Programming Guide

Data Management for Computing with VolumeViz™ LDM 8.1

Data Management for Computing with VolumeViz LDM 8.1

Section 1 – Introduction

Section 1.1 – Overview

We assume that you are already somewhat familiar with Open Inventor and VolumeViz programming. You should have read the VolumeViz chapter in the standard Users Guide (available as a PDF file in the Open Inventor installation directory tree).

VolumeViz extends the Open Inventor toolkit with a powerful set of tools for visualizing volume data. These tools include axis aligned slices, arbitrary slices, subvolume probes, volume rendering, volume combining, state of the art GPU based rendering techniques and an extensible framework for GPU programming. VolumeViz LDM (Large Data Management) is a data management extension that allows navigation and rendering of extremely large volume data sets. LDM provides fast display of initial images, consistent interactive response, automatic image refinement and multi-threaded background data loading.

VolumeViz LDM has changed the rules for visualization of very large volume data sets. It's no longer necessary to have, and schedule time on, a cluster in order to do visualization of large data sets. Using LDM multiple users can access a large data set simultaneously and work with it on their desktop or even laptop machines. This has made 3D visualization of large volumes feasible for a vastly larger group of both users and application developers. Seeing the success of VolumeViz LDM for visualization has greatly increased the interest in integrating LDM formatted data into other parts of the workflow. For example, storing computed data directly into LDM format could reduce total disk storage and make the data available for visualization much more quickly. Being able to compute and immediately visualize a subset of the data could increase productivity by allowing more iterations in the same amount of time.

In a typical scenario the goal is to load some data, perform some computation and save the result as quickly and efficiently as possible. We may visualize the data later, but computational performance is the highest priority. We might be modifying the data, for example applying a noise reduction filter, or we might be deriving a new data set, for example computing an attribute volume. For this scenario our first concern is data bandwidth, in other words how quickly can we get the data into a block that can be efficiently handled by our computing platform. If the data resides on disk or across the network the total processing time may be dominated by data transfers. VolumeViz LDM provides a powerful data access API allowing applications to take advantage of Large Data Management for accessing data associated with a sub-volume, plane, point or polyline at arbitrary resolution. This makes VolumeViz LDM not just a visualization toolkit but also a powerful middleware for volume data management.

The data access API allows an application to get the data associated with a specific tile at a specific resolution level. More importantly it allows an application to ignore tiling and request the data values in a trace, a slice or an arbitrary subvolume. VolumeViz LDM takes care of copying data from all the required tiles to form the requested block of data. This allows the application to adjust the requested block size to accommodate available memory, match requirements of the algorithm or to optimize for the computation platform. Whether the actual computation will be done using CPUs, GPUs or something else (for example a Cell BE processor), the data management framework will be generally be similar and follow this guide.

In this guide we will discuss how to efficiently load LDM data using synchronous and asynchronous calls, how to create a new LDM data file on disk, and how to create an LDM volume in memory.

Section 1.2 – Outline

Section 1.3 – Compatibility and New Features

NOTE: This guide and the accompanying example code were originally written for Open Inventor version 7.0. However both the text and the code have been updated to take advantage of new features added in Open Inventor 8.0 and the current examples will not compile with older versions of Open Inventor. If you are using an older version of Open Inventor you should upgrade to the current version.

The most significant change for Open Inventor 8.0 is that, where appropriate, the examples now use the new methods that take an *SoBufferObject* pointer instead of a direct pointer to a native data type. The old methods are still supported but will generate a "deprecated" compiler warning. We recommend updating to use the new methods.

Open Inventor now provides a computing framework that allows you to do many operations without writing low level code to access various devices. The SoBufferObject subclasses *SoCpuBufferObject*, *SoGLBufferObject*, *SoCudaBufferObject* and *SoOpenCLBufferObject* allow you to easily move data between the various devices you may be using for computation. The *SoCuda*, *SoCudaDevice* and *SoCudaContext* classes provide a convenient way to query and manage the CUDA computing devices in your system. There are corresponding classes for OpenCL (since Open Inventor 8.1). In addition to making it easier to integrate your own algorithms, Open Inventor includes optimized implementations of some commonly used algorithms. This means you can use the computing power of the GPU without writing a single line of CUDA code. The *SoAlgorithms* class loads algorithm modules and provides convenient methods to get algorithm interfaces, contexts and buffers appropriate for the implementing device. For each algorithm module there is a default CPU implementation plus corresponding CUDA and OpenCL implementations. For example, *SoConvolution* and *SoCudaConvolution*. The classes are:

- SoArithmetic : Add, multiply, combine, compute min/max, etc.
- SoConversion : Convert from one data type to another, e.g. unsigned char to float.
- SoConvolution : 1D and 2D separable convolution.
- SoDataExtract : Extract a slice of data from a tile.
- SoSeismic : Compute Hilbert transform and some basic seismic attributes.

Open Inventor also provides some very useful utility functions in the *SoDevice* classes. For example, *SoCpuDevice* provides a convenient, portable way to query available system memory. This is useful when you need to set the maximum system memory that VolumeViz LDM is allowed to use. For example:

```
// Get total memory installed on this CPU
SoCpuDevice *pCPU = SoCpuDevice::findFirstAvailableDevice();
uint64_t totalMem = pCPU->getTotalMemory();
```


Section 2 – Loading LDM Data

Section 2.1 – Performance

Now let's suppose that we need to load all or a substantial part of a volume and do some computation. Let's also assume that we're integrating LDM format data into our workflow. This includes any LDM-like data format that can be handled by VolumeViz through a custom VolumeReader class. By using the VolumeReader interface we can benefit from the LDM data manager even for proprietary data formats.

In any case where the data set is large or the computation is expensive we will be concerned about the total time to complete the operation. This is particularly true if the operation is intended to be interactive, but is true even for batch computations. As with any optimization task it is important to benchmark and determine which steps are taking the most time. Remember that the total time to complete the operation includes loading the data, computing the result and storing the result (or at least making it available for subsequent operations like visualization). Don't focus too much on the computation time until you know that effort will pay off.

> Note that Open Inventor provides the utility class *SbElapsedTime* for accurately measuring elapsed time (using a high resolution timer on platforms where this is available). For example:

```
// Create, use and re-use a timer
SbElapsedTime timer;
   // ... First operation to be timed ...
double time1 = timer.getElapsed(); // Time in seconds
timer.reset();
   // ... Second operation to be timed ...
double time2 = timer.getElapsed(); // Time in seconds
```

Performance will be limited by the available hardware resources. We will not discuss hardware after this section, but it may be valuable to briefly consider a few issues. There are two general issues. First, no matter how much optimization you do, the software cannot run faster than the hardware allows. So if you get to choose the hardware, get the fastest / biggest hardware you can afford. We'll talk briefly about which components are most important. Second, if you don't get to choose the hardware (for example your customers choose their own hardware), it's important that your software adapts to the available hardware. VolumeViz LDM performs well even on relatively low end hardware, but also scales up to perform better on high end hardware through the use of multiple threads and other techniques.

