Volume Graphics (2003)
I. Fujishiro, K. Mueller, A. Kaufman (Editors)

OpenVL - The Open Volume Library

Sarang Lakare and Arie Kaufman[†]

Center for Visual Computing (CVC)
and Department of Computer Science
Stony Brook University
Stony Brook, NY 11790-4400, USA

Abstract

OpenVL is a modular, extensible, and high performance library for handling volumetric datasets. It provides a standard, uniform, and easy to use API for accessing volumetric data. It allows the volumetric data to be laid out in different ways to optimize memory usage and speed. It supports reading/writing of volumetric data from/to files in different formats using plugins. It provides a framework for implementing various algorithms as plugins that can be easily incorporated into user applications. The plugins are implemented as shared libraries which can be dynamically loaded as needed. OpenVL is developed openly and is a free software available on the web.

Categories and Subject Descriptors (according to ACM CCS): I.3.8 [Computer Graphics]: Applications, I.4.10 [Image Processing And Computer Vision]: Volumetric

1. Introduction

Volumetric data is used in many disciplines ranging from biomedical sciences to seismic sciences. Even with the wide spread use of such data, there is no standard library for handling them. All the currently available systems, such as VTK [12, 13], VolVis [1], AVS [15], OpenDX (formerly Data Explorer) [6], Khoros [8] etc., provide high-level functionality mainly for the purpose of visualizing the data. Most of them do not provide low-level volume access functionality and a framework for handling volumetric data. Some libraries such as ITK [7] and ImLib3D [2], which were developed at the same time as OpenVL, do provide some low-level access functionality but lack the support for multiple data layouts and a dynamic plugin framework which we feel is critical for flexibility, extensibility, and ease of use. Other libraries such as VLI [11] are developed specifically for certain hardware and do not provide any data access functionality.

The main motivation for our work is the lack of a standard framework for working with volumetric datasets. Any researcher or developer intending to work with volumetric data has to build tools that provide the basic functionality needed for accessing the data. OpenVL is a framework which allows the users to concentrate on algorithm development and implementation and not bother with the low-level volume access issues. It also makes the code more manageable, less prone to errors, and more readable.

The second motivation for our work is the need for a standard platform for collaboration in the community. We want to encourage sharing of algorithm implementations to maximize code reuse and minimize duplication of efforts. For this, the OpenVL framework provides support for *plugins*. This allows researchers and developers to provide their algorithm implementations as OpenVL plugins which others can easily incorporate into their own code. For example, a plugin may provide a volume subdivision, a region-grow capability, or an implementation of a newly published work. As these plugins are used by other users, it is likely that they will be optimized and improved. As a result, all the users of OpenVL will have access to the most optimized implementation of various algorithms.

OpenVL is a low-level library that provides a uniform application programming interface (API) for volumetric data access, layout, and implementation of various volumetric algorithms. It is designed to meet the following key objectives:

Uniform volume access API: OpenVL provides uniform

[†] {lsarang,ari}@cs.sunysb.edu

© The Eurographics Association 2003.

and standard API for accessing volumetric data stored in different layouts. This makes handling volumetric data relatively easy. The resultant code is less error prone, and more readable.

Modular: OpenVL is modular. Almost everything in OpenVL is implemented as a plugin which makes it very easy to add and remove functionality.

Extensibility: OpenVL is designed to be extensible. All the functionality provided by OpenVL can be extended by implementing additional plugins. These plugins can be provided by third parties and need not be part of OpenVL. Their functionality will be available immediately to all OpenVL enabled applications.

High performance: Every part of OpenVL is implemented to provide maximum performance. The OpenVL design allows users to tradeoff between flexibility and performance, where flexibility can lead to reduced performance.

Ease of use: The various APIs used in OpenVL are designed to be as simple as possible. All APIs are documented and reference documentation is always available on the OpenVL website. The use of plugins allows users to employ algorithms implemented by others without knowing the intrinsics of the implementation.

Open source: We strongly believe in the fundamentals of open source. The entire source code for OpenVL is freely available on the Internet from the OpenVL website. The development of OpenVL is open and contributions are encouraged.

Modern techniques: OpenVL uses modern techniques for a flexible as well as optimized implementation. The library is implemented as a shared library which applications can dynamically link to. The library uses powerful C++ techniques such as templates, partial specialization of templates, code inlining, etc.

2. OpenVL Overview

Figure 1 shows an overview of OpenVL. The arrows indicate the calling sequence. The user application is at the highest level and makes use of various OpenVL components.

There are four main components of OpenVL:

- **Volume:** It is responsible for storing the volumetric data in various layouts and providing access to the data.
- **Volume File Input/Output:** It is responsible for loading volumetric data stored in user's files into the Volume component and writing the data in the Volume component to user's files.
- **Volume Processing:** It provides a framework for implementing various volume processing algorithms. By volume processing we mean any task that can be performed on a volume. This includes image processing. A task can be as simple as computing the histogram of a volume to performing some complex segmentation [5, 10].

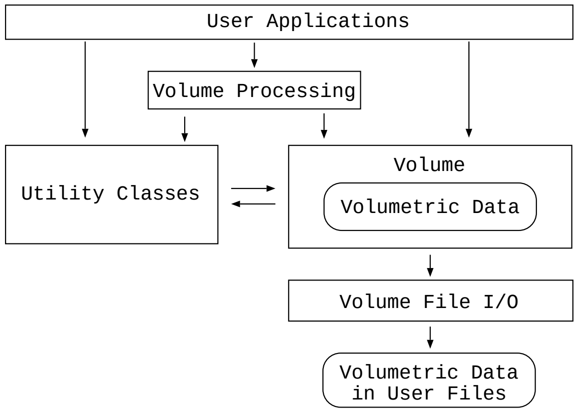


Figure 1: *Overview of OpenVL.*

- **Utility Classes:** These are a collection of classes commonly needed when working with volumetric datasets.

