Volume Scene Graphs

David R. Nadeau
San Diego Supercomputer Center
University of California, San Diego

Abstract

This paper discusses volume scene graphs – a flexible hierarchical structure for composing scenes containing volume data sets and space-filling functions. Scene graph nodes are functions that, when evaluated at a point in space and time, compute and return a value. Typical nodes return values sampled from volume data sets, compute values using procedural texture algorithms, or filter and composite values returned by one or more other scene graph functions. Voxelization of a scene graph repeatedly evaluates the graph’s functions over a gridded region of space. Examples are shown that compose scenes containing multiple volume data sets of differing resolutions and modalities.

CR Categories and Subject Descriptors: I.3.3 [Computer Graphics]: Computational Geometry and Object Modeling – Hierarchy and geometric transforms; Object hierarchies; Modeling packages.

Additional Keywords: scene graphs, volume graphics, volume visualization.

1. INTRODUCTION

A goal of visualization is to help individuals gain insight into real-world or computational processes. For processes that are complex or time-varying, the data collected about them may be contained in multiple data sets. CT, MRI, and PET scans of the same individual, for instance, yield separate overlapping volume data sets of differing modality and resolution. Numerical simulations in science and engineering often produce overlapping volume data sets for different time steps, simulation parameters, regions of interest, or resolutions. To gain insight into these multi-data set scenarios, visualization software must support easily overlapping volume data sets in space and time, coregistered, and appropriately combined to yield a composite result.

This paper introduces volume scene graphs – a flexible volumetric scene composition approach that enables data sets of differing resolutions and modalities to be overlaid, filtered, segmented, cut, and composited.

1.1. Scene graphs

Generically, a scene graph is a hierarchical organization of shapes, groups of shapes, and groups of groups that collectively define the content of a scene. Shapes and subtrees may be shared among multiple groups, creating a directed acyclic graph. Scene graphs are widely used to define complex 3D scenes with hundreds or thousands of separate shapes. APIs and file formats, such as PHIGS [11], Open Inventor [34], IRIS Performer [27], Java 3D [29], and VRML [12], all support the creation and rendering of scene graphs.

Traditionally, scene graphs contain shapes defined by surfaces, such as sets of polygons. Volume scene graphs introduced here support volumetric shapes defined procedurally or by data sets read from disk. Compositing operators in the graph specify the treatment of overlapping volumes. Transform operators translate, rotate, and scale shapes or subtrees to construct a scene. Additional operators support data filtering, classification, selection, masking, cutting, and so forth.

For example, Figure 1.1 shows a volumetric scene composed of three overlapping data sets from the head of the Visible Human Male [6]. The data sets include a CT volume at 512x512x230 resolution, and cryosection and segmentation volumes at 547x710x672 resolution. Each volume is a shape within the scene, transformed to register it into a common coordinate space.

Figure 1.1. A volumetric scene composed of three overlapping volume data sets at differing resolutions.

In Figure 1.1, CT data is classified to extract the skull, then translated and scaled to register it with cryosection data. Box-shaped cutting volumes cut the skull in half, and a transform rotates one half away. A segmentation volume is masked with cryosection data to extract the brain, optic nerves, and eyes. The results are volumetrically composited into a final rendered scene.

Volume scene graphs provide a natural way of building scenes containing multiple data sets. Each shape is a data set that is
filtered, classified, cut, transformed, and composited by operators higher in the graph.

Figure 1.2 shows a more complex example that visualizes the central region of the Orion nebula [19]. This scene uses procedural volume data that generates a glowing turbulent “fuzz” atop a polygonal terrain derived from analysis of nebula imagery [37]. Operators colorize the volume using high-resolution Hubble imagery. Proto-solar systems and shock fronts within the nebula are similarly built and composited together to form the full scene containing 2,112 scene graph nodes.

The Orion nebula visualization was developed with the Hayden Planetarium at the American Museum of Natural History, New York City. A fly-through of the nebula is featured in the planetarium’s daily show.

2. PRIOR WORK

Prior work in scene composition can be arranged in a spectrum from work that treats scenes as data, to work that treats scenes as functions over space and time. On the data side, scenes are composed of one or more data sets, such as MRI scans or polygons. On the function side, scenes are procedurally defined, such as with procedural textures.

