OpenVL User Manual

Sarang Lakare[1]

Jan 15, 2003

Revision : 1.8

[1]lsarang@cs.sunysb.edu

Contents

Chapter 1

Obtaining OpenVL

In this section we will first describe the versioning scheme used by OpenVL which will help you understand which version of OpenVL to use. We will then show how to download and get a working version of OpenVL installed on your system. We will also describe the directory structure used by OpenVL.

1.1 Understanding the version numbers

OpenVL's version number has the following format: "a.b.c", where a, b and c are some non-negetive integers. The first number "a" is the major version number, "b" is the minor version number and "c" is the revision number.

We follow a scheme which is similar to the one used by Linux kernels. In our scheme, all unstable or developmental[1] releases are characterized by an *odd* minor version number. That is, for any numbers a and c, a.1.c will always be an unstable release and a.2.c will always be a stable release.

The major version number indicates the API compatibility of the library. All versions of the library with the same major version number will be binary forward compatible. That is, version 0.2.1 will be forward compatible with version 0.4.0. Any program comiled with version 0.2.1 of the library will work with version 0.4.0 or with any version 0.b.c without needing a recompile. There is one exception however. The unstable releases are not guaranteed to be forward compatible.

The revision number among stable releases indicates a bug-fix release which is 100% compatible with the earlier releases with the same major and minor numbers. For unstable releases, the revision number simply indicates a release from the main branch of CVS repository.

We gave a version number of 0.1.0 to our first release. This release is an unstable release since the minor number is odd. The stable release following the unstable releases 0.1.0, 0.1.1 etc. will be version 0.2.0. Once a stable version is released, another unstable version 0.3.0 will be released. The unstable versions are always on the the HEAD[2] branch of CVS. If there are any bug fixes to be applied to 0.2.0, a new branch for 0.2.0 will be opened and the fixes will be applied there. A new release 0.2.1 will then be released. At the same time, we will release new unstable versions 0.3.1, 0.3.2 as more changes occur in CVS. Once the 0.3.x series is stabilized, we will release version 0.4.0 and continue work on 0.5.x series.

[1] Those releases which are meant only for developers and not for end-users.

[2] HEAD is the main CVS branch

The major number will be upgraded when we need to make binary incompatible changes to the OpenVL API. The first time we do that, the new version will be 1.a.b. After that, we will continue release 1.a.b releases till we again make binary incompatible changes which will lead to 2.a.b releases.

To summarize, if you are an end-user who wants to use the OpenVL library for your software, stick to stable releases which will have an even minor version number. If you are interested in living on the bleeding edge, use unstable releases or CVS.

1.2 Downloading

All the software that we release will be available at this central location:

http://sourceforge.net/project/showfiles.php?group_id=24923

We will always provide tarred source files compressed using the gzip and the bzip2 algorithm. Along with sources, we will also provide a source RPM file which you can use to generate binary RPM files for your distribution.

We will always try to provide binary RPM files. As of now, we are providing RPM files for Linux Mandrake 9.0. This RPM file depends only on gcc 3.2 and should thus be installable on any RPM based distribution which uses gcc 3.2.x. Please provide us feedback so that we can provide binary RPMs which can install on all Linux distributions out there.

Our naming conventions for RPMs is similar to the one used by MandrakeSoft. For example, the RPM files for version 0.1.1 are:

libopenvl0-0.1.1-1sar.rpm
libopenvl0-devel-0.1.1-1sar.rpm
libopenvl0-plugins-0.1.1-1sar.rpm

There are three RPM files for each release of OpenVL. The first is the core library. The second includes the header files needed for writing software based on OpenVL. The third RPM contains the plugins that come with OpenVL. To use OpenVL, you will need the first and third RPMs installed. To develop software using OpenVL, you will need the -devel package. The "0" suffix to libopenvl indicates the major version number of OpenVL. OpenVL packages of version 1.b.c will be named libopenvl1. This type of naming makes it easy to have two different version of the library installed on the system and also help in easy upgrading of the operating system.