In the next sections we discuss each of the OpenVL components in detail.

3. Volume

The volume component is the central and most important part of OpenVL. It holds the volume data and provides access to it. The internals of the volume component are shown in Figure 2. It comprises three parts: the Volume API, the Volume Access API, and the Volume Data Layout API.

3.1. Volume Data Layout

The OpenVL framework supports multiple *layouts* for volumetric data. A layout is a method of laying out volumetric data on a medium (memory, disk etc.). For example, loading volumetric data into memory as a linear array is one layout method.

Although a linear array can be used to lay out any volumetric data, it is not necessarily the best way. For example, a volumetric data of dimensions $X \times Y \times Z$ requires an array of size $X \times Y \times Z$. The total memory required for this type of layout is the size of the array times the size of each voxel.

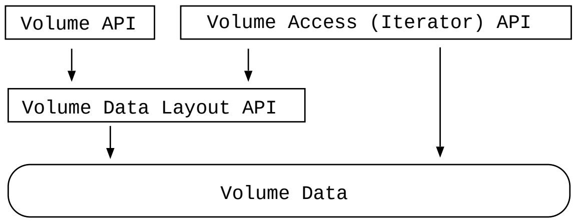


Figure 2: *Overview of the Volume component in OpenVL.*

Now suppose the data is sparse and contains only a few significant voxels. It would then be a real waste of memory to represent the whole data as an array. Instead, some compression technique [4] such as run length encoding (RLE) [9] can be applied to decrease the memory footprint of the layout.

Our goal with OpenVL is to allow different layout schemes for any volumetric data. We achieve this by abstracting the volume data from the volume access mechanism. We encapsulate the different layouts behind an API called the Volume Data Layout API. Figure 3 shows the organization of different data layouts inside the volume component of OpenVL. Every layout has to implement the Volume Data Layout API for its own data layout type. The dashed arrows indicate inheritance. As of this writing, OpenVL has two layouts implemented: the *Linear* layout and the *Simple Run Length Encoded* (RLE) layout. We show both of these in Figure 3. When a new layout for the data is implemented, it will have to implement the Volume Data Layout API. We show a new layout with dashed boxes. As the different data layouts are abstracted from the user applications, the applications can work with any data layout.

Additional data layouts promise to open many possibilities. We list a few of them here:

- **Disk-based layout:** The volume data can be kept on disk and accessed from the disk itself. This can be useful when memory is extremely critical such as with handheld PDAs.
- **Disk-based cached layout:** As in the previous case, the volume data can be kept on disk, but a part of the volume can be loaded into memory as needed [3]. This can be helpful when working with huge datasets and limited memory [16].
- **Network-based layout:** The volume data can be kept on a central node in a distributed environment with the nodes fetching a part of the data as and when they need.
- **Texture-memory layout:** The volume data could be loaded into texture memory available on most current commercial graphics cards.

Since the layout is hidden from the applications, any application written using OpenVL will automatically make use of the unique features of the layout. Suppose we have an application which computes the histogram of a dataset. This application can work on huge datasets using a disk-based layout, and at the same time, this application can work in a distributed environment using the network-based layout. The same application will also be able to use texture memory instead of main memory for loading the dataset it is given.

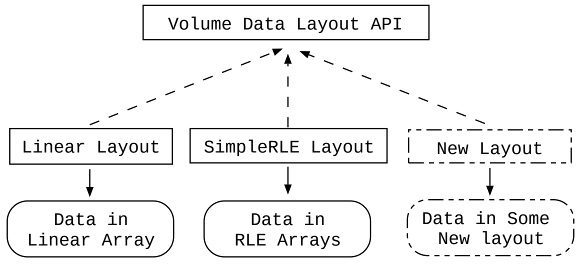


Figure 3: *Data kept in different layouts.*

The volume data layout API is simple and relatively small. It mainly exports information about the data. This information includes the dimensions of the volume, the datatype of the voxels, the voxel unit distance, etc. In addition, it provides a very minimal API for accessing the voxels in the underlying layout: `getVoxel()` and `setVoxel()`. The reason for this minimal access API will be clear when we look at the more elaborate volume access API in the next section.

3.2. Volume Access - Iterators

We now present a more elaborate API to access the data voxels. Separating data layout and access was a design decision in OpenVL, which provided flexibility. To give access to voxels in the data, we use *iterators*. An iterator is a concept in C++ used extensively by the standard template library [14]. An iterator is defined as an object that moves through a container of other objects and selects them one at a time, while hiding the implementation of that container. In our case, the container is the volume data stored in different layouts.

Figure 4 shows the design of iterators in OpenVL. The base for all iterators in OpenVL is the Volume Access API also known as the Iterator API. Any iterator in OpenVL implements this API. An iterator can be implemented for every data layout supported in OpenVL. Such iterators are given direct access to the data in the layout (Figure 2) to allow the most optimized access to the voxel data. For example, the Linear Iterator has direct access to the data stored in Linear Layout and implements the Iterator API to provide optimized access to the voxels stored in the layout.

Other than the iterators which are implemented for the data layouts, there are two special iterator implementations in OpenVL which are independent of the layouts: the *Generic* iterator and the *VirtualCall* iterator. The Generic iterator can be used to access data in a layout when there is no iterator specific to that layout. This iterator uses the minimal access API provided by the data layout API (Section 3.1) to access the data stored in the layout. This iterator can be useful when implementing a new layout. The new layout can be tested without actually implementing a native iterator for it. This iterator can also be useful in cases where a native iterator does not provide any significant performance gains.

