Open Inventor Programming Guide
Seismic Visualization

Overview

Open Inventor is a power, general purpose toolkit for 3D visualization. The Open Inventor extension *VolumeViz* provides the features needed for managing and visualizing volume data, including, with the LDM option, extremely large data sets. VolumeViz is a powerful tool for any kind of volume data but is particularly well suited for visualizing 3D seismic data. Many features like volume "skin" rendering, Large Data Management (LDM) and multiple volume combining (co-rendering) were developed specifically to address requirements of seismic data applications. Other extensions like MeshViz and ScaleViz are also very useful for seismic applications, but in this tutorial we will focus more on core Open Inventor and VolumeViz.

Chapter 20 of the Open Inventor *User's Guide* provides an excellent introduction to programming with VolumeViz. If you are not familiar with the VolumeViz API, you should read this chapter first. However the User's Guide does not specifically discuss visualization of seismic data. In this tutorial we will go into more detail and show how to use Open Inventor and VolumeViz to implement some common features of seismic visualization applications. Other resources include the many example programs provided with Open Inventor and VolumeViz. The *VolRend* demo, a mini-application for visualizing volume data, is very useful for exploring the many rendering options available in VolumeViz. Loading data into VolRend can also be useful as a benchmark if your application does not produce the expected rendering results.

Programming languages: The Open Inventor API is available in specific versions for developers using C++, Java or C# (as well as other .NET conforming languages). The discussion in this document is generally applicable to all languages, but for simplicity the example code is currently only shown in C++. Because the API is intentionally similar in all languages, in most cases it is straightforward to convert the example code into Java or C#. For example:

```
C++:  pViewer->setBackgroundColor( SbColor( 0, .5f, .5f ) );

C#:   Viewer.SetBackgroundColor( new SbColor( 0, .5f, .5f ) );
```

Example programs: Starting with Open Inventor version 8.0, the example programs referenced in this document are installed with the SDK in directory: $OIVHOME/src/Inventor/samples/seismic.

Contents:

Section 1 – Loading SEGY Data:

1.1 – Overview

VolumeViz always accesses data through a volume reader object (an instance of a class derived from *SoVolumeReader*). A number of predefined readers are provided for common data formats, including SEGY. You can also create your own reader class to access a proprietary data format. See the VolumeViz chapter of the Users Guide for more information about that. If you specify a filename in the SoVolumeData node, VolumeViz will attempt to automatically determine the correct predefined reader to use, based on the file extension. For example, if the file extension is ".sgy" or ".segy", VolumeViz will automatically use the SEGY reader (*SoVRSegyFileReader*) to load the data. If the file extension is not recognized by VolumeViz, or if you are using a custom reader class, create an instance of the appropriate reader class and set it using the SoVolumeData method:

```
void setReader( SoVolumeReader &reader, SbBool takeOwnership );
```

Using the SEGY reader, VolumeViz can load a 3D volume directly from a SEGY format file. For example, we can directly load the file Waha8.sgy provided with the Open Inventor SDK. This can be useful for very small volumes or to extract information from the file and trace headers in the file. It is not recommended to directly load large seismic data sets to do visualization. Most seismic data sets should be converted into Mercury's LDM (Large Data Management) format.

Using LDM format it is possible to interactively visualize and navigate through extremely large data sets (hundreds of gigabytes). It is possible because the LDM converter breaks the data into chunks called "tiles" and also creates a multi-resolution hierarchy of tiles similar to an octree. Using an LDM format file VolumeViz is able to very quickly load the low resolution data and provide an initial image, while loading higher resolution data in background threads. While moving the camera, moving a slice, etc VolumeViz continues to maintain interactivity by using lower resolution data until higher resolution data become available. However in order to do the LDM conversion we must first be able to correctly load the data from a SEGY file.