In many cases overall performance is limited by the time it takes to load the data. In fact this problem is getting worse because computation performance, particularly with GPUs, is increasing much faster than I/O performance. The slowest component of I/O performance is normally the network. Accessing data from a local disk drive is much faster. So you should store (or at least cache) the data on your local machine if possible. The second slowest component is disk I/O. So you should consider buying the fastest available disk drive or, even better, a RAID disk array. Accessing data from memory is normally much faster again than disk I/O. So you should consider using a 64-bit machine with a 64-bit operating system and installing a large amount of system memory. The data still has to be moved from disk to memory, but having a large memory means you may be able to keep the input data in memory for a subsequent computation and/or keep the output data in memory for faster visualization. Of course your data set may not fit in even a large system memory, so it's good to know that VolumeViz LDM data management efficiently supports "out of core" (that is, incremental) processing of large data sets for both computation and visualization.

The main factors in data loading performance are called *latency* and *bandwidth*. Latency is the time spent waiting for requested data to start transferring, for example waiting for the correct position on the disk to rotate under the read head. Bandwidth is the rate, for example megabytes per second, at which data can be transferred once the transfer starts. Both latency and bandwidth are normally part of the specification for

each component of the system that handles data (network, disk, memory, PCIe bus, etc). If you load data using a large number of small requests, latency tends to be the limiting factor. However you can "hide" the latency to some extent by having multiple read requests active at the same time. For larger data blocks bandwidth tends to be the limiting factor. However you can also hide this time to some extent by doing data loading and computation simultaneously. Next we will discuss how to accomplish this "hiding".

Multiple loading threads improve performance by (among other things) hiding latency. Each loading thread normally spends some time waiting. This could be waiting for the disk drive to seek to the correct position, waiting for data to transfer from the disk drive or waiting for data to arrive from the network. If data loading was done by a single thread, then no other work could be done during this wait time. Instead some threads will normally continue working while other threads are waiting. VolumeViz LDM automatically uses multiple loading threads whenever data is requested (either internally for visualization or externally for computation). By default VolumeViz LDM creates four data loading threads, but the application can specify the number of data loading threads. Four loading threads normally provide good performance even on a single core machine. Additional loading threads may provide some performance increase when using a RAID disk array on a multicore machine.

Asynchronous data loading improves performance by (among other things) hiding bandwidth delays. VolumeViz LDM supports both synchronous and asynchronous data loading. Using synchronous loading, the physical I/O is still multi-threaded as described in the previous paragraph, but the function waits for all the loading threads to complete their tasks, so that when the function returns all the requested data has been loaded. Using this mode reduces the complexity of computational algorithms, but does not take full advantage of multi-core CPUs. Using asynchronous loading the request function returns immediately and the application can continue with other work while the data loading continues in parallel. The application is notified via a callback function when the data is ready. Normally the application will synchronously request the first block of data, then make an asynchronous request for the second block of data and begin computing on the first block of data while the second block is loading. In this case if the computation time and the loading time for a block of data are approximately equal, then the loading time for most of the blocks is hidden.

The VolumeViz LDM data access API allows an application to get the data associated with a specific tile at a specific resolution level. It also allows an application to ignore tiling and request the data values in a trace, a slice or an arbitrary subvolume. VolumeViz LDM takes care of copying data from all the required tiles to form the requested block of data. This allows the application to adjust the requested block size to optimize for the computation platform. For example DMA transfers on the Cell BE processor are fastest when the size of the data block is a multiple of 128 bytes.

To fully optimize data loading the application should pay some attention to the tiling structure and also request data in the largest blocks possible. The tiling structure is important because we want to minimize the number of tiles that need to be loaded to satisfy the request. This is easily done by querying the tile size and using this to manage the size of the requests. Requesting large blocks is beneficial because it increases the opportunities for multiple data loader threads to work in parallel. The maximum block size may be limited by available system memory or by the computation platform.

Section 2.2 – Implementation Setup

Before we can use any VolumeViz objects we must initialize both Open Inventor and VolumeViz. If we need a 3D drawing window this will be the usual initialization, for example:

```
Widget myWindow = SoXt::init(argv[0]);
SoVolumeRendering::init();
```


However if our computation program does not need to do any actual visualization we can initialize Open Inventor without the window system classes like this:

```
SoDB::init();
SoVolumeRendering::init();
```

The data will be managed by an *SoVolumeData* node. If the data is stored in one of the formats for which there is a predefined VolumeViz reader class, for example the .ldm format, just set the *filename* field to contain the file's pathname:

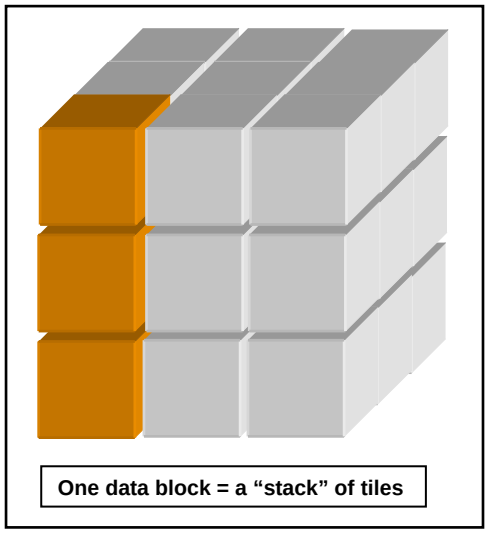
```
// Create volume data node and assign a data file
SoVolumeData *pVolData = new SoVolumeData();
pVolData->filename = "/some/path/name";
```

If the data is accessed through a custom volume reader then we need to create an instance of the reader class, initialize any parameters specific to the reader, then set the reader in the volume data node:

```
// Create, initialize and use a custom volume reader
MyCustomReader reader;
int rc = reader.setFilename( filename );
pVolData->setReader( reader );
```

We'll consider data block size first because that issue is common to both synchronous and asynchronous access modes. By data block we mean the chunk of data that our computation will process as one unit. The VolumeViz data access API allows us to request the data values in one specific tile, a subvolume, a plane, a line or a polyline. These are useful in different situations. For computation we will generally want to load a subvolume. One reason is that loading a single tile does not take advantage of the built-in multi-threaded loading in VolumeViz. So requesting multiple tiles is more efficient. Another reason is that the shape and/or size of a data block will often be constrained by specific requirements of the algorithm we are applying to the data. Let's look at a specific example.

Suppose we are dealing with seismic data which consists of a set of "traces", where each trace is a sampled signal on a time axis (time corresponds to depth, although in a non-linear way). Many algorithms require the complete signal so one constraint is that our data blocks should span the time axis of the volume. We also want to consider the LDM tile size because LDM always loads tiles of data (a tile is the granularity of data management in LDM). A reasonable strategy would be to load a block of data that is a "stack" of tiles spanning the time axis of the volume, as shown in the following figure. This will be a collection of tileSize*tileSize traces.



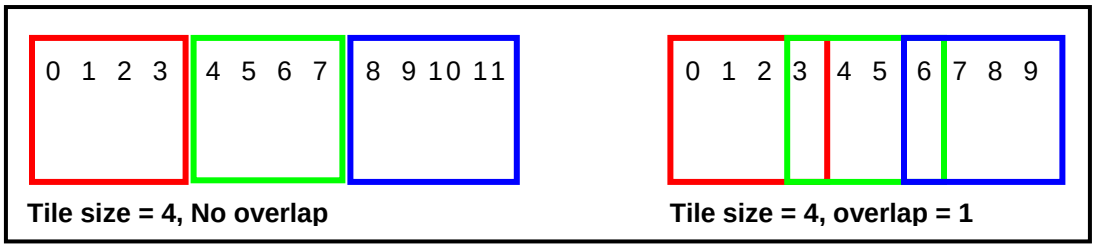
We know that VolumeViz always loads seismic data so that the time (or sample) axis is the first dimension of the volume. So if we have, for example, 1000 samples in each trace and the tile size is 64, then each block of data will be 1000 by 64 by 64, containing 4096 (64*64) traces. This is a convenient number for GPU computing because the largest texture dimension on most graphics boards is 4096. However it is still only about four million values (eight million bytes), so we might consider loading multiple stacks in one block and subdividing in the computation method.