The VirtualCall Iterator is responsible for abstracting the layout specific iterators from the user applications. This iterator acts like an interface between the Iterator API calls from user applications and the Iterator API of the layout being used. Any Iterator API calls received by this iterator are

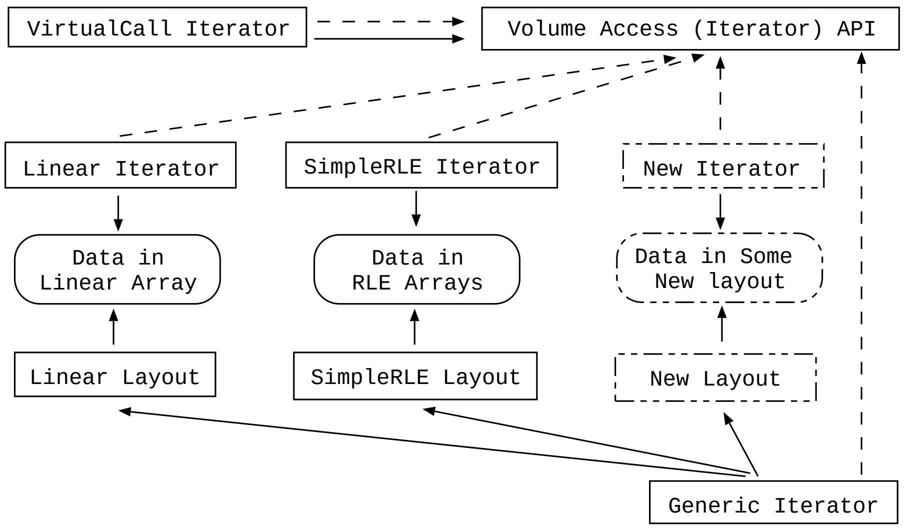


Figure 4: *Data access using iterators in OpenVL.*

piped to the corresponding call of the underlying layout's native iterator or the generic iterator in the absence of a native iterator. Since this is done at run time, the user applications can work with volumes in any layout without needing a recompile. We will talk more about this special iterator when we look at its implementation in the following sections.

Iterators in OpenVL are designed so that they always point to a voxel in the volume. They can be moved around the volume to point to any other voxel. The iterator API is carefully chosen to give quick and easy access to the voxel data and the data stored in the neighboring voxels. The iterator can be divided into three categories:

- **Position change operations:** These function calls can be used to move the iterator to another voxel in the volume. Some of the important functions which belong to this category are:

  `next()` moves the iterator to the next voxel in the data.
  `nextXYZ()` moves the iterator the next voxel along X axis.
  `moveTo()` moves the iterator to a new position.
  `prev()` moves the iterator to the previous voxel in the data.
  `prevXYZ()` moves the iterator to the previous voxel along X axis.

- **Value query operations:** These function calls give access to the data stored in the voxel pointed to by the iterator and the voxels in the neighborhood.

  `get()` returns the voxel data pointed to by the iterator.
  `getRelative(offset)` returns the voxel data at a relative position from the current voxel.
  `set(value)` sets the voxel data to `value`.
  `setRelative(offset, value)` sets the voxel at a relative position to `value`.

- **No bounds checking operations:** These special functions perform the position change and query operations without performing bounds checking. These are provided mainly because bounds checking is an expensive operation. At times it can be faster to perform explicit bounds checking (as is done when accessing volume data using pointers) and then using these functions. For example, in ray casting, when the exit point of a ray can be predetermined, bounds checking by the iterator is unnecessary. Using these functions has the same side effects of using pointers - errors can be introduced which can go undetected. To emphasize this danger, these functions are named similarly to the other functions but with a "NBC" suffix. For example, the no bounds checking version of `getRelative()` is called `getRelativeNBC()`.

It is also important to note that the Iterator API is the minimal API that each iterator has to implement. It is possible for an iterator to provide additional functionality. For example, the RLE iterator can provide additional function to get the size of the run-length to which the voxel belongs. In such cases however, the user application will need to know the type of layout used.

3.3. Volume Objects

The third part of the volume component of OpenVL is the Volume object. This object is encapsulated by the Volume API which is defined in a class called `vlVolume`. Objects of this class are the volumes which users' applications can use.

The `vlVolume` class does not store the volume data by itself. Instead, it uses the Volume Data Layout object to store the data. The main purpose of the Volume API is to provide information about the volume and file input/output functionality. The volume information such as the dimension, voxel units, etc., is simply fetched from the underlying data layout object and exported. There is also provision for associating additional information (metadata) with a volume such as author name, the name of the file from which the volume was loaded, date of creation, etc. Metadata of any type can be associated with a volume.

The Volume API provides two important functions for file i/o. The `read()` function is provided to read volumetric data from a file and `write()` function is provided to write the data to a file. The file format to be used is automatically detected (using the file extension) and the appropriate file format plugin is called to read or write. Optionally, the file format can be specified when calling read/write. The file i/o unit of OpenVL is discussed in the next section.

4. File Input/Output and Utility Classes

This component of OpenVL is responsible for providing read and write functionality for various file formats. To provide this functionality, we have an interface which can be implemented for a specific file format. This interface is called the File Input/Output API (Figure 5). At the time of this writing, OpenVL provides support for three file formats: the VolVis [1] Slice format, the Volpack [9] DEN format, and the raw file format. To provide support for a new file format, a new class which implements the File I/O API is needed (as shown by dashed box in Figure 5). The new file format will then be automatically used by the Volume API (Section 3.3).

The File I/O API is very simple which makes writing file i/o plugins easy. It consists of three functions which need to be implemented. The first is `readInfo()`. This function reads information about the volume data that resides in the file: its dimensions, data type, etc. The second function is `readData()` which actually reads voxels from the file and loads them into the volume. The loading of voxels can be implemented using iterators so that it is independent of the layout used by the volume. The third function is `writeData()` which writes the data stored in the layout to the file. In addition to these, there is one more function for finding which file extensions are suppored by the file format: `getFileExtensions()`.