Traditional scene graphs are on the data side, acting largely as a structure for organizing data. Nodes in these scene graphs hold data and rendering parameters but typically do not include procedural features. This reflects a necessary bias to constrain scene content to data that can be quickly pushed into a hardware graphics pipeline.

Much of prior work on volume visualization also has been on the data side of the scene composition spectrum. This work typically involves rendering a single volume data set read from disk. It is only when multiple volumes are needed in the same scene that a functional scene notion must be introduced to describe how overlapping volumes are composited. Operational semantics also must be defined to decide how to combine data sets with differing resolutions and grid orientations.

Zuiderveld et al, and Chunguang et al discuss resampling of multiple volumes to a common grid prior to compositing [7,38]. Wilhelms et al discuss operational semantics for rendering multiple intersecting regular grids [36] and Cai and Sakas discuss data intermixing (compositing) for overlapping volumes [4]. In these cases, there is no explicit description of a scene, only an implicit description resulting from a chain of independent operations.

Constructive Solid Geometry (CSG) introduces an explicit scene description expressed as a binary tree containing solid primitive shapes (box, sphere, etc.) at the leaves, and set operators (union, intersect, subtract) for interior nodes [26]. Breen et al discuss voxelization of CSG trees [2,3]. The procedurally defined primitives and set operators of CSG trees place them on the functional side of the scene description spectrum. Chen et al have extended CSG trees to Constructive Volume Geometry (CVG) that incorporates volume data sets as CSG primitives [6].

Mid-way between data-oriented polygonal scene graphs, and procedural CSG trees are shade trees, introduced as an architecture supporting customizable shading models [8]. This work later became RenderMan [31]. This approach supports scenes composed of surface data, as well as procedurally defined shaders, including those supporting 3D procedural textures (a.k.a., hyper-textures) [14,24,28].

Because the operational semantics of rendering surfaces differ from those to render volumes, much work has been done to convert either data type to the other. Numerous methods have been discussed to voxelize surfaces to turn them into volume data sets [2,5,10,13,15,16,22,30,32,33]. Multiple techniques also exist to express a volume as point clouds and 3D texture-mapped polygons suitable for rendering using hardware graphics pipelines.

The volume scene graphs discussed here are mid-way between data and functional scene descriptions. They are an explicit scene description reminiscent of CSG trees and related to the independently developed CVG trees [6]. Like CSG trees, leaves define shapes while interior nodes define compositing operators. Volume scene graphs extend CSG trees to support leaves containing volume data set primitives, and to support interior nodes that perform arbitrary filtering operations. Operational semantics also are defined to support sampling of the scene graph in a manner that is independent of the resolution and grid orientation of component data sets.

Because of the strong functional component of volume scene graphs, they are powerful, but not directly renderable by graphics hardware. A volume scene’s functions must be executed first to generate renderable voxels. Volume scene graphs may be thought of as a high-level scene description that, after evaluation, yields low-level renderable quantities.

Section 3 introduces basic features of volume scene graphs, Section 4 extends these features, and Section 5 discusses the operational semantics and implementation.

3. VOLUME SCENE GRAPHS

In a volume scene graph, each node is a function, \( f \), that may operate upon a set of arguments and a set of child nodes. Those children are themselves functions that may have arguments and more children. Evaluation of a function passes in an argument giving the floating-point \((x,y,z,t)\) location of a point in space and time. The function is responsible for returning a sample, \( s \), that is a value found at that point in space-time.

To describe nodes, we use a brief notation where

\[
s \leftarrow f(x,y,z,t)\]

assigns to \( s \) the result of executing function \( f \) on arguments \((x,y,z,t)\). We may define the essence of a function’s
implementation by adding indented statements ending with an \(\rightarrow\) giving the function’s return value.