Start by getting the dimensions of the volume and the dimensions of a tile as shown in the following code fragment. The volume dimensions can obtained using a convenience method. The tile dimensions (and other information) can be obtained using the LDM Resource Parameters object associated with the volume data object. Currently tiles are constrained to be symmetrical (i.e. a cube of voxels), so we could use any member of "tileDim" as the tile size and we will (loosely) talk about tile size as a single value.

```
// Get the volume dimensions:
SbVec3i32 volDim = pVolData->data.getSize();

// Get the tile size:
SbVec3i32 tileDim = pVolData->
         ldmResourceParameters.getValue()->tileDimension.getValue();
```

We also have to consider tile overlap (also called border). Normally adjacent tiles will overlap by at least one voxel to avoid visual artifacts (discontinuities) when interpolating data values.



This is a property of the data set that was assigned when it was converted to LDM format and cannot be changed. We can find out if the tiles overlap, and how much, using (again) the LDM Resource Parameters object associated with the volume data node, for example:

```
// Determine if volume has overlapping tiles
int overlap = pVolData->
    ldmResourceParameters.getValue()->overlapping.getValue();
```

There are two general ways to handle tile overlap. If it is important to avoid any redundant computation we can request the data in non-overlapping blocks. VolumeViz will automatically load the necessary tiles and copy the appropriate portions of those tiles into the application memory buffer. This adds some overhead to the loading process but is transparent to the application and provides a powerful tool for simplifying algorithm implementations. Alternatively we can request the data precisely on tile boundaries and accept that if there is tile overlap then a small number of traces will be computed twice. This minimizes copying and re-copying data from one location to another. For large data sets, the overall performance is often limited by the data transfers, not the actual computation. So it can make sense to minimize copying. In fact it's likely that the performance of alternative compute engines like the GPU and Cell BE will continue to increase faster than our ability to move data to and from the processor. In this guide we will load data on tile boundaries.

There is another issue at the volume boundaries. If the volume dimensions are not evenly divisible by the tile size (which is usually the case) then the tiles on at least one side of the volume will be partial tiles and therefore we will have partial data blocks. For CPU computation this is not a big problem. The

VolumeViz data access method will tell us the actual amount of data returned. We just have to be careful to use the actual dimensions when doing the computation. However for compute engines that are separate from the CPU, for example a GPU board or a Cell BE accelerator board, the time to transfer data blocks to and from the engine may be a significant part of the overall performance equation. In general these transfers are more efficient for larger blocks of data and in some cases there are specific block sizes that give optimal performance. Even if we choose the block size to maximize data transfer performance, we may get reduced performance for these partial data blocks. One solution could be to collect multiple partial blocks and combine them to make (temporary) complete blocks.

When allocating memory for the data blocks, remember that some compute engines require a specific alignment of the memory and/or specific allocation method to optimize data transfers to and from the engine. For example CUDA data transfers are optimal for "pinned" memory. But even if you are computing on the CPU, data alignment is required to use SSE instructions and affects cache performance even for standard instructions. The compilers default alignment is not necessarily optimal. Depending on your environment there may be a pragma to force alignment or a memory allocation function that allows specifying alignment (for example on Windows you can use the aligned_malloc system function). Data blocks allocated using Open Inventor's SoBufferObject classes are optimal for that device.

The basis of LDM data management for rendering is that the volume has been divided into tiles and also subsampled to create a hierarchy of volumes at multiple resolution levels. For computation we can take advantage of the data being subdivided into tiles. This allows VolumeViz to efficiently load just the data we need at any given time. However we will normally only do computation on the full resolution level of the hierarchy, which contains the original data values. This is resolution level zero, so we will normally always pass 0 for the resolution parameter of the data access methods.

We can compute the number of full resolution tiles on each axis of the volume as shown in the following code. Note that we take into account the amount of overlap (number of shared voxels) between tiles.

```
// Compute number of full resolution tiles
int xtiles   = ceil( (float)volDim[0] / (tileDim[0] - overlap));
int ytiles   = ceil( (float)volDim[1] / (tileDim[1] - overlap));
int ztiles   = ceil( (float)volDim[2] / (tileDim[2] - overlap));
int numTiles = xtiles * ytiles * ztiles;
```

Now we can easily compute the number of data values in one tile, the number of data values in one data block (where one data block is a stack of tiles spanning the X axis), and the number of data blocks we will need to process. We can also compute the number of bytes of memory to allocate for one block of data, using the number of values in a block and the number of bytes in each voxel.

```
unsigned int valuesPerTile  = tileDim[0] * tileDim[1] * tileDim[2];
unsigned int valuesPerBlock = xtiles * valuesPerTile;
unsigned int numBlocks      = ytiles * ztiles;

size_t numBytes = valuesPerBlock * pVolData->getDataSize();
```

Multi-threading: As discussed previously, VolumeViz automatically creates multiple threads to do the actual data loading. These threads are managed by the data access object. The application should not do multi-threading "on top of" the data access object. In other words a single application thread should make the calls to load data. We will see later how to use the asynchronous interface to decouple the application thread completely from data loading. Of course the computation done on each data block should be multi-threaded if possible and appropriate.

System memory: In some cases the VolumeViz system memory limit will not be important. For example, if we will only use each data block exactly once, then theoretically we only need enough system memory to load one data block. However in some cases when tile overlap is non-zero, VolumeViz will load adjacent tiles of data that are not actually required for the specified data block. If possible, make sure the system

memory limit is at least four times the size of one data block. (This should be corrected in the next release.) In other cases where we need to make multiple passes over the data we might be able to take advantage of VolumeViz automatic caching. The VolumeViz data manager will try to keep tiles in system memory as long as there is still memory available for new requests. For example, if we are processing a 4 gigabyte volume on a system with 8 gigabytes of memory, we could set the system memory limit to a little over 4 gigabytes and keep all the tiles in memory. The system memory limit can be set globally (the total for all volumes) or can be set for a specific volume using its *ldmResourceParameters* field. For example:

```
// Set system memory limit globally to 4 GB
SoLDMGlobalResourceParameters::setMaxMainMemory( 4000 );

// Set system memory limit for this volume to 4 GB
pVolData->ldmResourceParameters.getValue()->
  setMaxMainMemory( 4000 );
```

Performance: You may notice that loading the first block of data is slower than loading subsequent blocks. Some of this time is due to setup and initialization. However in some cases when the tile overlap is non-zero, VolumeViz will load adjacent tiles of data that are not actually required for the specified data block. This effect is most noticeable on the first data block requested. In most cases the extra tiles will be needed later for other blocks of data and so there will be zero load time for those tiles later because they will already be in memory when those requests are made. So the total load time for all blocks will be as expected. (This should be corrected in the next release.)