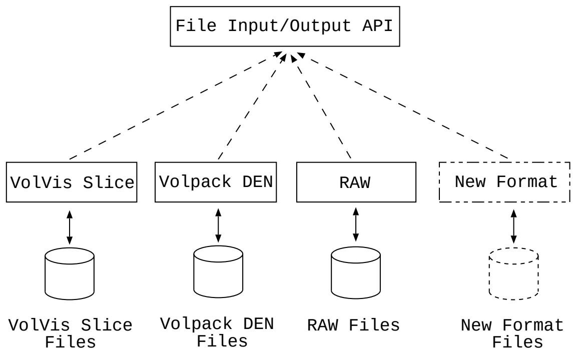


Figure 5: *File Input/Output component of OpenVL.*

OpenVL provides many utility classes that make it easy to work with volumetric data. Some of these classes are:

`vlSlice:` It gives a slice cut from a volume. It supports arbitrary orientation.
`vlTxFunction:` It provides a transfer function. It provides the API for manipulation of the transfer function and for creating a look up table of arbitrary width.
`vlClock:` It finds the time used with precision of the order of 1μ sec.

In addition to these, OpenVL provides commonly used classes, such as: `vlTriple, vlVector, vlNormal, vlMatrix, vlDim, vlUnit, vlPoint,` etc.

5. Volume Processing

The volume processing component is a framework for implementing various volume processing tasks and making them available to users for inclusion in their applications. This component lies on top of the other OpenVL components (Figure 1) presented in the earlier sections. The tasks implemented using this framework make use of the access functionality provided by OpenVL described previously.

Each volume processing task has to implement an API called the Volume Processing API. In Figure 6 we show some of the tasks which are implemented in OpenVL. The API is designed such that it can be used to implement any task.

The Volume Processing API includes just four functions. We believe that almost any task can be implemented using this API. To allow the flexibility to pass arbitrary data to a volume processing task, we designed a special class which can store data of any type: the `vlVarList` class. The name

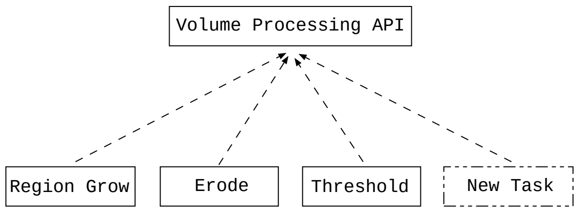


Figure 6: *Volume Processing component of OpenVL.*

is short for *Variable List*. Each `vlVarList` object contains a list of variables. Each variable is a name-value pair. That is, a name which is a `string` object and a value which can be of any class. This allows storing an object of any class with a name associated with it. This object can then be retrieved by simply providing the associated name.

The first function of the volume processing API using the `vlVarList` class is the `config()` function. This function returns a pointer of type `vlVarList`. The input variables required for the volume processing task are set by the user application using `vlVarList` function calls. The next function of the volume processing API is the `setVolume()` class. This class is used to set the volume on which the volume processing is performed. The third function is the `run()` function. This is called to initiate the volume processing task. The `run()` operation returns when the volume processing is complete. The fourth and final function, `results()` also makes use of the `vlVarList` class. This function returns a pointer of type `vlVarList` that holds the results of the volume processing task.

Since `config()` allows the user to pass any information to the volume processing task, it effectively extends the API to accommodate different volume processing tasks. The user just needs to know the name of the variables that the task expects and pass those to the task.

All the volume processing tasks are implemented as plugins. This will be described in the next section.

6. Implementation

We have implemented the OpenVL library using standard C++ and other cross-platform tools to make porting to other platforms easy. Our current development is on the Linux operating system and uses the GNU C++ compiler. The implementation of OpenVL accomplishes three goals:

- **Fast:** Since the major part of OpenVL is at a very low-level (voxel access level), speed is a very important concern.
- **Ease of use:** Our implementation focuses on the ease of use of our library.
- **Hiding templates:** One important goal is to hide the C++ templates from the user as much as possible, while making extensive use of them internally. This allows efficient and flexible implementation of application code.

We now present various considerations of the OpenVL implementation.

6.1. Multiple Data Types

Volumetric data can have a variety of data types. The data type can range from unsigned char to float to color. Our goal is to support all the commonly used data types and any new data types that a user might need. For this purpose, we have made use of C++ *templates* in our implementation. Templates allow writing code for one data type which can be used for other data types without any modifications.

In Section 3.1 we described the data layouts. Since these directly interact with the data, we use templates for this implementation. The volume data layout API is templatized over data type. The different layouts that implement this API can also templatize themselves over the data type enabling the layouts to support any data type. All the layouts which are built into OpenVL are templatized and can be used with any data type. The diagram in Figure 3 actually refers to just one instance of the templatized API. The diagram will replicate for each data type for which the code is instantiated.

The iterator API which provides access to the data in the layouts is also templatized over the data type. The diagram in Figure 4 shows the structure for only one particular data type. This whole structure repeats for all data types for which the code is instantiated.

6.2. Shared Library

OpenVL is built so that it compiles into a shared library. A shared library is a dynamically linked library which applications can link to at run time rather than at compile time (as with static libraries). This gives OpenVL all the benefits of shared libraries. One main advantage to this approach is that all applications using OpenVL will automatically benefit from any update to the OpenVL library without recompiling. Another advantage is that the memory consumption is reduced when multiple applications use the library as only one instance of the library remains in memory.

