Chapter 8

VolPack: A Volume Rendering Library

This chapter describes the design of an experimental software library called VolPack. The library provides a high-performance implementation of the algorithms in this thesis with a simple, flexible programmer's interface.

A fundamental problem when designing a volume rendering package is that rendering speed and flexibility are conflicting goals. The fastest rendering speed can be achieved by hardwiring all options, including the layout of a voxel and the functions for shading and classification, and carefully hand-tuning the code for a particular hardware platform. The most flexible system can be achieved by providing many rendering options and allowing the user to change any of them at run time.

Existing volume rendering packages generally fall at one of the two extreme ends of this spectrum. Montine [1990] proposes a flexible architecture for a volume rendering library based on user-defined callback functions for shading and classification. Alternatively, a general class of shading functions can be supported by letting the user specify a shading language procedure that is compiled and dynamically linked at run time [Corrie & Mackerras 1993a]. These approaches maximize flexibility but make it difficult to optimize specific cases since the renderer and the callback functions are developed and compiled separately and must adhere to a predefined interface. Rigid interfaces often lead to performance bottlenecks.

A number of existing systems take the opposite approach by providing a set of visualization "tools" with powerful but fixed functionality [Brittain et al. 1990, Avila et al. 1994].

The only way to extend such systems, for instance to add a new shading algorithm or to support a new voxel data type, is to modify the source code for an existing tool or to write a new tool. Typically the systems are designed with multiple layers of abstraction and modules with well-defined interfaces to facilitate adding new functionality, but extending the system still requires substantial effort. On the other hand, this architecture allows each tool to be fully optimized.

Finally, a number of commercial visualization systems (including AVS [Upson et al. 1989], Data Explorer [Lucas et al. 1992] and IRIS Explorer) are based on a data flow model: the system contains a collection of data processing and visualization modules that may be connected in a data flow network. Each module in a data flow system is independent and has a general interface to maximize reusability, so the system cannot optimize computation across modules. As a result, one can either implement a single monolithic volume rendering module with a fixed set of options and optimal rendering speed, or a flexible collection of interoperable shading, classification and rendering modules with significantly lower performance. However, there are no intermediate points in the flexibility/performance spectrum.

VolPack was designed to provide as flexible an interface as possible without significantly sacrificing performance. Specifically, the goals include avoiding hardwired formats for the volume, the shading function, and the classification function. Another goal is to provide a simple application programmer interface (API).

8.1 System Architecture

VolPack consists of a library of software routines for volume rendering and a specification of the interface between an application program and the library routines.

The software library includes a collection of core routines for specifying parameters and rendering a volume, plus a collection of utility routines that implement common shading and classification algorithms. These routines are designed to be building blocks for creating an application. The library does not specifically support a graphical user interface but can be combined with general user-interface toolkits.

The interface between the application program and the volume rendering library is the most critical part of the design since the interface impacts flexibility and performance. The