Section 2.3 – Synchronous Loading

First we need to get the data access object associated with the volume data node and declare an object to receive the result of data access requests:

```
// Get the data access object and result object
SoVolumeData::LDMDataAccess &dataAccess =
                              pVolData->getLdmDataAccess();
SoVolumeData::LDMDataAccess::DataInfoBox dataInfo;
```

We also need to allocate memory for the input block of data (to be loaded) and the output block of data (to be computed). In some cases we could re-use the same memory if our algorithm allows computing a result “in place”. However in the general case we will need a separate output block. Also, in the next section we will discuss why multiple input blocks may be useful in order to do parallel processing. We will use the *SoCpuBuffer* class to manage our memory. We can “map” this buffer into a device buffer to do computing on, for example, a CUDA device. It’s also convenient that the *setSize()* method automatically allocates “aligned” memory, which improves performance of SSE instructions, data transfer to a CUDA device, etc. After allocating memory we’ll check that the allocation succeeded and “map” the buffer for both reading and writing.

```
// Get the data access object and result object
// Allocate memory for input data block
SoCpuBufferObject *pSrcBuffer = new SoCpuBufferObject;
pSrcBuffer->setSize( numBytes );
if (! pSrcBuffer->map( SoBufferObject::READ_WRITE ) ) {
  std::cerr << "Failed to allocate " << numBytes << " bytes.\n";
  return -1;
}
```

The algorithm is simple. In pseudo-code it looks like this:

```
while (moreBlocks) {
  requestData( block++, pLoadBuf );   // Request next block
  compute( pLoadBuf, pResultBuf );    // Compute loaded block
}
```

First we will do some setup for computing data block boundaries exactly on tile boundaries. If there is no tile overlap this is trivial. The way we have defined a data block in this example, the X boundaries of a data block will always be the same: 0 and volDim[0]-1. The start voxel of each block in the Y and Z dimensions will depend on the amount of overlap. But once we have this value the end voxel is easy because we know that tiles always contain exactly tileDim voxels. We will define the limiting voxel position for each dimension. All this setup could be done inside the loop and would not have much effect on performance. We're doing it here to hopefully make the example clearer.

```
// Setup for calculating data blocks on tile boundaries
int yinc   = tileDim[1] - overlap;    // Increment in Y
int zinc   = tileDim[2] - overlap;    // Increment in Z
int xLimit = volDim[0] - 1;
int yLimit = volDim[1] - 1;
int zLimit = volDim[2] - 1;
```

Now we can look at the loop over all the data blocks. Some details are omitted for clarity in this document. In this example each data block is a stack of tiles spanning the volume X axis, so we know we can visit each block by looping over the number of tiles on the volume Y and Z axes.

```
// Loop over all stacks (data blocks)
SbBox3i32 subvol;
for (int iz = 0; iz < ztiles; ++iz) {
  for (int iy = 0; iy < ytiles; ++iy) {
```

Step 1 in each pass of the loop is to compute the bounds of the subvolume that defines this data block. Note that the X bounds are always the same (the volume dimension).

```
    // Compute bounds of this data block (stack of tiles)
    int xmin = 0;
    int xmax = xLimit;
    int ymin = iy * yinc;
    int ymax = ymin + tileDim[1] - 1;
    if (ymax > yLimit)
      ymax = yLimit;
    int zmin = iz * zinc;
    int zmax = zmin + tileDim[2] - 1;
    if (zmax > zLimit)
      zmax = zLimit;
    subvol.setBounds( xmin, ymin, zmin, xmax, ymax, zmax );
```

Step 2 is to request the data. Because this is a synchronous request we will automatically wait for the request to be completed (although remember that VolumeViz is loading the tiles that make up this data block using separate threads in order to overlap physical I/O delays like disk latency). The first parameter of the getData method is the tile resolution level requested. For computation this will normally be zero to request the full resolution data.

```
    // Load data
    dataInfo = dataAccess.getData( 0, subvol, pSrcBuffer );
```


Step 3 is to get the actual dimensions of the data returned and do the computation on the just loaded data block. It's important to get the actual dimensions because, as we discussed earlier, the tiles on the edges of the volume will typically be partial tiles. For the purposes of this discussion it's not important exactly what computation is done or how. However note that in the example program the doComputation method does a sleep (using the platform independent SbTime::sleep method) for a period of time depending on the number of data values. This is to simulate the time required for the computation and allow us to compare the performance of the asynchronous example and the synchronous example. Exactly what parameters need to be passed to the computation method will obviously depend on the particular algorithm.

```
    // Get the actual dimensions of the data loaded.
    actDim = dataInfo.bufferDimension;

    // Do computation
    doComputation( pSrcBuffer, pDstBuffer, actDim, dataType );
```

The complete example project *loadData* is provided in the Open Inventor SDK along with this document.

Section 2.4 – Asynchronous Loading

VolumeViz LDM already uses multiple threads to load multiple tiles in parallel. This is an automatic parallelization provided by the library and helps overlap wait times caused by disk or network latencies. However it does not allow the application to completely overlap data loading with computation, which is an increasingly critical goal on multi-core processors and helps to hide bandwidth delays.

Using the synchronous methods we only needed two data buffers, one to hold the input data and one to hold the output data (the result of the computation). Using the asynchronous methods we will need (at least) three data buffers. Initially we will always use one of the buffers over and over for the result of the computation. We will use the other two buffers for "double buffering" data. At any given time we will be using buffer1, the most recently loaded data block, as the input for the computation, while VolumeViz is loading the next requested data block into buffer2. When both computation and loading are finished, we will "swap" the buffers. That is, we will start using buffer2 as the input for the computation and start loading the next data block into buffer1.

This scheme can be extended to more than two buffers if there is enough system memory. The asynchronous data access methods allow more than one data loading request to be active, so we can actually schedule multiple data blocks to be loaded. This adds some complexity to the application's "book-keeping". For example the application must keep track of which requests have completed, particularly if the algorithm depends on processing the data blocks in a particular order. Whether the complexity of multiple buffers is justified depends on many factors. This is another reason to carefully measure the time required for loading and for computation. If the computation time for one data block is much longer than the load time then it may make sense to overlap loading of multiple data blocks. However this also indicates that more time should be invested in optimizing the computation, for example by using multi-threading or a compute accelerator.

Performance expectations should be reasonable. If the loading time and compute time for a data block were exactly the same and we could completely overlap these operations, then we might imagine that asynchronous loading would double the overall performance. This is unfortunately rare in practice. Generally it is difficult to balance the times and even more difficult to completely overlap them. For example, even if the load and compute times are the same, overlapping the times by 50% will only reduce the overall time by 25%. The amount of overlap will depend on how many CPU cores are available as well as hardware and operating system factors. Performance measurements on small data sets may be misleading. We need to load one block of data before the loop and compute one block after the loop, so these operations cannot be overlapped. The load and compute time for these two blocks will skew the

overall performance if there are only a small number of data blocks in total.