We have seen that OpenVL uses templates internally to support multiple data types. Here, we present OpenVL as a pre-compiled shared library that applications use at run time. Since templates are instantiated at compile time, it becomes difficult to pre-compile code for all the different data types, and impossible to pre-compile code for unknown data types. To solve this problem, we pre-compile code only for commonly used data types. For any other data type, there are two possibilities. The first is to compile the code for the new data type into the user application itself. This way, the code for commonly used data types is inside OpenVL, and the code for the new data type is inside the user application. The second option is to recompile OpenVL with support for the new data type. We have made the list of compile-time data types configurable so that users can easily add new data types and recompile OpenVL.

6.3. Dynamic Plugins

One of our main goals with OpenVL was to make the library extensible. To realize this goal we use dynamic plugins. Dynamic plugins are basically shared libraries which are loaded at run time when the functionality provided by them

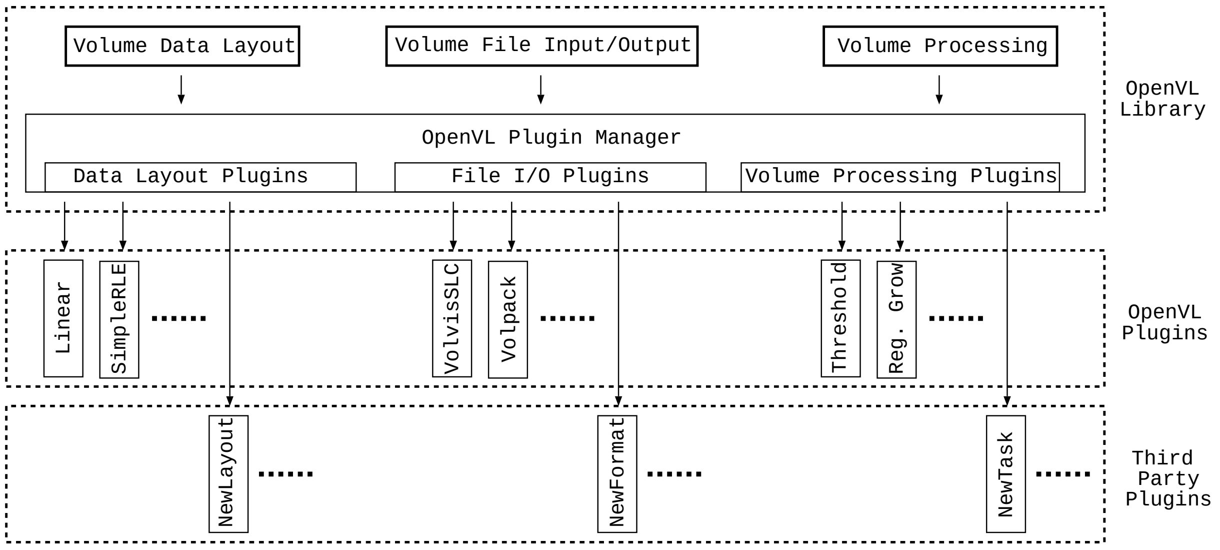


Figure 7: *Use of dynamic plugins in OpenVL.*

is needed. Since these plugins are implemented as shared libraries, they provide us with all the benefits of shared libraries that we discussed before.

To provide support for plugins, OpenVL has another component called the *Plugin Manager*. For simplicity we left out this component when discussing the structure of OpenVL in the earlier sections. The responsibility of the plugin manager is, as the name suggests, managing the plugins. This includes finding all the plugins that are installed in the system and creating a database of the available plugins and the functionality they provide, loading the appropriate plugin when a certain functionality is requested, unloading the plugin when the library is finished using it, etc.

The plugin manager can load any plugin that implements a specific plugin API. To create plugins for a specific interface, the plugin API provides a mechanism by which the plugin can provide objects for the given interface if one exists in the plugin. This mechanism can be used to create plugins for absolutely any interface. In OpenVL, we use plugins for three main interfaces - the volume data layout API (Figure 3), the file input/output API (Figure 5), and the volume processing API (Figure 6). This means that all the different layouts, the file i/o filters for different file formats, and the various volume processing tasks are implemented as dynamic plugins. In Figure 7 we show how these plugins are laid out. The OpenVL library consists of multiple shared libraries - the core library and the various libraries that each implements a dynamic plugin. The OpenVL library itself comes with a set of plugins. Users can implement their own plugins and OpenVL will pick them up automatically once the shared object files are stored in the correct directories on the system. In Figure 7 we show the user plugins as third party plugins.

The plugins for data layouts and file i/o are internally used by OpenVL and are never exposed to the user. The plugins for the volume processing tasks are however meant to be used by the user's application. To allow the user to query for an available task and use the plugin, we provide a *Trader* interface. This interface can be used to query the plugin manager for a specific task, say region growing. The plugin manager then searches through the plugins that implement the volume processing interface to find one that can perform region growing. If a plugin is found, it is returned to the user. The user can then use this plugin to perform the task she requested using the simple volume processing API introduced in Section 5.

The extensibility of the OpenVL plugins does not end here. The OpenVL plugin mechanism can be used to create dynamic plugins for any desired interface. For example, a volume rendering application using OpenVL can create an interface for volume rendering and then implement the different volume rendering methods as plugins. OpenVL thus provides a powerful mechanism which allows extending not only OpenVL itself, but any application that uses OpenVL.

6.4. Speed Optimization - Avoiding Late Binding

Throughout the design process of OpenVL, there is one factor that was given the most importance - speed. The design of OpenVL is influenced by the famous 80-20 rule: 80 percent of the execution time is spent in 20 percent of the code. OpenVL thus concentrates on making that 20 percent of the code as fast as possible, while making the other 80 percent as flexible as possible. Since the part of OpenVL which will execute most often is volume data access, making that part as fast as possible is our primary goal.