For instance, the simplest scene graph node, Color, returns a constant RGB-alpha color for all \((x,y,z,t)\). This is the volumetric equivalent of a background color:

\[
\text{s} \leftarrow \text{Color}(x,y,z,t); \\
\quad \leftarrow (r,g,b,a)
\]

A Box node compares the sample location \((x,y,z,t)\) to the size and location of a 3D box. If \((x,y,z,t)\) is inside the box, it evaluates its first child and returns its value. If \((x,y,z,t)\) is outside the box, the node evaluates a second child and returns its value.

\[
\text{s} \leftarrow \text{Box}(x,y,z,t); \\
\quad \text{if } (x,y,z,t) \text{ inside,} \\
\quad \quad \leftarrow f_1(x,y,z,t) \\
\quad \text{else} \\
\quad \quad \leftarrow f_2(x,y,z,t)
\]

A Volume node is a kind of Box node. For sample locations inside the box it returns a sample from a volume data set read from disk. The floating-point \((x,y,z,t)\) location is mapped to the storage coordinate space of the data, such as a grid indexed by integer array indices \((i,j,k)\). The function returns a sample from the data set, using interpolation if appropriate.

\[
\text{s} \leftarrow \text{Volume}(x,y,z,t); \\
\quad \text{if } (x,y,z,t) \text{ inside,} \\
\quad \quad (i,j,k) \leftarrow \text{map}(x,y,z) \\
\quad \quad \leftarrow \text{dataArray}[i,j,k] \\
\quad \text{else} \\
\quad \quad \leftarrow f(x,y,z,t)
\]

A Group node has one or more children and a compositing operator. When evaluated, it evaluates its children, composites the samples they return, and returns the result:

\[
\text{s} \leftarrow \text{Group}(x,y,z,t); \\
\quad \text{s}_1 \leftarrow f_1(x,y,z,t) \\
\quad \text{s}_2 \leftarrow f_2(x,y,z,t) \ldots \\
\quad \leftarrow \text{composite}(s_1, s_2, \ldots)
\]

Compositing operators include imaging operators Over, Atop, Inside, Outside, and Xor [27], CSG set operators, Union, Intersect, and Subtract, math operators, Add, Subtract, Multiply, and Divide, etc.

A Transform node is a group that includes a 3D transform, \(T\). When evaluated, the node transforms its incoming \((x,y,z,t)\) into a new coordinate, \((x',y',z',t')\), then evaluates its children with that new coordinate, compositing and returning the results:

\[
\text{s} \leftarrow \text{Transform}(x,y,z,t); \\
\quad (x',y',z',t') \leftarrow T(x,y,z,t) \\
\quad s_1 \leftarrow f_1(x',y',z',t') \\
\quad s_2 \leftarrow f_2(x',y',z',t') \ldots \\
\quad \leftarrow \text{composite}(s_1, s_2, \ldots)
\]

These basic nodes are sufficient to begin to create interesting scene graphs. Start with a Box node with two child Color nodes, one each to define the inside and outside colors for the box. Make the outside color transparent black and the inside color white:

![Figure 3.1. A scene graph defining a volumetric box.](image1)

Replace the Box with a Volume node for a skull volume data set with an RGB-alpha value at each voxel:

![Figure 3.2. A scene graph for a volume data set.](image2)

Add a Box node big enough to span half of the data set. Make the Volume node the inside child, and set the outside child to a Color node with transparent black:

![Figure 3.3. Cutting a volume data set in half.](image3)

The Box node is a cutting volume, dividing space into inside and outside regions. The portion of the skull inside the box contributes to the scene. To include both skull halves in the scene, use a second Box to cut out the other half of the skull, a Transform to rotate that half away, and a Group to composite the halves together.

![Figure 3.4. Compositing two halves of a data set.](image4)

Transforms used in a Transform node are typically 4x4 matrices supporting translation, rotation, and scaling. However, transforms can be any coordinate space mapping functions, including those that warp space, such as the “twirl” in Figure 3.5. Warping transforms can be used to correct for distortions and coregister data sets.

![Figure 3.5. A 3D “twirl” transform to warp space.](image5)
These examples illustrate the simplicity of the volume scene graph approach. Data sets are easily cut, transformed, and composited to construct a scene.

### 3.1. Voxelizing the Scene Graph

The scene graph describes scene content, not rendering. To render the result, the scene graph may be voxelized to generate a new volume data set containing the scene. Voxelization repeatedly evaluates the scene graph, starting at its root, over a set of $(x,y,z)$ coordinates arranged in a grid. The result of each evaluation is stored as a voxel value in a new volume data set.

