRenderAction include:

- 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
- Built-in support for multipass shading techniques like volumetric lighting and tagging
- Transparent bricking and interleaving of texture data
- Support for applications using multiresolution and volume roaming techniques

Volumetric Shaders

In OpenGL Volumizer, we introduce the concept of volumetric shaders to apply specific rendering techniques to generate desired visual effects using the same rendering algorithm described above. Each shader implements a particular technique by setting the appropriate OpenGL state and using multiple rendering passes if necessary. The TMRenderAction supports multiple built-in shaders that accept parameters for the particular technique being applied. Figure 8 shows the results generated from three different shaders applied to the same medical data set.

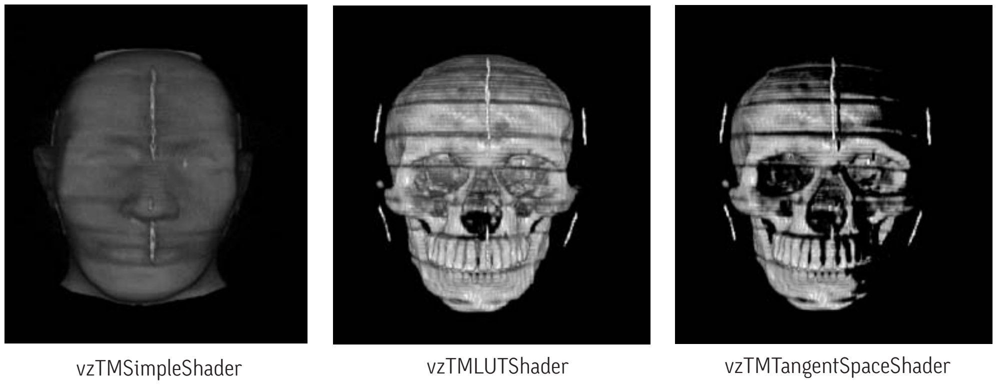


Fig. 8. Shading examples

Transfer Functions

For effective visualization of the data sets, 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 lookup tables supported in the graphics pipeline. Different alpha values in volumetric data often correspond to different materials in the volume 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 postinterpolation lookup-table parameter, mapping color and opacity values after texture interpolation. To edit transfer functions, a simple lookup-table 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 application writers can use to their advantage. Figure 9 shows the pipeline used by a typical volume rendering application using the TMRenderAction.

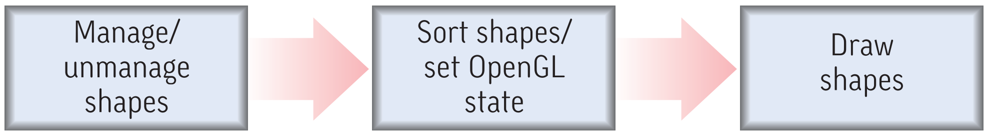


Fig. 9. Pipeline used by a volume-rendering application using the TMRenderAction

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, etc. Once the application is done `manage`'ing and `unmanage`'ing the shapes for the current frame, it is ready to draw them.

The TMRenderAction 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 for performing volume rendering, such as enabling blending and setting the appropriate blending functions. The TMRenderAction 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].

Once the above is done, the application lets the render action know that it is ready to start drawing the shapes by calling `beginDraw`. The `beginDraw` method marks the end of the texture-management phase and the beginning of the rendering phase. Inside the method, the render action:
- Computes the total resources required for the list of managed shapes
- Performs the OpenGL state management (push application's OpenGL state, store transformation matrices, etc.)
- Performs the OpenGL resource management (creates and downloads texture objects, lookup tables, etc.)

Then, the application draws all of the shapes in the visibility-sorted order computed above. Inside each `draw` method, the render action:
- Invokes the shader's initialization routine, which sets the appropriate OpenGL state (bind texture objects, enable lookup tables, etc.)
- 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`. These matrices are stored and used for all the subsequent `draws` before the next `endDraw`. Finally, in the `endDraw`, the render action restores all of the OpenGL states that it modified, including texture-related settings, lookup tables, and pixel store.

Understanding the OpenGL state management in the TMRenderAction can be used to implement alternative functionality not supported by the render action. For example, application developers can render arbitrary polygonal geometry with the shape's volume texture applied to it. Since the `draw` method does not restore any OpenGL state, if the previously rendered shape's appearance used the `vzTMSimpleShader` or `vzTMLUTShader`, the corresponding volume texture will still be bound along with the appropriate `texgen` settings. Also, applications can implement the spherical sampling technique described earlier by rendering the appropriate tessellated shells after the corresponding draw. This method necessitates caveats. For example, the technique would not work correctly with multipass shaders like `vzTMTangentSpaceShader` and shapes that have been bricked internally by the render action (if their textures do not fit in texture memory). You can make sure that the render action does not draw any polygons either by setting the volumetric geometry to be degenerate or by using slicing planes with all the planes disabled.