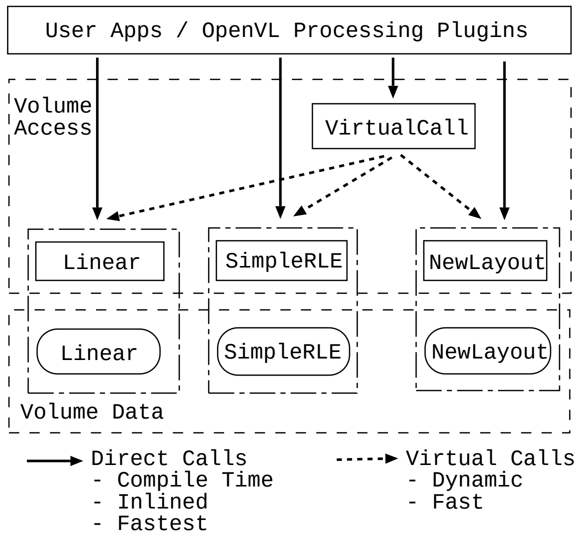


Figure 8: *Different ways to access data in OpenVL.*

In the earlier sections we saw that the data layouts are implemented as dynamic plugins which are loaded at run time. This gives us the flexibility to run the user applications on different data layouts - even new data layouts about which the user application has no prior knowledge. Since this needs run time function redirection, we use the late binding (virtual functions) mechanism provided by C++. This gives a very fast implementation of data access while maintaining the flexibility we desire.

In certain cases however, virtual function calls might not be fast enough. It is well known that virtual function calls are slightly slower than direct function calls due to pointer dereferencing. Direct function calls however are generated by the compiler at compile-time only. To provide data access using direct calls, the prior knowledge of the data layout would be needed. We have designed the data access mechanism such that the prior knowledge of data layout will give the user application the maximum speed via direct function calls, thus avoiding virtual calls. In Figure 8 we show the options the user has to access data. If the speed of virtual function calls is enough, the user can use the *VirtualCall* iterator to access the volume data. This iterator automatically calls the correct iterator functions at run time using a virtual function call. In case direct function calls are desired, the user can call the appropriate iterator directly as shown by the solid arrows.

There is also a third option which can combine the two previous options. The user can use a macro defined in OpenVL to call the appropriate iterator directly, or in case the iterator is unavailable, the macro will call the VirtualCall iterator. This type of data access will be fast for data layouts which provide iterator implementations at run time and slow for other layouts which do not. However, this has the drawback of adding some code bloat to the user application because every routine will be implemented for each supported layout type and data type. We suggest that this method be used only where there is a significant performance hit when compared to using the VirtualCall iterator.

6.5. Hiding Templates from Volume API

Although we use templates inside OpenVL, we prefer to hide them wherever we can. The reason for this is to provide a clean API. Addition of a template parameter into an interface gives multiple interfaces for each instance of the template parameter. This can be highly undesirable when designing an API for an application based on OpenVL.

We chose to hide templates completely from our Volume API (Figure 2). Any user's program using our volume objects is likely to have function declarations or interface declarations in which the volume object is passed as an argument. For example, consider a function `foo` declared as `foo(vlVolume * vol)`. If we do not hide the template from our volume API, the user will have to declare a templatized function such as `foo(vlVolume<DataType> * vol)` which will eventually convert to multiple functions, one for each supported data type. This can be highly undesirable when the function is part of an interface.

To accomplish template hiding, we introduce a base class to the Volume Data Layout API which is non-templatized. A pointer of this base class is stored in the `vlVolume` class which at run time points to the correct data layout. In addition, the Volume API consists of functions which give information about the layout being used and the data type of the volume data. This information can be used to select the appropriate iterator for accessing the data.

7. Example Code

In this section, we present example code to show how OpenVL is used.

Creating volume object: In the first example we show how to create a volume object. The following code creates a volume with dimensions $50 \times 40 \times 20$, and voxel data type of unsigned char (8 bit unsigned).

```
vlVolume *vol = new
vlVolume(vlDim(50,40,20), UnsignedInt8);
```

File input/output: To read and write a volume from/to a file, the volume API has to be used. The following code will read volumetric data stored in file `sample.slc` into the volume object `vol` and then write the volume data in raw format to another file `sample.raw`.

```
vlVolume *vol = new vlVolume();
vol->read("sample.slc");
vol->write("sample.raw", "Raw");
```

Accessing volume information: Information about the volume stored in a volume can be obtained using the volume API.

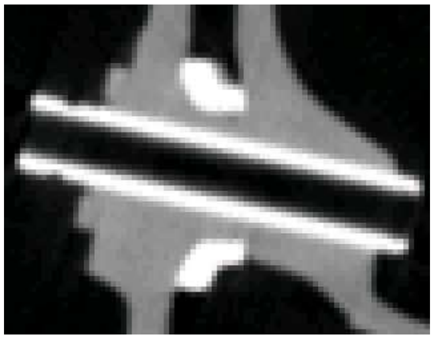


Figure 9: *CT cross-section of engine dataset.*

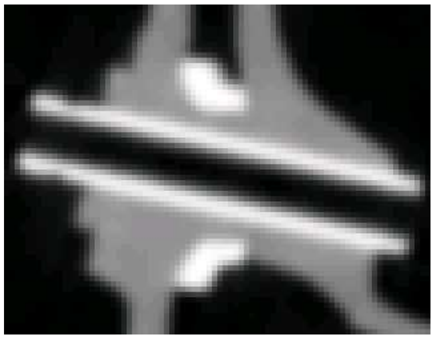


Figure 10: *Gaussian filter applied to the engine dataset (Figure 9) using OpenVL.*

```
cout « "Dimensions : " « vol->dim();
cout « "Voxel units : " « vol->units();
cout « "Bytes per voxel:" «
          vol->bytesPerVoxel();
```