- $root(x_1, y_1, z_1, t)$
- $root(x_1, y_2, z_1, t)$
- $root(x_1, y_3, z_1, t)$
- $root(x_1, y_4, z_1, t)$
- ...

**Figure 3.6.** Voxelizing a scene graph on a grid.

Voxelization may occur at any resolution since the scene graph’s nodes define values at all points in space-time. Voxelizing at a low resolution gives a quick, rough result usable during construction and debugging. Voxelization at a high resolution produces higher-quality output. Once voxelized, the resulting volume data set can be rendered using a standard volume renderer.

**Figure 3.7.** The same scene graph voxelized at 32$^3$, 64$^3$, 128$^3$, and 256$^3$ voxels.

Scene graph voxelization samples the scene on a chosen grid. The size, orientation, and resolution of that grid may differ from that of any child data set.

**Figure 3.8.** The scene’s voxelization grid need not align with that of any child data set.

Voxelization of a scene graph need not occur on a rectilinear grid. For instance, we are developing a perspective ray casting volume renderer that evaluates the scene graph at each step along each ray. In perspective, the rays spread outwards in a fan and use a sample spacing that increases with distance from the eye. This creates a fan-shaped voxelization pattern.

**Figure 3.9.** Voxelizing a scene graph within the view frustum during ray casting.

Perspective voxelization at rendering time avoids creating an intermediate volume data representation of the scene - rendering is a result of sampling the scene directly. Voxelization occurs only for those portions of the scene within the view frustum. With early ray termination (the point at which pixel opacity sums to 1), voxelization only occurs for visible portions of the scene.

### 4. EXTENDING VOLUME SCENE GRAPHS

In the prior section, scene graph nodes were defined as functions that return a sample containing an RGB-alpha value. However, not all volume data sets contain voxels with RGB-alpha values. In fact, few data sets do. To support data sets of other types, scene graphs must support multiple data types.

Extending the approach, scene graph nodes are functions that return a sample value of a specific data type. The Color node, shown earlier, returns an RGB-alpha tuple. Float, Double, Integer, Short, and Long nodes return scalar values of the named type. Point and Vector nodes return 3D positions or vectors, and so forth.

Functions in the scene graph must be dynamically typed, unlike most programming languages [35]. The Group node, for instance, composites together samples returned by its children. The type of the group’s return value is that of its children (which all must have the same type). So, a Group’s type is dynamic – it changes based on use. Similarly, the Volume node’s type depends on the type of data it reads from disk. For CT data, it returns scalars, for cryosection data, it returns RGB values, etc.

In a statically typed environment, type checking can be done at compile time. With dynamic types, checking must be done at run time. A Group’s children must be known before the data type of the Group is known.

Type checking for volume scene graphs is done immediately prior to voxelization. A type-checking pass recursively traverses the graph. Type checking also checks that a group’s children are compatible with its composite operator: Over, Atop, etc, only apply to RGB-alpha tuples, while math operators apply to most types.

Voxelization of a scene graph produces samples whose data type matches that returned by the graph’s root node. While it is conventional that the root return RGB-alpha tuples suitable for rendering, scene graph voxelization can produce values of any data type.
4.1. Node Fields

Earlier we defined nodes as functions that take an \((x,y,z,t)\) location argument and return a sample. To this we add that functions also have fields that provide node-specific parameters, such as the size and location of the box for a Box node, the name of a data file for a Volume node, or the color to return from a Color node.

Fields may contain literal values, such as a color. For added power, any literal value can be replaced by a node that, when evaluated at an \((x,y,z,t)\) location, returns a value whose data type matches that of the field.

For example, similar to the Box node, the Sphere node divides space into regions inside and outside a sphere. The sphere’s radius is a field whose value may be a literal or a node that, when evaluated, returns a radius. Figure 4.1 shows a sphere whose radius has been replaced by a Turbulence node using a Perlin noise function [23,24].