1.2 – Problems with SEGY files

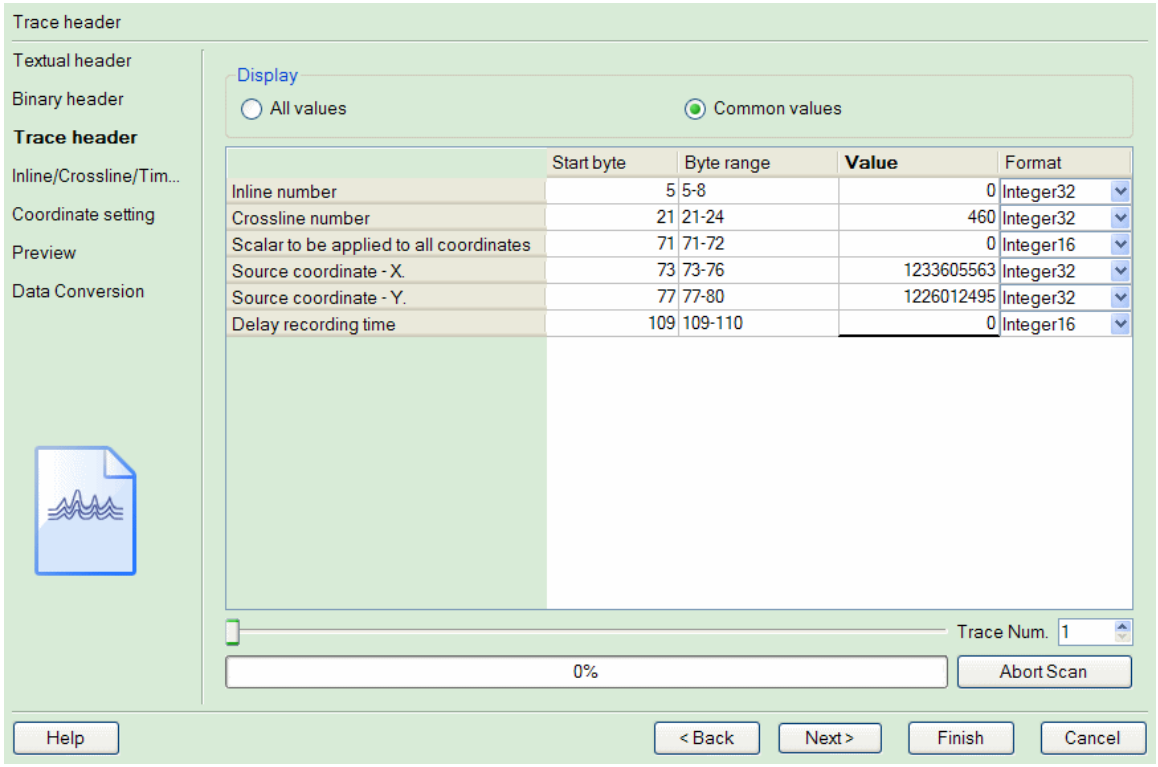
In many cases the SEGY reader can automatically determine the characteristics of a SEGY file and correctly load the 3D volume. The reader uses a simple detection algorithm by default to maximize performance, but more complex algorithms may be needed to correctly interpret the file. For example when there are a variable number of traces per line and/or a variable number of samples per trace. Some *SoPreferences* parameters are provided to tell the reader to use the more complex (but time consuming) algorithms. In these cases it may be possible for the application to detect that the file was not interpreted correctly and attempt to reload the file after setting the appropriate parameters.

In other cases the file may follow proprietary conventions that put critical values in non-standard locations in the trace header and/or store float values in integer trace header fields.

→ The *SoVRSegyTraceHeaderBytePosition* class allows you to specify the actual locations of trace attributes and/or the actual data type of those parameters.

In these cases it may be necessary to visually inspect the textual portion of the SEGY file header and/or have prior knowledge of the conventions used to write the file. Many seismic applications find it necessary to create a user interface to allow the user to modify these parameters. We will not address that task in this document, although we will discuss the classes and methods that can be used to do it.

The following image shows an example SEGY "Wizard" user interface (from Mercury's Avizo application). The file being read is the Waha8.sgy data set distributed with the Open Inventor SDK. You can see that by default the SEGY reader expects the Inline number at byte 5 (tracr) and the Crossline number at byte 21 (cdp). (Note we follow the SEGY convention of byte numbers starting at 1, rather than the programming convention of byte numbers starting at 0.) You can also see that using byte 5 the first Inline number is zero, and according to the file's text header it should be 720. In fact the Inline numbers are stored at byte 9 (fldr). But also note that the Source coordinate values do not look reasonable. That's because they are actually IEEE float values, even though all the trace header fields are specified to be integers. We can adjust for these differences using the *SoVRSegyTraceHeaderBytePosition* class as mentioned above. In this particular dialog the user is allowed to the change the start byte location by typing and to change the data type by selecting from a pull-down menu.



Example SEGY "Wizard" user interface from the Avizo application

Here are some of the problems commonly encountered with SEGY files and a brief description of the solution:

- Line and Crossline numbers stored at non-standard locations in the trace header.
  Use *SoVRSegyTraceHeaderBytePosition::setBytePosition* to specify the actual locations.

- Integer fields, e.g. survey coordinates, actually contain floating point values.
  Use *SoVRSegyTraceHeaderBytePosition::setByteFormat* to specify the actual format.

- File header says data is format 1 (IBM float), but data is actually IEEE float format.
  Use *SoPreferences::setBool* to set IVVR_SEGY_FLOATISIEEE.

- Reader does not correctly detect byte ordering (little-endian vs big-endian).
  Use *SoPreferences::setBool* to set IVVR_SEGY_SWAPBYTES.

- Reader does not correctly detect variable length traces.
  Use *SoPreferences::setBool* to set IVVR_SEGY_INCONSTANT_TRACE_LENGTH.

- Reader does not correctly detect variable number of traces per line.
  Use *SoPreferences::setBool* to set IVVR_SEGY_IRREGULAR_TRACE_NUM.