Accessing volume data: The following code uses the Iterator API to access the voxels stored in a volume. This sample code goes over the volume once and counts the number of non-zero voxels in the volume.

```
vlVolume *vol = new
vlVolume(vlDim(50,40,20), UnsignedInt8);
vlVolIter<uint8> iter(vol);
uint32 count(0);
while(!iter.end()) {
   if(iter.get() != 0) ++count;
   ++iter;
}
cout « "No. of non-zero voxels : " « count;
```

Performing a volume processing task: The following code shows the use of the volume processing component of OpenVL. Here we perform thresholding on a volume using the *Thresholder* plugin included with OpenVL. After the thresholding is done, it then writes the output to another file.

```
vlVolume *vol = new vlVolume();
vol->read("sample.slc");
vlPlugin *proc=
  vlKernel::trader()->getPlugin(
     "VolProcessor", "Thresholder");
if (!proc) return;
proc->setVolume(vol);
proc->config()->set("ThresholdLow",
   (uint8)(100));
proc->config()->set("ThresholdHigh",
   (uint8)(150));
if(!proc->run()) return;
cout « "Thresholding done." « endl;
vol->write("sample_thresholded.slc");
```

More complete examples are available for downloading from the OpenVL website. Figure 10 shows the results after applying the *GaussianApprox* volume processor to the CT engine dataset of Figure 9. Color Plate 1 shows some more results from volume processing using OpenVL.

We have also developed entire applications based on OpenVL. One such application is the slice editor shown in Color Plate 2. The application uses all the functionality provided by OpenVL. It uses the `vlVolume` objects to represent volumes. It loads volume data from any file format supported by OpenVL and uses the Volume API to do so. It also makes use of the utility classes provided by OpenVL. For example, the slice view on the left uses the `vlSlice` class and the transfer function on the right uses the `vlTxFunction` class. This application is thus merely a frontend to the backend functionality provided by OpenVL.

8. Conclusions and Future Work

In this paper we presented a library which provides a uniform and standard framework for handling volumetric data. The library includes a standard and uniform iterator style API for accessing volumetric data, different layouts for volumetric data, and framework for implementing various volume processing tasks. The library is designed and built to be fast, extensible, modular, and easy to use.

Almost everything in the library is built as plugins which are binary shared object files that can be loaded and used at run time. This allows dynamically extending the functionality provided by OpenVL, making the library extensible. Since all the plugins are simple files, they can be added or removed to control the functionality provided by OpenVL. This gives OpenVL a modular structure.

All the APIs in OpenVL are clean, simple, and well documented. A reference documentation is always available on the OpenVL website. This makes it easy to learn and use the library. To provide a clean and simple API, we hide the internal use of C++ templates from the user. This also has the advantage of controlling the size of the library. With extensive use of templates, it is possible for the run time size of the library to grow exponentially.

OpenVL supports multiple file formats for volumetric data storage in files. This is achieved through the use of plugins. Each file format has a plugin which provides the input/output functionality for that format. Since these plugins are dynamic, existing applications using OpenVL can make use of new plugins at run time, without recompiling.

The implementation of OpenVL uses modern C++ techniques, such as templates, partial template specialization, code inlining, etc. This results in a high performance and flexible implementation. The library is developed as a free software and encourages contribution. The entire source code is readily available on the web.

Our next goal with OpenVL is to provide as much functionality as possible. This includes adding plugins for various volume processing tasks, file formats, and data layouts. We also aim to add more utility classes to the library.

In the future, we would also like to extend OpenVL to include a visualization framework just like the volume processing framework we have now. For this, we would like to add a volume rendering API (for example, VLI [11]) and a volume modeling API with plugin support for different rendering engines and modeling methods, respectively. We also plan to add user interface widgets to the library. This will allow easy integration of the functionality provided by the library into user's application interfaces.

9. Acknowledgements

This work has been partially supported by ONR grant N000140110034, NIH grant CA82402, a CAT Biotechnology grant, and a NYSTAR grant. The engine dataset is courtesy of GE. The authors wish to thank Manjushree Lakare, Klaus Mueller, Suzanne Yoakum-Stover, and Susan Frank for their help, encouragement and discussions. We also thank Sourceforge.net for providing CVS code repository, mailing lists and initial WWW hosting for our project. The OpenVL website can be currently accessed at http://openvl.sf.net. More information about OpenVL can be found at http://www.cs.sunysb.edu/∼vislab/projects/openvl.

References

1. R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A Diversified Volume Visualization System. In *Proc. of IEEE Visualization '94*, pages 31–38, Oct. 1994.

2. M. Bosc and T. Vik. The ImLib3D Homepage, 2002.

3. M. Cox and D. Ellsworth. Application-Controlled Demand Paging for Out-of-Core Visualization. In *Proc. of IEEE Visualization '97*, pages 235–244, Oct. 1997.

4. J. E. Fowler and R. Yagel. Lossless Compression of Volume Data. In *Proc. of IEEE Visualization '94*, pages 43–50, Oct. 1994.

5. R. M. Haralick and L. G. Shapiro. Image Segmentation Techniques. *Computer Vision, Graphics, and Image Processing*, 29(1):100–132, Jan. 1985.

6. IBM Corp., Armonk, NY, USA. *Data Explorer Reference Manual*, 1991.

7. Insight Consortium, http://www.itk.org. *The Insight Segmentation and Registration Toolkit (ITK) Website*, 2002.

8. K. Konstantinides and J. Rasure. The Khoros Software Development Environment for Image and Signal Processing. *IEEE Transactions on Image Processing*, 3:243–252, May 1994.

9. P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. *Computer Graphics*, 28:451–458, 1994.

10. N. R. Pal and S. K. Pal. A Review on Image Segmentation Techniques. *Pattern Recognition*, 26(9):1277–1294, 1993.

11. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *Computer Graphics, SIGGRAPH 99*, pages 251–260, Aug. 1999.