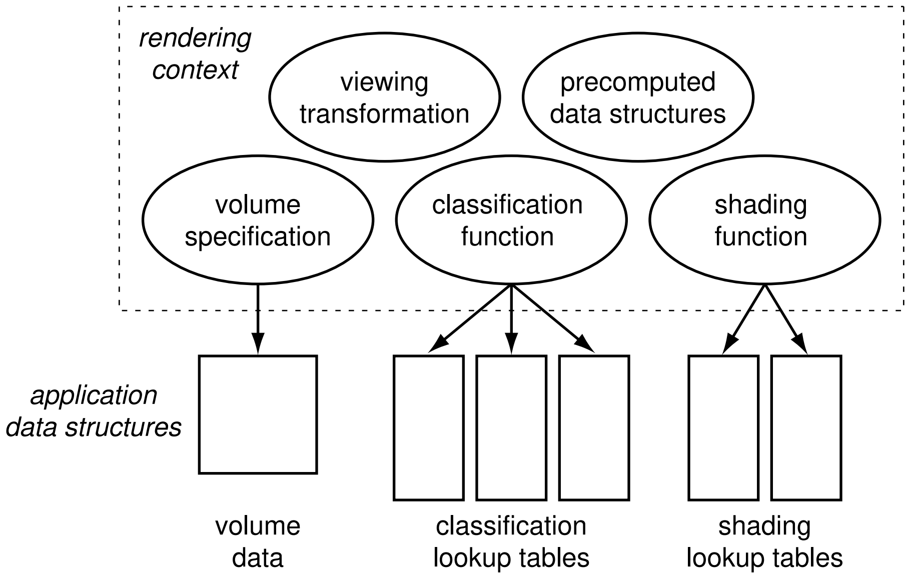

Figure 8.1: Schematic diagram of a VolPack rendering context. The context encapsulates user-specified parameters and internal precomputed data structures, as well as pointers to external application-specific data structures.

two aspects we focus on in this chapter are volume representation (i.e. the specific layout of the data in a volume) and specification of the shading and classification functions.

A volume rendering application built on top of VolPack first calls several library routines to set rendering parameters and then calls a rendering routine. The rendering parameters and all of the internal state required to render a volume are stored in a data structure called a *rendering context* (Figure 8.1). All of the library routines require a context as one of the parameters, except for the routine to create a new context.

Most of the routines in VolPack operate by transforming the state in a rendering context, a model inspired by the OpenGL interface [OpenGL Architecture Review Board et al. 1993]. The library routines allow the programmer to set options in the context, and the options remain in effect until they are disabled by the programmer. Most of the options have reasonable defaults, simplifying the programmer's job. The context also provides a place for routines to store precomputed data structures (e.g. a min-max octree or a run-length encoded representation of a volume).

The specific information stored in the rendering context is not visible to the programmer and is not defined by the VolPack interface. The context abstraction allows the library to store information in the best format for use during rendering without burdening the programmer with a complicated interface. All necessary state is stored in the context rather than in global variables internal to the library so that the library code is reentrant.

The rendering context abstracts the internal implementation of the library's data structures in a way similar to object-oriented systems. However, in contrast to such systems VolPack defines only a single type of object (the rendering context) that contains all necessary state. Partitioning the state of a volume rendering system into a deep hierarchy of objects with well-defined interfaces has been shown to result in poor performance since the interfaces lead to overhead and hinder optimization [Zuiderveld & Viergever 1994]. Encapsulating the entire library in an object for inclusion in a larger system may be practical, but the potential benefits of the object-oriented model are dubious for the internal implementation.

8.2 Volume Representation

The volume rendering algorithms described in this thesis apply to volumes sampled on a regular grid and stored as a 3D array of samples, so VolPack does not currently support other data representations. However, the library does allow an arbitrary format for the voxels in the 3D array.

The application must define a volume by specifying its dimensions (width, height, and depth), a pointer to an array containing the voxel data, and a description of the layout of a voxel. Each voxel may contain a sequence of fixed-size fields. Every voxel in a volume must have identical fields, but the application defines all aspects of the layout including the number of fields, the sizes of the fields, and the interpretation of each field. For example, the representation can support voxels containing the following types of information:

- 8-bit, 16-bit or 32-bit scalar values
- multiple scalar values per voxel, e.g. data acquired using different scanning technologies
- precomputed data in addition to the original scalar values, e.g. normal vectors and gradients

In an application the layout of a voxel can be described using a compound data type such as a C-language structure or a Pascal record. The programmer then describes the voxel layout to the library using a set of routines provided by VolPack.

In the internal implementation these language constructs cannot be used since the layout of a voxel is not known when the library is compiled. Instead, the size of each field (in bytes) and the offset of each field from the beginning of the voxel are stored in an array in the rendering context. Each field can be identified by its ordinal index in the array. Library routines that must access the data in a voxel, such as classification and shading functions, take a field index as an argument and use the field offset array to find the location of the field within the voxel.

This design allows a single library routine to render volumes with arbitrary voxel layouts. It is possible to write shading and classification functions that are independent of a specific voxel layout. It is also possible to define the format of a voxel at run time based on user input or information in the header of a data file. The overhead of the extra level of indirection is small: to compute the address of a voxel field the renderer need only add the field offset to the address of the voxel.