Before we start we need to "prime the pump" by loading the first data block synchronously. Then we can submit a request for the second data block and begin computing on the first block while the second block is being loaded. When we are finished computing the first block we may need to wait for the loading of the second block to finish. When the second block is loaded we can swap the buffer pointers, request the third block and begin computing the second block. In pseudo-code it looks like this:

```
pDataBuf = Buffer0;
pLoadBuf = Buffer1;
getData( block, pDataBuf );         // Load 1st block
while (moreBlocks) {
  requestData( block++, pLoadBuf ); // Request next block
  compute( pDataBuf, pResultBuf );  // Compute loaded block
  waitForData();                    // Wait for asynch block
  temp = pDataBuf;                  // Swap buffer pointers
  pDataBuf = pLoadBuf;
  pLoadBuf = temp;
}
compute( pDataBuf, pResultBuf );    // Compute the last block
```

The *SoLDMDataAccess* class we used for synchronous data access also contains the methods for asynchronous access. The *requestData* method takes a resolution level (normally 0 for full resolution), the subvolume to be loaded and a buffer to be filled with the data. This method schedules the data request and immediately returns a *requestId*. If the requestId is negative this indicates that all the necessary tiles are already in memory. In other words the data can be retrieved immediately. This is not likely to happen if we are only making a single pass through the volume, so the example program just checks this when it's time to wait for loading to finish. Otherwise the requestId can be used to check the status of the request and to complete the transaction when the data is ready. When the data (and the application) are ready we'll call the *getRequestedData* method to complete the transaction. This method takes a requestId (remember that multiple load requests can be "in flight" at the same time) and returns a *DataInfoBox* object. Just as in the synchronous case this object contains the completion status and the actual dimensions of the data loaded (remember that some tiles are only partial). How do we know when the data is ready?

When the requested data is ready VolumeViz will signal this by calling the data access object's *endRequest* method. This method is called with a requestId indicating which request has completed. Note that this method is called from a VolumeViz data loader thread, not the main application thread, so we need to be careful about accessing application data. If necessary we could use the *SbThreadMutex* class to lock access to data that needs to be "thread safe", but generally it will be sufficient to set a flag that the application can check. We need to provide our own implementation of the endRequest method, so we need to create a new class derived from SoLDMDataAccess.

We also need a way to wait for the requested data to be ready, because we don't know if the loading or the computation will finish first. Currently SoLDMDataAccess does not provide a *waitForData* method. In fact it is important not to call getRequestedData before the data is actually ready because this call deletes a pending request. We will implement our own waitForData method. To do this we will override the requestData method and set a flag (m_dataReady) that we can check to see if the endRequest method has been called yet. When we need to wait for a data request to finish we will simply loop checking the completion flag and sleeping for a short time if the flag has not been set yet. For this example we know that only one data request will ever be pending at a time. Here is the custom class:

```
class MyAsynchLDMDataAccess : public SoLDMDataAccess
{
  // Override standard requestData method.
  // Set data-not-ready before submitting request to parent class
```



```
  int requestData( int resolution,
                   const SbBox3i32& subVol, void* buffer)
  {
    m_dataReady = false;
    return SoLDMDataAccess::requestData(resolution,subVol,buffer);
  }

  // Do not return until the data is ready
  // (wait for endRequest to set our data-ready flag)
  void waitForData( int requestId )
  {
    while (! m_dataReady)
      SbTime::sleep( 10 );  // msec
  }

protected:
  bool m_dataReady;

  // This method will be called from a VolumeViz data loader thread.
  // Set data-ready flag so the waitForData method can return.
  virtual void endRequest(int requestId)
  { m_dataReady = true; }
};
```

Instead of requesting the data access object from the volume data object (as we did for synchronous access), we will create an instance of our custom data access object. Then we will use the data access object's *setDataSet* method to associate it with the volume data object.

```
// Create data access object and associate with volume data
MyAsynchLDMDataAccess asynchDataAccess;
asynchDataAccess.setDataSet( pVolData );
```

Before we start the loop we need to synchronously load the first data block as discussed previously (see "priming the pump"). This will be the initial "current" data block. Currently it is not possible to make synchronous loading calls (getData) using a separate data access object. We can only use this object for asynchronous loading. In order to synchronously load the first block of data (before the loop) we will use the volume's built-in data access object, just as we did in the synchronous loading example. Here we load the first data block into the data buffer (the one that our computation will use):

```
// Compute bounds of first data block
subvol.setBounds( xmin,0,0, xmax,tileDim[1]-1,tileDim[2]-1 );

// Load first block synchronously (we have to wait anyway)
dataInfo = pVolData->getLdmDataAccess()
           .getData( 0, subvol, pDataBuffer );
```

Now we can start the loop over all the data blocks. Some details are omitted for clarity and because the computation of the block boundaries depends on the application and how it needs to structure the data blocks. Step 1 in every pass of the loop is to request the next block of data so that VolumeViz can be loading this data while we are doing the computation on the current block of data.

```
// Compute bounds of NEXT data block (code omitted)
subvol.setBounds(xmin, ymin, zmin, xmax, ymax, zmax );

// Request next block of data (this call returns immediately)
```



```
int requestId =
    asynchDataAccess.requestData( 0, subvol, pLoadBuffer );
```

Step 2 is to do the computation on the current data block while VolumeViz is asynchronously loading the next block. For the purposes of this discussion it's not important exactly what computation is done or how. However note that in the example program the doComputation method does a sleep for a period of time depending on the number of data values. This is to simulate the time required for the computation and allow us to compare the performance of the asynchronous example and the synchronous example.

Step 3 is to wait (if necessary) for the next block data to finish loading. Note that we have to explicitly handle the case where requestId is negative (discussed previously) because the endRequest method will not be called in this case and getRequestedData must not be called with a negative value.

```
// Wait for requested block to finish loading
if (requestId < 0) {  // Data block was already in memory
  requestId = -requestId;
}
else {                // Data not ready yet, wait for it
  asynchDataAccess.waitForData( requestId );
}
// Complete the data request
asynchDataAccess.getRequestedData( requestId, dataInfo );
```

Finally, when the loop is finished we have loaded every block of data, but we still need to do the computation on the last block of data (the one we just waited for). The complete example project *loadDataAsynch* is included in the Open Inventor SDK along with this document.

Section 3 – Storing LDM Data

Section 3.1 – Store LDM data on disk

Now that we know how to load LDM data and do some sort of computation, we need a way to save the new volume on disk so it can used for later computation and/or visualization. The *SoLDMWriter* class provides this service. SoLDMWriter creates an LDM file (in VSG's .ldm format) and allows the application to store data blocks in any order. The data blocks may be specific tiles or arbitrary regions of the volume. The most common usage is to store blocks of full resolution data. SoLDMWriter incorporates an instance of the LDM converter (see *SoConverter*) which can automatically generate the lower resolution (subsampled) tiles from the full resolution tiles. This can be done incrementally when the *writeSubVolume()* method is called or after all the full resolution data has been stored, specifically when the *finish()* method is called. The *writeTile()* method also allows the application to directly store lower resolution tiles in case a proprietary subsampling algorithm is being used. The result will normally be a .ldm file (LDM header) and a .dat file (data).

It is not necessary, in all cases, to create the lower resolution tiles or even to create the complete set of full resolution tiles. SoLDMWriter supports the converter's partial conversion feature. If some tiles are missing when the *finish()* method is called, then in addition to the header and data files there will also be a .fcp file with the same name as the header and data files. The LDM header file will contain a reference to this file in the CompletionFilename tag. The .fcp file stores information about which tiles actually exist. Tiles that do not exist are considered to be filled with a constant default value (see *getHoleData()*). This feature allows us, for example, to compute and visualize a subset of the volume before committing to computation on the whole volume. However note that the converter currently pre-allocates disk space for the ".dat" file assuming that all tiles will exist. So skipping creation of the subsampled tiles or writing only a subset of the tiles can reduce computation time and disk I/O time, but it does not reduce the disk space requirement for the volume.

The output volume is not required to have the same characteristics as the input volume. This allows us, for example, to extract a subset of a volume as a new (smaller) volume. All the options of the LDM converter are available. So the LDM writer can also create a volume with a different data type or a different tile overlap value. It can also create a compressed data file to reduce disk space requirements.