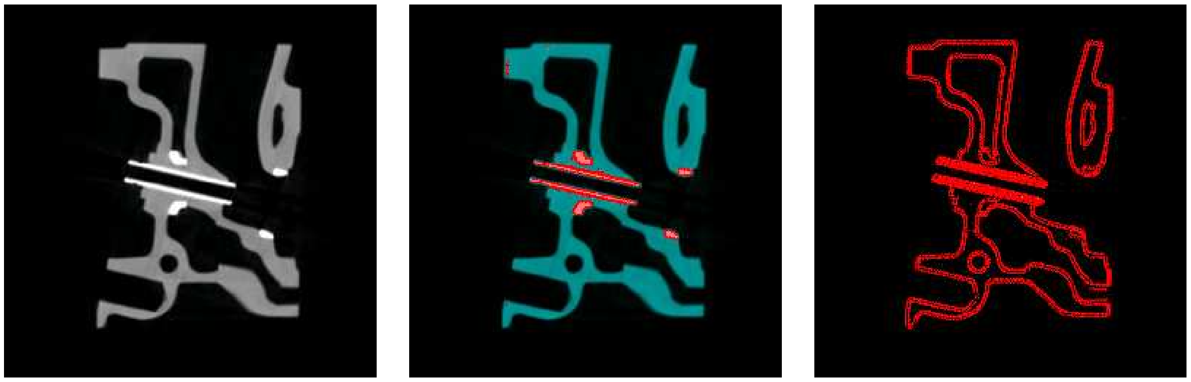
12. W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Prentice Hall, 2 edition, 1998.

13. W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *Proc. of IEEE Visualization '96*, pages 93–100, 1996.

14. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

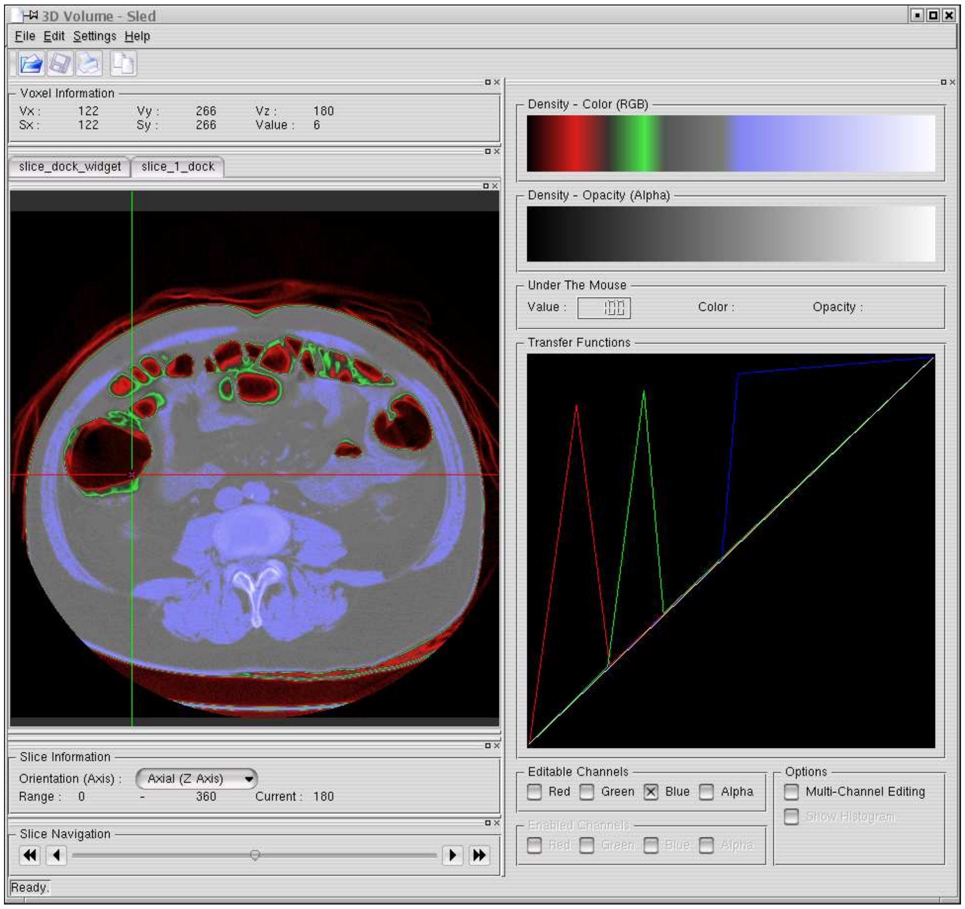
15. C. Upson, T. Faulhaber, D. Kamins, D. Schlegel, D. Laidlaw, F. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, July 1989.

16. C. Yang and T. Chiueh. I/O-Conscious Volume Rendering. In *Joint Eurographics - IEEE TCVG Symposium on Visualization*, pages 263–272, May 2001.



(a) Original CT cross-section (b) Thresholding (c) Edge detection

Color Plate 1: Results of volume processing performed on the engine dataset using OpenVL.



Voxel Information
Vx : 122 Vy : 266 Vz : 180
Sx : 122 Sy : 266 Value : 6

Under The Mouse
Value : Color : Opacity :

Slice Information
Orientation (Axis) : Axial (Z Axis)
Range : 0 - 360 Current : 180

Editable Channels
☐ Red ☐ Green ☒ Blue ☐ Alpha

Options
☐ Multi-Channel Editing
☐ Show Histogram

Enabled Channels
☐ Red ☐ Green ☐ Blue ☐ Alpha

Ready.

Color Plate 2: A slice editor application built using OpenVL.

© The Eurographics Association 2003.