1.3 Compiling and installing from sources

The first step is to uncompress the source archieve. If you have downloaded the `.tar.gz` package, then run:

```
tar -xzvf package.tar.gz
```

and the package will be uncompressed into a subdirectory. If you have downloaded the `.tar.bz2` package, then you need to run:

```
tar -xjvf package.tar.bz2
```

to uncompress. You can also uncompress from inside the konqueror file manager (right click and select "extract") or using software like `Ark`. Once you have uncompressed, go into the directory and run the following command:

```
./configure --enable-plugins
```

The `--enable-plugins` option is needed to enable the compilation of plugins. Otherwise you can go into the *plugins* subdirectory and run make there to compile the plugins separately. You can also use the option `--prefix` to specify the directory in which you want to install OpenVL. For example, if you do not have root access (that is, administrative access) to your machine, you might want to install OpenVL in `$HOME/openvl` directory. In that case, give `--prefix=$HOME/openvl` option to the `./configure` command. By default, the installation directory is `/usr/local`.

If you see any errors, you probably do not have all the software needed by OpenVL. Carefully read the error message and install the required software. You can also check the *config.log* file for the exact location of the error. If you are unable to figure out the problem after really trying hard, then post your problem on the openvl-devel mailing list and someone will answer. If the configuration ran without any errors, then run the following:

```
make
```

If you get any errors, try to find out the reason. If you cannot, then post the output to the mailing list. To install, run the following command:

```
make install
```

This will install OpenVL in the directory you specified. If you are using the default installation directory or want to install OpenVL in a directory which is owned by root, you will have to first login as root by typing `su` on the command prompt before running the above command.

1.4 Installing binary packages

We provide binary RPM packages for OpenVL. To install, download the RPM packages. Then run

```
rpm -ivh *.rpm
```

in a directory which contains only the OpenVL packages. If you are upgrading from an earlier version of OpenVL already installed on your system, then use:

```
rpm -Uvh *.rpm
```

This removes earlier versions of OpenVL and installs the new version.

You can also install/upgrade from inside the konqueror file manager. Just click on the RPM packages you downloaded one by one and install/upgrade them. You will always need to install/upgrade the *libopenvl* package before the *libopenvl-devel* and *libopenvl-plugins* packages.

Note that it is possible to install multiple unstable releases of OpenVL with different major and minor version numbers at the same time. However, you can only have one installation of unstable releases with the same major and minor version numbers. Similarly, you can only have one installation of stable releases with same major version number. You will only be able to upgrade between these releases. Since these releases maintain binary compatibility, an upgrade should not cause any problems.

1.5 Understanding the OpenVL directory structure

The OpenVL directory structure is present under the `prefix` directory that you specify during installation. The core library is installed in the `lib` directory. The include files are installed in the `include` directory. In addition to these, OpenVL creates a new directory for installing plugins. This new directory resides in a directory named `openvl` under the `prefix/lib` directory. The name

for the plugins directory depends on the version number of OpenVL to which the plugins belong. For a release of OpenVL with version a.b.c, the plugin directory will be:

`plugins-a.x` for stable releases. "x" is the x alphabet.

Each stable release with different major number will have its own directory for installing plugins. This way plugins from different OpenVL installations with different major numbers do not interfere with one another.

`plugins-a.b` for unstable releases.

Each unstable release with different major or minor number will have its own directory for installing plugins. This way the plugins of unstable releases do not interfere with plugins from stable releases or those from other unstable releases.

For example, plugins of OpenVL version 0.1.1 go in the directory

`prefix/lib/openvl/plugins-0.1`

and plugins of OpenVL version 0.3.3 go in the directory

`prefix/lib/openvl/plugins-0.3`

Similarly plugins of OpenVL version 0.2.0, 0.4.0, 0.4.1 etc will go in the directory

`prefix/lib/openvl/plugins-0.x`

Every user can also install plugins in his or her home directory. This directory has the same name derived from the OpenVL version number as above under `$HOME/.openvl` directory. Thus for an user, the plugin directory for OpenVL version 0.2.1 will be

`$HOME/.openvl/plugins-0.x`

where (s)he can install additional plugins.

Chapter 2

Getting Started

In this section we will get you started with using OpenVL.

2.1 "Hello World! I am a Volume"

Here we show simple usage of OpenVL.

Creating volume object: In the first example we show how to create a volume object. The following code will create 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.