The first step is to create an SoLDMWriter object and initialize it with the desired characteristics for the output volume. The *initialize()* method takes the name of the output file to be created. Note that the return value from initialize() is actually the *SoConverter* enum ConverterError. Success is indicated by the value CVT_NO_ERROR (zero). One version of the initialize() method takes a pointer to the input volume and initializes the writer with the dimensions, size and data type of the input volume.

```
// Create an LDM Writer and initialize from the input volume
SoLDMWriter ldmWriter;
ldmWriter.initialize( outfilename, *pVolData );
```

This is convenient but limited. Generally we will need to specify other parameters or options to the converter. For example the default for the converter is "verbose" mode, meaning that by default the converter will printf a lot of information. Generally we will want to disable this output in a production environment by setting the converter's "-q" option. The other form of the initialize() method takes explicit parameters for the volume size, dimension and data type, plus "argc" and "argv" parameters that can be used to pass command line arguments to the converter. In addition to the size, dimension and data type we will normally need to use command line options to specify the tile size and overlap. We may also want to specify a larger value for working memory if we want the converter to generate the subsampled resolution levels. This interface also allows you to invoke the data compression feature of the converter ("-c" option). Using the more powerful initialize interface can be done like the following code. Note that each token of

the command line (for example the option and the option value) must be a separate string.

```
SoLDMWriter ldmWriter;
SbString s_tileSize, s_overlap;
int   writer_argc = 0;
char *writer_argv[10];                  // Max 10 options
writer_argv[writer_argc++] = "-q";      // Quiet mode (no printf's)
if (tileSize != 64) {
  s_tileSize.sprintf( "%d", tileSize ); // Format tileSize as string
  writer_argv[writer_argc++] = "-t";    // Tile size parameter
  writer_argv[writer_argc++] =
    const_cast<char*>(s_tileSize.getString());
}
if (overlap > 0) {
  s_overlap.sprintf( "%d", overlap );   // Format overlap as string
  writer_argv[writer_argc++] = "-b";    // Overlap parameter
  writer_argv[writer_argc++] =
    const_cast<char*>(s_overlap.getString());
}
ldmWriter.initialize( outfilename, volSize, volDim, dataType,
                      writer_argc, writer_argv );
```

After computing each data block of the output volume we can write that block into the output file using the *writeSubvolume()* method. The first parameter is an *SbBox3i32* specifying the subvolume in voxel coordinates. In this example it is the same subvolume specification we used to load the data block and defines a stack of tiles. The second parameter is a pointer to the data, which should be full resolution data values. *writeSubVolume()* will automatically determine which tiles are contained in or intersect (are partially contained in) this block of data based on the *subVolume* parameter. All tiles that are completely inside the subvolume are written into the file (overwriting any already existing data for those tiles). All tiles that are partially inside the subvolume are also written into the file. However if a partially inside tile already exists in the file (perhaps from a previous call to writeSubVolume), that tile will first be read back from the file, updated with the new voxel values, then re-written into the file. The special case of tiles that intersect the subvolume only because of the built-in tile overlap (i.e. overlap > 0) is handled by the *doOverlappingTiles* parameter, discussed below.

In VolumeViz 7.0 the *writeSubVolume()* method did not generate the corresponding lower resolution tiles (they could only be generated by the *finish()* method). This behavior has changed in VolumeViz 7.1 so that, by default, the *writeSubvolume()* method will (if possible) immediately generate the lower resolution (subsampled) tiles. This can be significantly faster than generating them in the *finish()* method because the full resolution data is already in memory (does not need to be read back from disk). However to really take advantage of this feature the specified subvolume should contain blocks of eight adjacent tiles (each lower resolution tile is generated from a block of eight tiles at the next higher resolution level). In this example our subvolume is a single stack (or column) of tiles so we cannot take full advantage. The program could be enhanced to process a subvolume consisting of four adjacent stacks in order to take advantage of this feature. To suppress immediate generation of lower resolution tiles, set the *doMultiResolution* parameter to FALSE. In this example we will only write the full resolution tiles.

By default, *writeSubVolume()* assumes that all tiles that intersect *subvolume* to any extent must be written (and if necessary, read back, updated, then written). This allows you to write arbitrary subvolumes. For example, you can compute using non-overlapping subvolumes regardless of what the tile overlap setting is for that volume. However if tiles overlap (overlap > 0) it may require many more read/write operations (and therefore be much slower) than expected. When tiles overlap, even if the subvolume is exactly the extent of a single tile, that subvolume actually overlaps up to 8 neighbor tiles. As discussed previously, in this example we are using subvolumes that correspond exactly to tile boundaries. This is because disk and network I/O is almost always more expensive than computation, so it is generally better to redundantly compute overlapping voxels and thereby avoid redundantly writing tiles. In order to avoid writing tiles that

intersect our subvolume because of tile overlap we will set the *doOverlappingTiles* parameter to FALSE. In this case tiles that intersect the subvolume by tile overlap (or less) are not considered intersecting.

```
ldmWriter.writeSubVolume( subvol, pSrcBuffer, FALSE, FALSE );
```

After all the data blocks have been written into the output file, call the *finish()* method to complete creation of the LDM file. Note that the default behavior is to invoke the converter and generate the subsampled resolution levels. This may take a significant amount of time for a large volume. In this example we will skip this step by passing false for the doMultiResolution parameter. (Note that in version 7.0 this method took a parameter of type *bool* instead of *SbBool*.)

```
ldmWriter.finish( FALSE );
```

In most cases writing an LDM file will be slower than reading an LDM file (you will see this in the timing output from the example program). One reason is that, unlike the LDM data loader, the LDM writer is not internally multi-threaded (in this version). Another reason is that the LDM writer does not provide an asynchronous interface (in this version) that would allow us to overlap writing with loading and computing. It should be possible to run the LDM writer in a separate thread at the application level, but we will not attempt that in this example. In some cases the new volume is temporary or at least its value cannot be determined until it has been visualized and examined on screen. In these cases it might make sense to create a new LDM volume (or partial volume) in memory and visualize it before committing to disk. We will discuss this in section 3.3.

Section 3.2 – Compute Size of LDM Data

It can be useful to compute how much storage will be needed for a volume in LDM format. We might use this for example to determine if enough disk space is available before trying to create a file or to determine if enough free space is available to create the volume in memory. At this point it should be clear how to compute the size of the full resolution data in LDM format. In section 2.2 we showed how to compute the number of full resolution tiles given the volume dimensions, the tile dimensions and the tile overlap. The size of the full resolution data is simply the number of tiles times the number of bytes in one tile. This size will normally be slightly larger than the raw size of the data (i.e. number of voxels times bytes per voxel) because there will be some tiles on the edges of the volume that are only partially filled. However for large volumes this is only a small percentage of the total volume size.

```
// Compute number of full resolution tiles (level 0)
int xtiles = (int)ceil( (float)volDim[0] / (tileDim[0] - overlap));
int ytiles = (int)ceil( (float)volDim[1] / (tileDim[1] - overlap));
int ztiles = (int)ceil( (float)volDim[2] / (tileDim[2] - overlap));
size_t numTiles = xtiles * ytiles * ztiles;
```