![Figure 4.1. A sphere with its radius set by turbulence.](image)

4.2. A Few More Nodes and Examples

An Interpolator node returns a value computed by linear interpolation at a point along a path of values. The path’s values may be positions, colors, scalars, etc. The point on the path at which to compute a value is set by a field. If the field contains a node, the interpolator returns a value that may vary in space and time.

One use of an Interpolator node is to classify scalar data – mapping scalar values to RGB-alpha values. To classify a scalar CT volume data set, use a Volume node together with an Interpolator:

![Figure 4.2. CT data classified by an Interpolator.](image)

The Distance node computes the distance from the current sample location \((x,y,z,t)\) to a chosen point in space. Together with an Interpolator node, the Distance node can be used to create spatial color gradients:

![Figure 4.3. A spatial color gradient.](image)

Use a Multiply compositing operator to multiply a spatial color gradient times a data set to add stripes that highlight surface contour:

![Figure 4.4. Use a color gradient to highlight contour.](image)

Polygonal data may be incorporated into the scene graph using a SurfaceDistance node. The node computes a distance field that, at each point in space, returns the distance from that point to the nearest point on a polygonal surface [2,10,22]. The distance can be used with an Interpolator to vary opacity with distance from the surface. Or use Turbulence to deform the surface:

![Figure 4.5. A distance field for a smooth surface (left), and with turbulence to deform the surface (right).](image)

A Gradient node uses central differencing to compute a gradient vector that may be used for simple shading effects [18]. The Gradient node samples its single child multiple times at the eight points of a 3D compass to yield a gradient vector. Convolution filters may be implemented in a similar fashion.

5. OPERATION

A volume scene graph is, essentially, a composition of functions, each defined over all space and time. Voxelize or rendering of the scene requires evaluation of those functions at chosen points in space-time. Each evaluation delivers a single sample value. So, to reconstruct, say, a volumetric teapot, the scene graph containing the teapot must be repeatedly evaluated across a grid, voxelizing the space whose content is defined by the scene graph’s functions.

Evaluation of a volume scene graph at a point in space-time does a recursive traversal of the graph. At each node, the node’s doSample method is invoked. That method can do anything, as long as it returns a new sample. Typical nodes get a few values from their fields, combine those values, and return a computed sample. Getting a value from a field may, itself, require recursion since the field’s value may be supplied by yet another node.

Recursion to walk the graph incurs a small per-node method-call overhead that is generally dwarfed by the cost of executing the statements within a node’s doSample method. For instance, a tri-linear interpolation to sample a volume data set is far more expensive than a method call. So, the computation cost of evaluating the scene graph is dominated by the cost of the interpolation, filtering, and compositing being done, and not by the scene graph infrastructure that orchestrates those operations.

Volume scene graphs are implemented as a general-purpose API in C++. The implementation currently supports 40+ different
node types, an interactive scene graph editor, and a text file format that encodes the scene graph structure using a VRML-like syntax.

Volume scene graphs have been used to compose volumetric scenes containing multi-modal medical data. Using scene graph cutting volumes, medical data was cut apart to produce volumetric pieces that were then constructed as physical models using freeform fabrication equipment [20].

Volume scene graphs were recently used to develop a visualization of the Orion nebula, in collaboration with the Hayden Planetarium at the American Museum of Natural History in New York City [19]. The Orion visualization started with a surface model of the nebula’s ionization layer, derived from infrared and visible light observations [37]. The surface model was incorporated into a scene graph that used a distance field to vary opacity and emissivity with distance within the nebula’s ionization layer. To give the layer a rougher, more turbulent look like that observed in nebula imagery, the distance field was perturbed by procedural turbulence. A high-resolution Hubble image of the nebula was projected through the scene to colorize the volume. The resulting scene has 2,112 nodes. Figures 1.2 and 5.1 show the nebula.

Figure 5.1. A close-up of the nebula interior.

While the full scene graph contained 2000+ nodes, the actual render job was performed by manually cutting the scene graph into 86 separate scene sub-trees. One sub-tree described the main ionization layer, while 85 sub-trees described very small-scale shock fronts and proto-solar systems within the nebula’s interior. Each of these sub-trees were voxelized separately, then rendered simultaneously using a custom multi-volume renderer.

