A Volume Rendering Extension for the OpenSG Scene Graph API

Thomas Klein Manfred Weiler Thomas Ertl

Institute of Visualization and Interactive Systems, University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany; E-mail: {klein, weiler, ertl}@vis.uni-stuttgart.de

Abstract

We will present the current state of our ongoing work on a simple to use, extensible and cross-platform volume rendering library. Primary target of our framework is interactive scientific visualization, but volumetric effects are also desirable in other fields of computer graphics, e.g. virtual reality applications. The framework we present is based on texture-based direct volume rendering. We apply the concept of volume shaders and demonstrate their usefulness in terms of flexibility, extensibility and adaption to new or different graphics hardware. Our framework is based on the OpenSG scene graph API, that is designed especially with multi-threading and cluster-rendering in mind, thus, it is very easy to integrate volumetric visualizations into powerful virtual reality systems.

Keywords: texture-based direct volume rendering, scene graph API, virtual reality

1 Introduction

Visualization of volumetric data is of utmost interest not only in the field of scientific visualization but also in other areas of computer graphics, like virtual reality or computer animation. Although there are already some scene graph APIs available that support volumetric objects [1, 4] their solutions mostly lack the possibility for easy development of platform-independent and extensible applications. Our extension to the OpenSG scene graph library [2] is especially focused on these issues. The major design goals were: platform independence, extensibility, flexibility, usability, and seamless integration into the existing scene graph system.

Our work is closely related to SGI's OpenGL Volumizer [3]. However our implementation is not limited to SGI hardware only, especially as with the OpenGL Volumizer 2.x releases the support for graphics systems other than InfiniteReality was canceled by SGI. Instead we support a wide range of platforms and graphics adapters from low-cost PC hardware like the NVIDIA GeForce to high-end visualization systems as the SGI Onyx family.

The framework we present is built upon the OpenSG scene graph API, a real-time rendering system especially designed for use in multi-threaded, multi-pipe, and cluster-rendering environments. OpenSG is a freely available open source project written in C++ utilizing OpenGL as low-level graphics library. It provides an extensive set of scene graph nodes based on multi-threading aware container classes, thus, allowing the easy development of multi-threaded virtual reality systems. Another advantage of OpenSG is its portability. It is known to work on many different platforms including Linux, Irix, Solaris, and Microsoft Windows. Our volume rendering extension is originating from the OpenSG PLUS project [2], funded by the German Ministry for Research and Education (BMBF), in which nine German research institutions (universities and independent research groups) are cooperating in the development of important basic technology for OpenSG. This includes support for very large scenes, higher level primitives like subdivision surfaces, and high-level shading on contemporary graphics hardware.

2 Implementation

The volume rendering extension we have implemented provides a special volume rendering node for the scene graph consisting of several modules, that provide the basic infrastructure, and a couple of volume shaders encapsulating the actual mapping algorithm. We use a texture-based direct volume rendering algorithm [5] employing either 3D or 2D texture maps depending on the capabilities of the underlying graphics hardware. Fig. 1 shows the internal structure of the volume node and the interaction between the different modules.

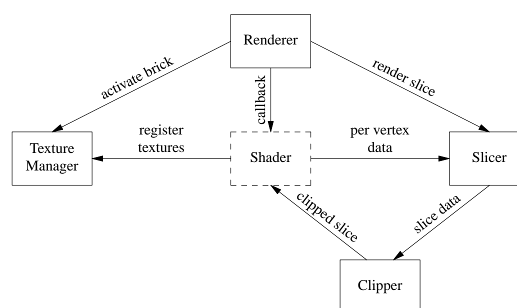

Figure 1: The modular design of the volume node.

The renderer module is the controlling instance that is responsible for steering the whole rendering process. It initiates the generation of slice polygons—either view port parallel or axis aligned, depending on the available texture targets—by the slicer and calls a shader module that renders the resulting slices with an appropriately OpenGL setup and the textures supplied by the texture manager.

In 3D texture mode, the renderer and the texture management modules are also responsible for volume bricking, since texture memory is always a short resource with respect to the ever growing size of volume data sets. The volume is split into bricks or tiles which completely fit into the available texture memory. The renderer assures, that the bricks are rendered in back-to-front order with the texture manager providing the suitable texture maps. Special care is taken of textures that cannot be bricked but have to stay resident in the texture memory, e.g. textures used for dependent lookups or transfer function tables.

The shader module is different from the other modules shown in Fig. 1, in such that it can be exchanged by means of a plug-in concept. This makes it possible to change the visualization algorithm by simply replacing the shader module. The shader is responsible for registering the volume data as a texture with the appropriate format. It is also possible for a shader to specify an arbitrary number of per-vertex attributes which will be linearly interpolated along the edges of the slice geometry. Because the volume node only implements the infrastructure needed for rendering and the actual OpenGL setup is done by the pluggable shader modules, new volume rendering algorithms or hardware specific implementations can easily be handled by providing customized shader objects. In this context the shader functionality also provides an abstraction