What about the subsampled levels of the resolution hierarchy? We know that at each level a block of eight tiles will be collapsed into a single tile at the next higher (more subsampled) level. However we can't just divide the number of tiles by eight because we have to take into account the edges of the volume where the "leftover" tiles (after collapsing all the blocks of eight) have to be collapsed. The simplest way is to divide the number of tiles on each axis by two, round up to the nearest integer, then multiply the new tiles per axis together to get the total number of tiles at the new level. Do this for each resolution level. When we reach a level with only one tile that must be the root tile (the top of the resolution hierarchy).

```
// Loop over resolution levels
size_t totalTiles = numTiles;
while (numTiles > 1) {
  xtiles = (int)ceil( (float)xtiles / 2);
  ytiles = (int)ceil( (float)ytiles / 2);
```



```
  ztiles = (int)ceil( (float)ztiles / 2);
  numTiles = xtiles * ytiles * ztiles;
  totalTiles += numTiles;
}
```

Finally compute the number of bytes in one tile using the tile dimensions to get the number of voxels and multiplying by the number of bytes in each voxel (e.g. four bytes for float values). Then compute the total number of bytes required to store the complete volume in LDM format.

```
// Compute total bytes of LDM data
size_t  bytesPerTile = tileDim[0]*tileDim[1]*tileDim[2] * bytesPV;
int64_t totalBytes   = bytesPerTile * totalTiles;
```

The complete source code is included in the Open Inventor SDK along with this document.

Section 3.3 – Store LDM data in memory

Suppose we want to visualize the new volume we computed before deciding to commit the data to disk (which will take a lot of time). Suppose we want to compute only a subset of the volume and visualize that subset before committing to computation of the whole volume. Both of these scenarios could be addressed by creating a new LDM volume in memory. We can visualize the new data using this memory volume and then, if appropriate, write the data out to disk. Clearly the volumes we can handle using this technique are quite limited compared to large seismic surveys, but we can save so much time, especially in an iterative situation, that it may be worth considering.

VolumeViz does not currently provide a built-in class for creating a volume in memory. But we know that VolumeViz always accesses the volume data through a "volume reader" object, an instance of a class derived from *SoVolumeReader*. So effectively VolumeViz can access data on any type of storage (or in any format) that can be adapted through a custom volume reader class. In this section we will create a volume reader class that provides an interface to store data in memory. During the computation loop we will use it just like we used SoLDMWriter in the previous section (except without the long time delay for actually writing to disk). After the computation loop we can create a new volume data object using our custom reader and use it to visualize the new volume (or subvolume). Just as with SoLDMWriter we do not need to create every tile in the volume. The custom reader will keep a record of which tiles exist and return an "empty" tile (filled with a constant value) when necessary.

There are many possible ways to manage and store the tile data. For this example we will use the simplest and fastest implementation. We know that VolumeViz will call the *readTile()* method with the "fileID" of the requested tile. In VolumeViz a "tileID" is the index of a tile in a conceptual complete, symmetrical hierarchy. A fileID is the index of a tile in a sequential numbering of the tiles that actually contain data. In a symmetric cubical volume tileID and fileID would be identical, but in most real world volumes there are many fewer fileIDs than tileIDs. Once we know the total number of fileIDs in the volume we will allocate an array of (that many) pointers and initialize them to NULL. When a tile is stored into the volume we will allocate memory, copy the data and save the address of that memory in the tile pointer array. When a tile is requested, if the corresponding pointer is valid we will copy that data, else we will copy the data from a predefined "empty" (or undefined) tile.

Section 3.3.1 – Declaration

We'll start by deriving our new class from SoVolumeReader. We will need the header file for SoVolumeData so we can use that type in the declaration of the initialize method. And we will need to declare the class name SoLDMTopoOctree so we can declare a member variable. This powerful utility

class allows us to query the tile structure of the volume, for example the total number of tiles.

```
#include <LDM/readers/SoVolumeReader.h>
#include <VolumeViz/nodes/SoVolumeData.h>

class SoLDMTopoOctree;

class LdmMemoryReader : public SoVolumeReader  {
```

We will provide two ways to initialize the LDM memory volume. First, the volume can be initialized by passing a pointer to an existing SoVolumeData object. This is the most convenient method if you are creating a new volume with the same characteristics as a "source" volume. The volume can also be initialized using specific values for dimension, size, tileSize, etc. This will be convenient if we need to create a new volume “from scratch”.

```
int initialize( const SoVolumeData *pVolData );

int initialize( const SbBox3f & size, const SbVec3i32 & dimension,
                SoDataSet::DataType dataType, int tileSize = 64,
                int overlap = 1 );
```

We can imagine many different ways of passing data to the memory volume (see for example the methods of SoLDMWriter). For this example we will only implement one. Blocks of data can added to the volume by calling the *putStackOfTiles()* method with a region and a block of data. In the current (limited) implementation the region must be a "stack" of tiles along the volume X axis. In other words the Y and Z dimensions of the region must be exactly tileSize (e.g. 64). This would be suitable for seismic data where the volume X axis is the time (or sample) axis, so the region can be, for example, a tileSize by tileSize block of complete traces. This is the way we defined the data blocks in the previous sections of this document. Of course the class could be extended to support arbitrary regions, specific tiles, etc. Note that with the current interface it is only possible to store full resolution data. All tiles at subsampled levels of the hierarchy are considered to be filled with a constant value (zero by default). This should not be a problem for visualizing a small volume (or a small subvolume of a large volume).

```
/* Store a stack of tiles identified by a region. */
virtual SbBool putStackOfTiles( const SbBox3i32& region,
                                const void* data );
```

Because this class is an LDM volume reader we must implement the *readTile()* method.

```
/* Index specifies a “fileID” identifying a tile.
 * tilePosition specifies the extent of the tile.
 * Data should be copied into the provided buffer. */
virtual SbBool readTile( int index, unsigned char*&buffer,
                         const SbBox3i32& tilePosition );
```

Section 3.3.2 – Initialization

The most important thing the constructor must do is set the inherited member variable that tells VolumeViz this is an LDM tiled data set.

```
// This member variable is inherited from SoVolumeReader
// and tells VolumeViz that we are an LDM volume
m_dataConverted = TRUE;
```


Initializing from an arbitrary set of volume parameters is straightforward. We just copy the specified volume characteristics into our member variables. Initializing from an existing volume data object is only slightly more complicated. In this case we will save the address of the volume data object, increment its reference count (by calling *ref()*) to keep it alive while we are using it, query the volume characteristics (size, dimension, etc) and save their values in our member variables.

In both cases we need to find out how many tiles we will potentially need to store to represent the volume. This is the total number including both full resolution and subsampled tiles. Since we are not actually storing subsampled tiles in this version we could use the slightly smaller number of full resolution tiles. But the amount of memory saved in the tile lookup table would be small compared to the total size of the tiles themselves. First we will create an instance of the utility class *SoLDMTopoOctree* and initialize it with the dimensions, tile size and overlap values of the volume we are representing. We will also use this object later when storing tile data. The getNumFileIDs method returns the total number of tiles.

```
// Create and initialize topoOctree object
// Find out how many tiles (potentially) exist in this volume
m_pTopoOctree = new SoLDMTopoOctree();
m_pTopoOctree->init( dimension, tileSize, overlap );
m_numFileIds = m_pTopoOctree->getNumFileIDs();
```

Section 3.3.3 – Store Data

The first thing the putStackOfTiles method must do is create the array of pointers and initialize them. We already saved the total number of tiles in member variable m_numFileIds.

```
LdmMemoryReader::putStackOfTiles( const SbBox3i32& region,
                                  const void* data )
{
  // Initialize tile map if necessary
  if (! m_tileMap) {
    m_tileMap = new unsigned char*[m_numFileIds];
    memset( m_tileMap, 0, m_numFileIds * sizeof(unsigned char*) );
  }
```