Direct rendering from the scene graph, without voxelization, is a project currently in progress. Preliminary results yield high-quality images and a lower memory footprint than when using voxelized data. However, scene graph rendering can be slower than volume rendering – it requires repeated scene graph evaluation at rendering time. When rendering a fly-through animation of a static scene (such as the Orion nebula), the data doesn’t change from frame to frame. Re-evaluation of the scene graph during rendering is therefore inefficient. In such cases, it is faster to voxelize the scene graph once, then render those voxels over and over. The performance improvement this yields is significant and is why the Orion sequence was voxelized then rendered, instead of being rendered directly.

The final 2 ½ minute fly-through animation of the nebula was produced for the Hayden Planetarium’s daily show. The planetarium uses seven 1280x1024 video projectors to seamlessly cover the interior of the planetarium’s dome, so animation production computed seven images for each frame of the show. The 2 ½ minute animation required about 31,000 images.

6. FUTURE WORK

The volume scene graph’s nested functions resemble a programming language expression. By viewing a scene graph as an expression, we link it to a body of knowledge on expression optimization [1]. Similarly, a scene graph resembles a database query tree, linking scene graphs to a body of knowledge on query optimization [9]. In either case, a scene graph is a human notation describing a scene. It is not necessarily the data structure used for efficient evaluation of the scene.

For example, consider a scene graph that uses a Volume node to read in a data set. A Box node selects a subset of the volume and a chain of filtering nodes process the data to produce a final result. Figure 6.1’s left-hand diagram shows this scene graph.

![Image of scene graphs](image)

Figure 6.1. An unoptimized (left) and optimized (right) scene graph using selection and filtering.

The left-hand scene graph of Figure 6.1 is not optimal. To see why, recall that a Box performs a spatial selection – the “inside” child is evaluated only for sample locations inside the box. In the left-hand graph, the Box selects only a portion of a data set, but filtering operators higher in the graph are performed for all points in space, inside or outside the box. If regions outside the box are empty space, then filtering such values is a waste of time (for most filters). A more optimal scene graph, like that on the right in Figure 6.1, moves the Box high in the graph so that filtering only occurs for points inside the box.

Figure 6.1 illustrates a scene graph transformation rule equivalent to the database rule that:

\[
(A \text{ JOIN } B) \text{ WHERE restriction-on-B}
\]

can be transformed into the more efficient:

\[
(A \text{ JOIN } (B \text{ WHERE restriction-on-B}))
\]

In other words, for better efficiency, use selection operators, like Box, as early as possible.

In our current work, we are investigating rules that reorder a graph prior to evaluation. Transformation rules that move selection nodes upwards, also may propagate information downwards into leaf nodes, like Volume, that read data from disk. With selection knowledge, a Volume node can reduce the amount of data it reads into memory.

Transformation rules also may propagate certain filtering and selection operations downwards into intelligent I/O subsystems, such as the Active Data Repository (ADR) [17]. These subsystems support spatial selection directly and apply filtering operations on-the-fly as data is streamed from disk.
7. CONCLUSIONS

This paper has introduced volume scene graphs—a flexible hierarchical structure for composing scenes containing volume datasets and space-filling functions. The approach is modeled loosely after existing surface-based scene graphs, but adds a procedural innovation that incorporates classification, filtering, compositing, and procedural texture features into the same structure.

Evaluation of a scene graph, from root to leaf, computes a single sample value at a point in space and time. Repeated evaluation for sample points arranged on a grid voxelizes the scene graph and generates a renderable volume data set representation of the scene. Alternate voxelization patterns are also possible, such as a fan-shaped pattern used by a perspective ray caster.

Acknowledgements

The semantics of scene graphs were developed through collaboration with Jon Genetti and John Moreland. Jon Genetti developed the VISTA volume renderer used to render the Orion Nebula and the examples in this paper.

This work was done at the San Diego Supercomputer Center, at the University of California, San Diego. It was funded, in part, by the American Museum of Natural History and its Digital Galaxy Project. Additional funding was provided by the National Science Foundation. The software is part of a larger suite of scalable volume visualization tools being developed under the NSF-funded National Partnership for Advanced Computational Infrastructure (NPACI). The scalable tools project is led by B. A. Pailthorpe and A. Olson.

References