8.3 Classification and Shading Functions

VolPack uses the flexible lookup-table shading algorithm described in Section 7.1 for both voxel classification and shading. The application specifies a shade tree and a set of lookup tables for each of the classification and shading functions. The speed of the method was demonstrated in Section 7.1.2.

In the current implementation of VolPack the shading tree is limited to a sum-of-products expression with lookup tables at the leaves, and the classification tree is limited to a product of the values from the lookup tables. Even with these restrictions a wide variety of shading functions can be implemented since an application can initialize the tables arbitrarily.

The method extends to more general shading functions as described in Section 7.1.3. To achieve optimal performance with user-defined shading functions we envision a system that operates as follows. The shade tree can be translated automatically into a C-language macro that implements the shading function. The translator can also provide a list of input values required by the function, such as values from particular voxel fields. The system can then dynamically compile and link a specialized rendering routine with the shading macro inlined. The list of input values can be used to tailor the rendering routine so that computation that is not required can be omitted. For instance, if the user wishes to generate a maximum intensity projection instead of a true volume rendering then the shader will not reference the opacity of a voxel, so the code to compute and resample opacities can be omitted from the renderer.

VolPack currently contains predefined macros implementing a small number of shading functions. Specialized renderers for each of these shaders are generated by inlining these shading functions at compile time. We have not yet implemented the dynamic compilation and linking facility but this technique has been used by others. Corrie's implementation of data shaders [Corrie & Mackerras 1993a] uses dynamic linking to support shaders defined at run time and implemented via callback functions. We propose going one step further by inlining the shader in the renderer, customizing the renderer for the shader, and then compiling and linking the optimized renderer.

8.4 Viewing Model

VolPack uses four-by-four homogeneous transformation matrices to specify viewing transformations, giving the application writer the flexibility to construct viewing transformations with any 3D viewing model. For convenience VolPack also provides routines to construct matrices using the viewing models defined in the OpenGL standard [OpenGL Architecture Review Board et al. 1993] and the PHIGS standard [Foley et al. 1990, Chapter 6].

8.5 Functionality Provided by VolPack

The previous sections have described the architecture of VolPack and the techniques used to provide a flexible, efficient programmer's interface. This section briefly describes the categories of routines provided by the library:

context manipulation Routines to create and destroy a rendering context and to retrieve state variables from a context.

volume specification Routines to define the size and layout of a volume and to attach an array of volume data to a context.

classification Routines to define a classification function (including the lookup tables and the shade tree) and to precompute data structures for fast classification (i.e. a min-max octree) and fast rendering (i.e. a run-length encoded volume).

shading Routines to define a shading function. A higher-level routine to build a shading lookup table for the commonly-used Phong illumination model is included for convenience.

view transformation Routines to construct a four-by-four viewing transformation matrix. The current transformation matrix for a volume is stored in the rendering context.

image specification Routines to define the size of the rendered image, the format of the pixels, and an array for storing the image data.

rendering Routines to invoke one of the volume rendering algorithms and to set options specific to a renderer. The options are stored in the rendering context.

utilities Routines for file I/O, volume reformatting (extracting and inserting subvolumes, resampling, and transposing), and linear algebra.

Given these building blocks, a simple volume rendering application can be written using 30-50 lines of C code.

8.6 Chapter Summary

In this chapter we have discussed approaches to designing a volume rendering library interface that strikes a balance between speed and flexibility. VolPack uses a procedural interface and encapsulates all internal state in a rendering context rather than using an object-oriented interface or a data flow model. VolPack's high-level interface eliminates the overhead associated with a large collection of independent objects or modules.

VolPack's volume representation supports volumes with arbitrary data defined on a regular grid. The flexible voxel format does not impact performance.

We have proposed a shading and classification system based on rendering routines that are automatically specialized for a particular shade tree through dynamic compilation and linking. This method supports a very general class of shading functions and provides good performance. Rendering functions that have been hand-tuned for a particular set of shading parameters and a specific voxel layout are likely to perform marginally better, but with a great loss in flexibility.

The current version of VolPack only partially implements the architecture described in this chapter, but the existing interface requires little modification to incorporate all of these features. Combined with the shear-warp rendering algorithms, VolPack provides fast rendering with substantial flexibility.