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

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), vlUnsignedChar);
vlVolIter<uint8> iter(vol);
uint32 count(0);
while(!iter.end()) {
    if(iter.get() != 0)
      ++count;
    ++iter;
}
std::cout << "Number of non-zero voxels :  " << count << std::endl;
```

2.2 Reading/Writing of Volumetric Data Files

In this section we show how you can read from a file which contains volumetric data and how to write a volumetric data to a file on disk. OpenVL supports reading/writing in multiple file formats. File I/O for each file format is implemented as a dynamic plugin. OpenVL will automatically find and load these dynamic plugins and will perform File I/O for the specified format using the plugins. Thus, the supported file formats can be dynamically extended in OpenVL. The general syntax for reading from a data file is:

```
vlVolume vol;
vol.read("myfile.extension");
```

This will read `myfile.extension` and load the data stored in that file into the volume object `vol`. The datatype and the size of the `vol` is automatically changed to match that of the volume being loaded. Any data and other information stored in `vol` earlier is lost. The file format of the `myfile.extension` file is automatically detected by OpenVL using the available File I/O plugins. This is done by querying each of the available plugins to check if they are able to handle the file. The first plugin to say it can handle the file format gets to do it.

The syntax to write a volumetric data to a file is similar to that of reading:

```
vol.write("myfile.extension");
```

This will write the data stored in the volume object `vol`. The file format in which to write is automatically detected using the `extension` given to the filename. You can also explicitly specify which file format to use when storing the file by specifying it as the second argument to the `write` function call. For example:

```
vol.write("myfile.extension", "VolVisSLC");
```

This will write to file `myfile.extension` in the VolVis SLC format. If no plugin is found which can write in this format, then the function will return `false`.

2.2.1 Reading RAW Data Files

Inspite of the file i/o plugin support, it is possible that there is no plugin for the format in which you have some data. In order to read this data, you have two options: either write a file i/o plugin for that file format, or extract the raw data from the file and then load it using OpenVL's raw file i/o plugin. It could also be that you simply prefer to use raw data files. In all these cases, you can make usre of the raw file i/o plugin.

In order to read from a raw file, you need to provide some information about the data. The minimum information that is needed is the dimensions and the size of each voxel in bytes. There are two ways to provide this information.

1. **In a separate file:** You can specify the needed information in a separate file. If the name of your raw data file is `myfile.raw`, then the file that the raw file i/o plugin looks for by default is `myfile.inf`. On the first line of this file, specify the dimensions of the volume as separate integers. On the second line, specify the size of each voxels in bytes. For example, if your raw file is of size $50 \times 50 \times 30$ and containes unsigned short data, then your `.inf` file would look like:

   ```
   50 50 30
   2
   ```

The main advantage of this method is that you can make any OpenVL based application read a raw file. Simply create a file with the required information and the application will automatically read the file.

2. **Coding the information:** The second option is to specify the information in the code itself. This can be done by using another version of the `read()` function call. First create an object of class `vlVolInfo` and set the relevent information (mainly dimensions, bytes per voxel, and data type). Then pass this object to the `read()` function call:

```
vlVolInfo info;
info.setDim(vlDim(50, 50, 30));
info.setBytesPerVoxel(2);
info.setDataType(UnsignedInt16);
vol.read("myfile.raw", info, "RAW");
```

This technique also has its uses. For example, you can program your application to ask the user for information regarding a raw volume that they are trying to load. This can save your users the trouble of writing an extra info file for each volume file they have.

2.3 Accessing Volumetric Data using Iterators

In this section we will show how to access the volumetric data effectively using iterators. Iterator's are OpenVL's mechanism for giving access to the volumetric data stored in a volume. The iterators used in OpenVL are similar in concept to the generic iterators used in the standard template library (STL), but are very different in implementation and usage. In OpenVL, the iterator is basically a class that has direct access to the volumetric data stored inside the volume. The iterator interface is very elaborate compared to the generic iterators used by STL. We have chosen an elaborate interface because we feel that accessing a volumetric data is a complex task compared to accessing the data that STL usually stores.

The iterator interface is properly documented and is available under *API Reference* from the OpenVL website. In this document we will look at some commonly used iterator functionality.

2.3.1 Initializing an Iterator

Before initializing an OpenVL iterator, you need to know the datatype of the data stored in the volume. That is, you need to know if the data is unsigned char, short, float etc. For example, if the data is 8 bit unsigned, then the usual way of initializing an iterator would be:

```
// For a volume defined as vlVolume *vol
vlVolIter<uint8> iter(vol);
```

The default constructor of the iterator expects a pointer to a `vlVolume` object on which the iterator will be initialized[1]. After initialization, the iterator position is always set to the start of the volume data. The start of the volume data is the first non-zero voxel in the data and need not mean the position $(0, 0, 0)$.

[1] It should be noted that this initialization is different from the way STL iterators are initialized.

2.3.2 Accessing Voxel Data

At any instance of an iterator, the value stored in the voxel pointed to by the iterator can be accessed using:

```
data = iter.get();
```

The value returned by the iterator is of the same datatype that you initialized the iterator with (and hence the datatype of the volume). Similarly, you can set the value of the voxel pointed to by an iterator using:

```
iter.set(data);
```

In addition to accessing value at a voxel, the iterator API provides functionality to access the neighborhood of the voxel. The following code shows some of the functions that you can use:

```
// Get the value of the next voxel along X
data = iter.getRelativeX(1);
// Get the value of the next voxel along Y
data = iter.getRelativeY(1);
// Get the value of the prevous voxel along Z
data = iter.getRelativeZ(-1);
// Get the value of the voxel 2 position to the left along X
data = iter.getRelativeX(-2);
// Get the value of the voxel 3 positions to the right along Z
data = iter.getRelativeZ(3);
// Get the value of the voxel at a position (2, -1 , -1) with respect to current position
data = iter.getRelative(vlPoint3i(2, -1, -1));
```

2.3.3 Moving Iterator Around

The iterator can be moved around from the position it is at a certain instance. Functionality is provided to move the iterator to any of its neighboring voxels. The simplest function is `next()` which moves the iterator to the next voxel in the volume. The important point to note about `next()` is that the voxel pointed to by the iterator after a call to `next()` might not be a neighborhood of the earlier voxel. That is, `next()` does not specify which voxel it will move the iterator to. It is simply the fastest way to access all the voxels in a volume.

If you want to move the iterator to a voxel position in the neighborhood of the current voxel, you can use the following:

```
// Move to the next voxel along X. If there are no more voxels
// along X, then move along Y, then Z ...
iter.nextXYZ();
// Move to the next voxel along Z
iter.nextZXY();
// Move to previous voxel along Y
iter.prevYZX();
```

The iterator can also be moved to any voxel that is a relative position away from the current. For example, to move the iterator to another voxel at say $(3, -1, 2)$ from the current position:

```
iter.moveRelative(vlPoint3i(3, -1, 2));
```

The iterator can also be moved to any position in a volume using:

```
bool ret = iter.moveTo(vlPoint3ui(10,5,8));
if (ret == false)
  std::cout << "moveTo failed.  Moving outside volume?" << std::endl;
```

The `moveTo` function takes a position in the volume coordinates and moves the iterator to the new position if the position is valid. That is, the given position must be inside the bounds of the volume. If it fails to move, it returns `false`, else it returns `true`. It should be noted that moving iterators may not be very effecient and should be avoided as much as possible. The OpenVL iterators are always optimized for accessing and moving within the immediate neighborhood of a voxel.

2.4 Using Volume Processing Plugins

In this section we will describe how to use volume processing plugins. The first step is to find out which plugin you want to use. All the available plugins are listed under the "Plugin Document" link from OpenVL's main page.

Suppose you want to perform region growing on your volume. You will need the plugin *RegionGrow* for that. The first step is to get an object of the plugin. This can be done by querying OpenVL for the plugin: `vlKernel::trader()->getPlugin("VolProcessor", "RegionGrow");`

or you can also query using: `vlVolProcessor *proc = vlKernel::trader()->getVolProcessor("RegionGrow"`

Once this is done, make sure you have a valid plugin by looking at the proc pointer. It should be non-null. If the plugin is loaded, then you need to set the volume on which this plugin will run.
`proc->setVolume(&vol);`

After this, you need to configure the plugin. The configuration depends on the plugin you are using. Consult the plugin documentation for the parameters that you can configure. For e.g., the *RegionGrow* plugin allows you to specify the thresholds, the mask volume, the start point etc. Here we configure these:

```
proc->config()->set("thresholdLow", (uint8)100);
proc->config()->set("thresholdHigh", (uint8)160);
proc->config()->set("insideMask", (uint8)100);
proc->config()->set("maskVolume", &maskVol);
proc->config()->set("startPosition", startPos);
```

Now, all you need to do is run the plugin:

```
if(proc->run())
  cout << "Success" << endl;
else
  cout << "Failure" << endl;
```

Thats it folks! You ran the plugin! Here is a sample code that you would have to write to use one of the OpenVL plugins (in this case, the *RegionGrow* plugin) :