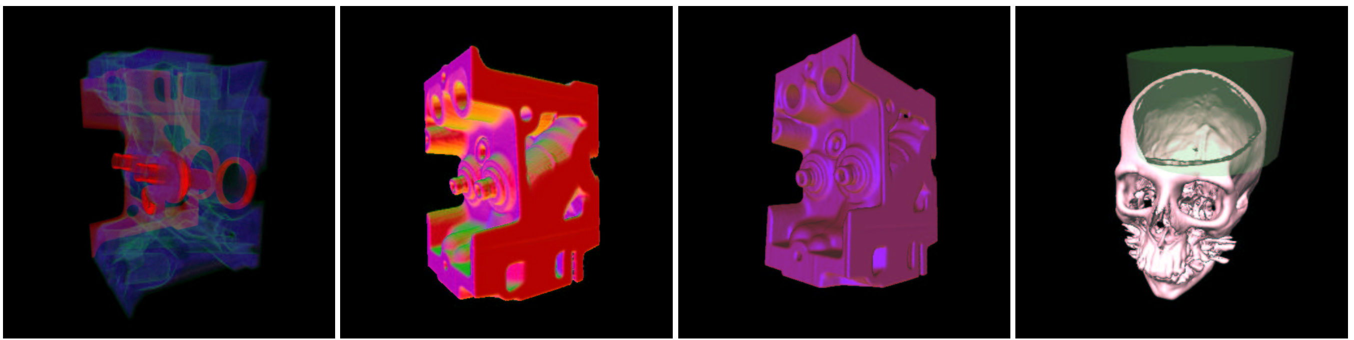

Figure 2: Different shading modes for the same volume data set are presented. On the left an appropriate transfer function is applied. The second image shows an iso-surface diffusely lit by 6 differently colored light sources. The third one shows the same iso-surface lit by 3 light sources with diffuse and specular contribution. The last image depicts how a volume can be modified by a geometrically defined clip object.

layer, separating the desired rendering effect from the available hardware support.

A major problem in volume visualization is occlusion and, therefore, it is often desirable to remove parts of the volume in order to reveal interior structures. A transfer function is often not sufficient to achieve that goal and at the same time to emphasize the structures one is interested in. In order to bypass this limitation volume clipping can be used, that removes parts of the volume given by one or more clip geometries. The clip geometries representing closed manifolds specified by geometry nodes in the scene graph can be interactively assigned to a volume. Clipping is implemented as slice clipping which means that we are not rendering the complete slice polygons but only the sections that—depending on the user-selected clipping mode—lie either within or without the clip objects. These sections are computed by intersecting the triangulated clip geometries with the volume slices using a fast incremental Sutherland-Hodgman-like algorithm that exploits the coherence between the contours on successive slices. Afterwards the clipped slice polygons are determined by tessellation based on those polylines and the clipping mode. Because clipping is done completely in software this does not interfere with the shader concept. Note that clipping based on tagged clip textures [6], could easily be implemented as a special shader module.

3 Results

In this section some example images, generated using a simple interactive volume viewer application built upon the previously described framework, will be presented. In order to demonstrate the applicability of the volume shader concept we show some example images using different volume shader modules.

The first example, the semi-transparent rendering shown in the leftmost image of Fig. 2, was generated by rendering a $256 \times 256 \times 128$ voxel data set using our color table shader. This shader implements the most common approach in direct volume rendering—the mapping of data values to color and opacity using a tabulated transfer function. The image shown was generated on a SGI Onyx4 UltimateVision visualization system using a fragment program that realizes a post-shading transfer function by means of a dependent texture lookup. Second, we present an example of the extraction of iso-surfaces from volume data sets. We implemented a shader module that renders shaded iso-surfaces with an algorithm as introduced in [7] that was slightly enhanced regarding the lighting computation. Both shaders adapt to the capabilities of the available graphics hardware in selecting an optimal OpenGL setup with respect to image quality and performance. The second and third image in Fig. 2 show two examples of illuminated iso-surfaces from the aforementioned data set rendered on a NVIDIA GeforceFX. They differ in the number and properties of the applied light sources. The first one was illuminated by six different purely diffuse lights while in the second one three lights with both, specular and diffuse contributions were used. The last image in the row of Fig. 2 shows an example of a clipped volume. A cylindrical geometry is applied as clip object to unveil the interior of the skull that is rendered as a specularly lit iso-surface.

4 Conclusion

In this document we have briefly described an extension of the OpenSG scene graph with a framework for texture-based direct volume rendering. Volumetric objects can be included into any OpenSG scene. The framework fits seamlessly into the existing scene graph structure of OpenSG, thus, enabling the application programmer to use volumetric effects without any additional effort. This in particular includes parallel rendering applications in a cluster environment, as we will demonstrate with a simple setup using four PCs to drive a large stereographic rear projection display system. The framework hides the intricate tasks of texture management, slice generation or volume clipping from the developer reducing his work to the task of providing the data and selecting the right shader to achieve the desired effect. Additionally, employing the hardware abstraction layer provided by the volume shader concept, it is easy to realize new shaders to support different graphics adapters or adapt an existing shader to use new features of upcoming graphics chips generations. We have demonstrated the usefulness of this concept with examples of shaders encapsulating different volume rendering techniques, e.g. iso-surfaces.

References

[1] OpenRM, http://openrm.sourceforge.net/.

[2] OpenSG, http://www.opensg.org/.

[3] SGI OpenGL Volumizer, http://www.sgi.com/software/volumizer/.

[4] TGS, Open Inventor VolumeViz extension, http://www.tgs.com/.

[5] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Eurographics / SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118,147, 2000.

[6] D. Weiskopf, K. Engel, and T. Ertl. Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization. In *Procceedings of IEEE Visualization '02*, pages 93–100, 2002.

[7] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, pages 169–177, 1998.