Understanding the texture management can help you improve the performance of the rendering by the render action in many common cases. The TMRenderAction computes the total amount of resources required to render the given set of managed shapes in the `beginDraw` 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 10.

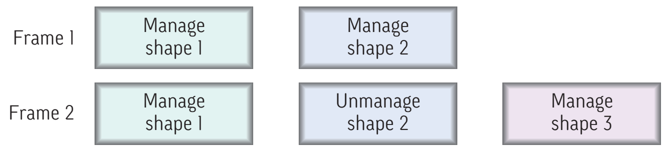


Fig. 10. Shapes managed and unmanaged in a sequence of two frames

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 above discussion assumes that the textures in the shapes fit in texture memory and have the same data 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, multiresolution 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, since the shape node can be used as the leaf nodes of such a scene graph. Figure 11 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.

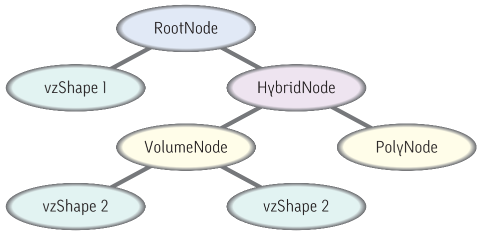


Fig. 11. A complex scene graph

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 12 shows a volumetric data set rendered along with opaque geometry using this technique.

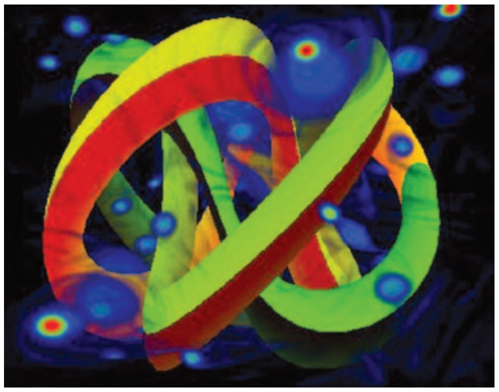


Fig. 12. Volume and opaque geometry integrated in a single scene

Using Multiple Graphics Pipes

Thread safety allows applications the ability to run on large platforms for large immersive displays or scale the graphics performance and resources use by sharing the scene graph among multiple rendering threads/processes. Typically used with the OpenGL Multipipe SDK, the application will be scalable and able to run in an SGI® Reality Center™ environment. Applications can scale the rendering performance of the system by compositing the intermediate results from different pipes to get the final image. Figure 13 shows *n* pipes rendering the same scene using one thread/process per pipe.

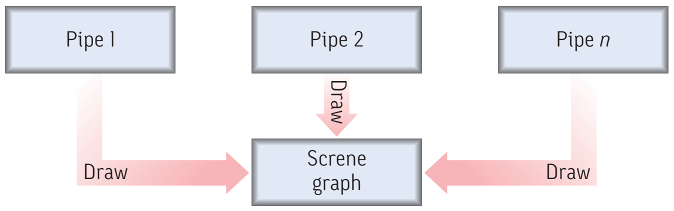


Fig. 13. Multipipe architecture

Rendering performance can be scaled using multiple compositing schemes. Figure 14 shows an example of DPLEX decomposition, where consecutive frames are rendered over different pipes. This example shows a sequence of frames as the user modifies the transfer function for this seismic data set. The even frames are rendered on pipe 1 (red) and the odd frames on pipe 2 (blue). This technique effectively doubles the frame rate with minimal application effort.

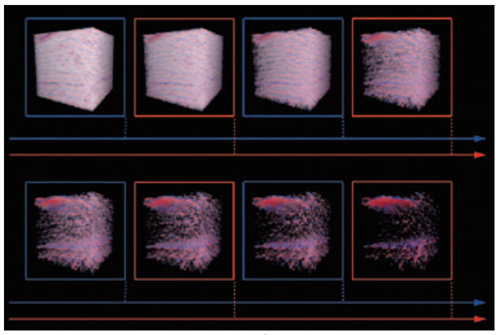


Fig. 14. DPLEX decomposition

Visualizing Large Data

As the power of computing platforms or acquisition-device capabilities increase, 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. Here, by large data we mean 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 like 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.