```
vlVolume vol;
// Query the processor
vlVolProcessor *proc = vlKernel::trader()->getVolProcessor("RegionGrow");
if(proc) {
  cout << "Loaded Vol Processor :  " << proc->info().name() << endl;
```

```
} else {
  cout << "Vol Processor with name regionGrow not found." << endl;
  return (-1);
}
// Read volume from data file - format autodetected
vol.read("smallLobster256.slc");
// Create a mask volume
vlVolume maskVol(vol.dim());
vlPoint3ui startPos(120, 140, 43);
// Set the volume on which to run the processor
proc->setVolume(&vol);
// Configure the plugin
proc->config()->set("thresholdLow", (uint8)100);
proc->config()->set("thresholdHigh", (uint8)160);
proc->config()->set("insideMask", (uint8)100);
proc->config()->set("maskVolume", &maskVol);
proc->config()->set("startPosition", startPos);
// Run the processor
if(proc->run())
  cout << "Success" << endl;
else
  cout << "Failure" << endl;
// Print results
cout << "Results :  " << endl;
uint64 voxelCount;
const vector *boundaryVoxels;
proc->results()->get("voxelCount", voxelCount);
proc->results()->getPtr("boundaryVoxels", boundaryVoxels);
cout << "in the region :  " << voxelCount << endl;
cout << "voxel count :  " << boundaryVoxels->size() << endl;
// write the mask volume to disk - format autodetected
maskVol.write("output_mask.slc");
```

Chapter 3

Writing Plugins

3.1 Volume Processing Plugins

Volume processing plugins are those that perform a certain task on a given volume. For e.g., thresholding would be considered a volume processor (similar to image processing). We have provided a sample plugin which you can use as skeleton for your plugin. Download it from:

`http://openvl.sourceforge.net/pub/sample_plugin.tgz`

Go to a directory where you have write access, and untar the file using:

`tar -xzvf sample_plugin.tgz`

Inside the directory you will find myplugin.h and myplugin.cpp files. These are the only files that you need to implement a volume processing plugin.

The header file: You do not need to modify the header file at all unless you want to change the name of the class MyPlugin. The plugin factory can stay the way it is.

The cpp file: At the top of the plugin file you will see:

```
VL_EXPORT_COMPONENT_FACTORY( myplugin, PluginFactory )
```

Here, *myplugin* indicates the name given to the plugin file. The plugin file name should always be the first entry in the above macro prefixed by `vl` and suffixed by `.so`. Thus, in this case, the plugin file should be `vlmyplugin.so`. To change the name from myplugin, change the name in the above macro, and then edit the Makefile and change the `$TARGET` field. You will have to edit the `$CPPFILE` and `$HEADERFILE` fields too if you decide to name your cpp and h file something other than `$TARGET.cpp` and `$TARGET.h` respectively.

Every plugin of OpenVL is recognized by two fileds: the service that it provides, and the service group to which this service belongs. The service group for volume processing plugins is set to *VolProcessor*. Inside your plugin, you need to specify the service that your plugin will provide. The users of your plugin will use the service name you give to your plugin to query for your plugin.

In the constructor of your plugin class, specify the information about the plugin. For e.g., use `setVersion(...)` to set the version, `setService(...)` to set the service the plugin provides.

The `run()` method is the starting point of the plugin. Whenever a user will request this plugin, `run()` will be called on the volume set by the user. The 3D volume itself can be accessed using `vol()` function. `vol()` returns the pointer to the volume. Any configuration parameters set by the user can be accessed using `configRef()`. This function returns the reference to a vlVarList object which stores the configuration. Use the object returned by `resultsRef()` to store the results which can be accessed by the users.

In addition to `run()`, there is another method which you can override. It is the `init()` function. This function is called whenever the user calls `setVolume(...)`. This is the right place to setup the configuration variables which are dependent on the data type of the volume.

Compiling the plugin: Simply type make to compile the plugin.

Installing the plugin: The makefile provided allows you to locally install the plugin. Plugins can be locally installed in the plugins directory under `$HOME/.openvl`. The name of the plugin directory will depend on the version of OpenVL you are using (See Section 1.5). Set the correct name of the plugin directory in the makefile. After that, to install the plugin type:

```
make inst
```

Make sure that the plugin directory exists before running `make inst`. You can check if the plugin was properly installed by checking the contents of the plugin directory.

3.2 File Input/Output Plugins

3.3 Volume Data Layout Plugins

3.4 Interpolation Plugins