Next we get the min and max coordinates of the specified region and compute how many traces, voxels and bytes we're working with. We need this because we're going to "disassemble" the stack of tiles and copy the appropriate portions of the data into the memory allocated for separate tiles. The numBytesPerTile and numBytesInStack variables will tell us how to move the pointers when copying portions of the data.

```
  // Get bounds of region
  SbVec3i32 regmin, regmax;
  region.getBounds( regmin, regmax );
  // How many traces, voxels and bytes are we working with
  int numVoxelsY      = regmax[1] - regmin[1] + 1;
  int numVoxelsZ      = regmax[2] - regmin[2] + 1;
  int numTraces       = numVoxelsY * numVoxelsZ;
  int numVoxelPerTile = m_tileSize[0];
  int numVoxelInStack = regmax[0] - regmin[0] + 1;
  int numBytesPerTile = numVoxelPerTile * m_bytesPerVoxel;
  int numBytesInStack = numVoxelInStack * m_bytesPerVoxel;
```

The voxpos variable will represent a position in voxel coordinates. Using our *SoLDMTopoOctree* object we can query which tile contains this position and know where to store the tile data. We'll start with the first voxel in the region, but if the tiles overlap we'll move farther into the tile to ensure that our test

position belongs to exactly one tile (i.e. is not one of the overlap voxels).

```
// Start with the first voxel inside the region.
SbVec3i32 voxpos = regmin;
int xmin = regmin[0];
int xmax = regmax[0];
int xinc = m_tileSize[0];
if (m_border) {
  // Use 2nd voxel inside region to avoid ambiguous overlap voxels
  if ((regmax[1] - regmin[1]) > 1)
    voxpos[1] += 1;
  if ((regmax[2] - regmin[2]) > 1)
    voxpos[2] += 1;
  xinc = m_tileSize[0] - 1;
}
```

Now that we have a starting position in voxel coordinates we're going to loop, copying the data for each tile from the input buffer to the tile storage. Notice that one of the powerful features of *SoLDMTopoOctree* is a query that tells us which tile (at a specified resolution level) contains a specified point in voxel coordinates. Each time we update the position (voxpos) we will use this query to get the tileID of the containing tile, then use *SoLDMTopoOctree* again to convert that tileID into a fileID, which we can use as an index into the tile pointer array.

```
int itile = 0;
do {
  // Get id of tile containing this voxel
  SoLDMTileID tileId = m_pTopoOctree ->getTileID( voxpos, 0 );
  int fileId = m_pTopoOctree ->getFileID( tileId );
```

If this is the first time this tile has been written into the volume, the corresponding entry in the tile pointer array will be NULL. In this case we need to allocate memory for the tile data and initialize it to zero.

```
  // Create tile storage if necessary
  if (m_tileMap[fileId] == NULL) {
    size_t numBytes = m_tileSize[0] * m_tileSize[1]
                    * m_tileSize[2] * m_bytesPerVoxel;
    m_tileMap[fileId] = new unsigned char[numBytes];
    memset( m_tileMap[fileId], 0, numBytes );
  }
```

Now we'll setup to copy the tile data. It's not possible to copy the data for one tile with a single call. The data comes in as a contiguous block with all the values in the first trace, followed by all the values in the second trace, etc. The first tileSize values of the first trace belong to the first tile, but the next tileSize values belong to the second tile, and so on. We need each tile to be a contiguous block with tileSize values taken from the first trace, followed by tileSize values taken from the second trace and so on. So the best we can do is make tileSize*tileSize calls to memcpy. The variable numBytesToCopy is how many bytes will be copied on each call. The variables pSrc and pDst will be updated for each call to memcpy.

```
  // Setup to copy tile data
  int xbeg = itile * xinc;
  int xend = xbeg + m_tileSize[0] - 1;
  if (xend > xmax)
    xend = xmax;
  int numBytesToCopy = (xend - xbeg + 1) * m_bytesPerVoxel;
  unsigned char *pSrc =
          (unsigned char*)data + (itile * xinc * m_bytesPerVoxel);
```



```
    unsigned char *pDst = m_tileMap[fileId];
```

This loop efficiently (as possible) copies the data for one tile.

```
    // Copy tile data
    for (int iTrace = 0; iTrace < numTraces; ++iTrace) {
      memcpy( pDst, pSrc, numBytesToCopy );
      pSrc += numBytesInStack;
      pDst += numBytesPerTile;
    }
```

Finally we increment the voxel position and check that we are still inside the source data block.

```
    // Increment voxel position
    itile++;
    voxpos[0] = (itile * xinc) + 1;
  } while (voxpos[0] <= xmax);
```

Section 3.3.4 – Get Data

Implementation of the readTile method is straightforward using this simple data management strategy. First we compute the number of bytes in a single tile using the tile dimensions and number of bytes in one voxel (saved in the constructor). Since we are not using compression, every tile will contain the same number of bytes. If data exists for the requested tile there will be a non-NULL entry in the array of pointers *m_tileMap* and we can simply copy the data into the provided buffer. Note that VolumeViz does not currently provide a way to simply return a pointer to the tile data. So we are forced to maintain two copies of the data (one here in the custom reader and one inside VolumeViz).

```
LdmMemoryReader::readTile(int index, unsigned char*&buffer, …
{
  // Number of bytes in a tile
  size_t tileBytes =
         m_tileSize[0]*m_tileSize[1]*m_tileSize[2]*m_bytesPerVoxel;

  // Copy data if we have data for this tile
  if (m_tileMap[index] != NULL)
    memcpy( buffer, m_tileMap[index], tileBytes );
```

If no data exists for the requested tile we should return a copy of the “empty” or undefined tile. Of course we only need to create this tile once. The utility method *getHoleData()* returns a pointer to this data and, if necessary, creates the data. The code fragments shown here illustrate the principles, but are not intended to be production quality. The actual implementation includes error checking, debug output, and so on.

```
  else
    memcpy( buffer, getHoleData(), tileBytes );
```

The complete source code for this class is shown in Appendix D (and should be available with this document).

Section 3.3.5 – Using the Memory Volume

Generally using the memory volume is similar to using SoLDMWriter. First we create a memory reader object and initialize it to have the same characteristics as the input volume. Note that starting with version 7.0 volume readers are reference counted objects just like nodes and paths. After creating the memory

reader we should increment its reference count.

```
// Create an LDM Memory Reader
// Initialize it to match the input volume
LdmMemoryReader *pMemReader = new LdmMemoryReader();
pMemReader->ref();
pMemReader->initialize( pVolData );
```

Storing a block of data in the memory volume is essentially the same as using SoLDMWriter:

```
// STORE: Store computed block in new (memory) volume
pMemReader->putStackOfTiles( subvol, pDstBuffer );
```

The memory volume does not require a “finish” call. It can be immediately used to display the computed volume. Just create a new SoVolumeData object and register the memory volume as the data source (volume reader) using the setReader method. Then build a VolumeViz scene graph as usual.

```
// Create new volume using memory volume as data source
SoVolumeData *pNewVolData = new SoVolumeData();
pNewVolData->ref();
pNewVolData->setReader( *pMemReader );
```

When you are finished using the memory reader remember to “unref” it to clean up allocated memory.

```
// Cleanup allocated memory and destroy object
pMemReader->unref();
```

End