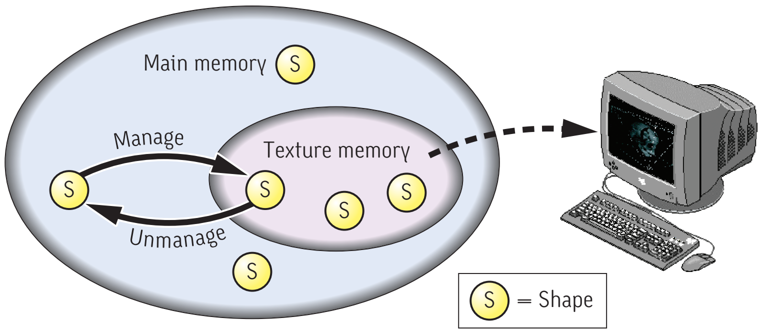


Fig. 15. Large data and resource management across multiple devices

To handle these issues, OpenGL Volumizer can benefit from the SGI® Onyx® 3000 series architecture by exploiting the high bandwidths and low latencies of such systems. The data transfer process can be supported by dividing the whole volumetric data into smaller components called bricks, not to be confused with the various hardware bricks that comprise an SGI Onyx 3000 series system. In this context, a brick represents one volume shape. The application controls the frame rate by moving these 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, the TMRenderAction automatically bricks textures too big to fit in texture memory, allowing them to be rendered using OpenGL. That is, the TMRenderAction handles all texture-memory-management processes hiding all hardware-specific details and therefore making this task transparent to the application. The following section briefly mentions two techniques that can be used by large data visualization applications for interactive rendering of the data.

Volume Roaming

Volume roaming is an efficiency technique that allows the user to explore large volumetric data using a volumetric probe, which can be interactively moved inside the volume. The probe allows the user to have a viewing window and helps the user concentrate on a specific section of the whole data set. Such a technique can achieve improved performance by using:

- Intelligent texture-management techniques that use predictive texture downloads to maintain near-constant frame rates during user motion
- Intelligent memory-management techniques that allow roaming through a data set that might not even fit in main memory
- The concepts of toroidal mapping from clip textures for volume roaming; the granularity of the texture element is a volume brick here rather than a texel

Multiresolution Volume Rendering

Multiresolution volume rendering allows applications to interactively render huge volume data by assigning varying levels of detail (LOD), thus making a trade-off between performance and image quality [LaMar, 1999; Weiler, 2000]. Lower resolutions help improve performance, since they limit the texture memory and fill-rate consumption of the application. Many researchers have worked on multiresolution techniques for interactive volume rendering, typically using an octree decomposition of the whole volume. The following techniques can be used to improve the performance while maintaining acceptable image quality:

- Coupling texture management with LOD switching in order to ensure near-constant frame rates
- Using the sorted order of texture bricks to determine the LOD to be rendered
- Using clipping geometries to optimize the use of texture memory available on the graphics subsystem
- Rendering higher resolutions during stages of less or no user interaction

In addition, time-varying techniques allow users to run a volume movie and can be easily implemented with the same techniques. Such techniques can be implemented for visualizing animated fluid dynamics or crash analysis data.

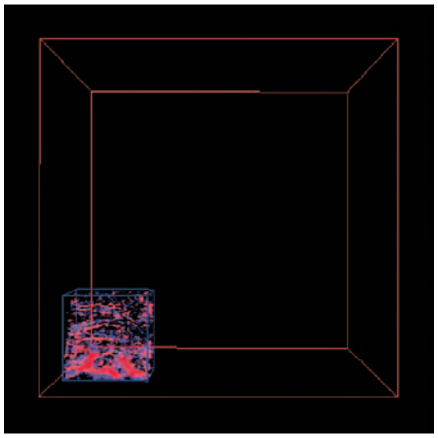


Fig. 16. Volume roaming with a 3D probe

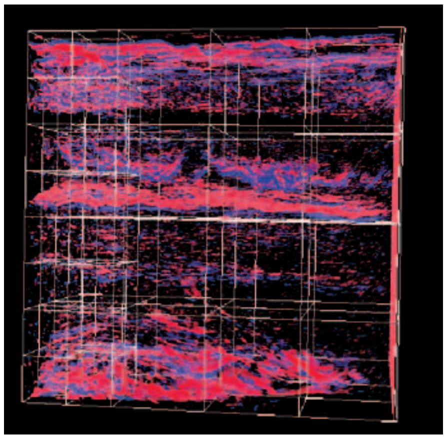


Fig. 17. Multiresolution volume rendering

A Simple Volume Rendering Example

The following example creates a shape node and renders it using the TMRenderAction.

```
// 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
vzTMRenderAction  renderAction =
new vzTMRenderAction(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.x 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*. Please find complete documentation and resources from this page.

References

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.

Hall, P.M.; and Watt, A.H.; Rapid volume rendering using a boundary-fill guided ray cast algorithm. In N. M. Patrikalakis, editor, Scientific Visualization of Physical Phenomena (Proceedings of CG International '91), pages 235-249. Springer-Verlag, 1991.

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.

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 2.1 Programmer's Guide, *http://www.sgi.com/software/volumizer/documents.html*

OpenGL Volumizer 2.1 Reference Manual, *http://www.sgi.com/software/volumizer/documents.html*

OpenGL Volumizer 2.1 Release Notes, *http://www.sgi.com/software/volumizer/documents.html*