Section 1.3 – Troubleshooting SEGY files

If we are unable to read a SEGY file (or the resulting volume is incorrect), the next step could be to investigate the contents of the file. Specifically we want to look at the textual file header, the binary file header and the trace headers. If we're really not sure what convention was used to write the file, then we may need to look for a pattern in the primary values of the trace headers. There are tools available to help with this, but it may be convenient to use the Open Inventor SDK and walking through this will illustrate some of the powerful features of the SEGY reader.

It may be useful to enable debug output from the SEGY reader. This is easily done by using *SoPreferences::setBool* to set IVVR_SEGY_DEBUG. The resulting output may give some clues about how the reader attempted to interpret the file. Mercury Customer Support will normally ask for this output if a problem is reported reading a SEGY file. Note that on Windows the output will only appear on screen if the application was built as a "console" application or explicitly creates a console window. If the application is run in the Visual Studio debugger, the output will appear in the Visual Studio output pane. The application can also override the Open Inventor error handler and handle the strings as desired.

Let's create a small application to extract information about a SEGY file. We'll start, as usual, with initialization. But we don't need a toplevel window or a viewer, so we will only initialize "core" Open Inventor and the VolumeViz extension.

```
// Initialize (don't need any window system classes)
SoDB::init();
SoVolumeRendering::init();
```

Next create an instance of the SEGY reader class and tell it the name of the file we want to load. If the file can be opened and appears to be a SEGY file, this method will return zero.

```
// Create SEGY Reader and open file
SoVRSegyFileReader reader;
int rc = reader.setFilename( FILENAME );
```

The reader method *getSegyTextHeader* returns the text header as an *SbString* object. The text header should contain 40 lines of 80 characters. It is actually a single block of 3200 characters because there are no "newline" characters. We can use the *getSubString* method to print each line as a separate line of

output. By default the reader assumes the text header is in the traditional EBCDIC format. Call the method *setSegyTextHeaderAscii* to override this behavior. Also note that the reader supports the SEGY "Extended Textual File Header" feature, which allows the file to contain additional blocks of 3200 characters. The getSegyTextHeader method returns a single SbString object that contains the concatenation of all the text blocks. You can detect this case by calling the SbString method *getLength* to determine the total number of characters.

```
// Get/print text file header
// Should be 40 lines of 80 characters (as a block of chars)
SbString textHeader = reader.getSegyTextHeader();
if (!textHeader.isEmpty()) {
  printf( "Text header:\n" );
  for (int iline = 0; iline < 40; ++iline) {
    int start = iline * 80;
    SbString substr = textHeader.getSubString( start, start + 79 );
    printf( "%s", substr.getString() );
  }
}
```

The reader method *getSegyFileHeader* returns the binary header in an instance of the class *SoVRSegyFileHeader*. This class allows the binary file header fields to be conveniently accessed as member variables. For example we will probably want to know the sample interval, the number of samples per trace and the sample data type (format code).

```
// Get the binary file header
SoVRSegyFileHeader fileHeader;
SbBool ok = reader.getSegyFileHeader( fileHeader );
if (ok) {
  sampleInterval  = fileHeader.hdt;     // Byte 17
  samplesPerTrace = fileHeader.hns;     // Byte 21
  sampleFormat    = fileHeader.format;  // Byte 25
}
```

For our test data, Waha8.sgy, these values are:

Sample interval = 4000 micro-seconds,
Samples per trace = 876 samples and
Sample data type = format 8 (signed byte).

Now let's look at the trace headers. The *getNumTraces* method returns the total number of traces in the file. The content of a trace header is returned by the *getSegyTraceHeader* method. With these two methods, we could, for example, step through all the trace headers and determine the range (min and max values) of each trace header field (or at least the interesting ones). It may be useful to print the values of potentially useful fields for the first N trace headers. If we write these values into a file in CSV (Comma Separated Value) format, we can then load the file into a spreadsheet program like Excel for convenient viewing. Usually by knowing the expected range of inline and crossline numbers, or simply by seeing the pattern of repetition, we can determine which fields contain the inline and crossline numbers. The "TraceHeaders" example program fully implements this, similar to the following code.

```
// Get total number of traces
int numTraces = reader.getNumTraces();

// Initialize trace header min/max values for first N fields
const int nFields = 10;
```

```
int minValue[nFields], maxValue[nFields]
for (int i = 0; i < nFields; ++i) {
  minValue[i] = 1 << 30;
  maxValue[i] = -minValue[i];
}

// Column headers for trace header spreadsheet
fprintf( fp,
"Trace#,tracl(1),tracr(5),fldr(9),tracf(13),cdp(21),cdpt(25)\n" );

// Write trace header spreadsheet and find min/max values
SoVRSegyTraceIdHeader trHeader;
for (int i = 0; i < numTraces; ++i) {
  ok = reader.getSegyTraceHeader( i, trHeader );
  fprintf( fp, "%d,%d,%d,%d,%d,%d,%d\n", i+1,
    trHeader.tracl, trHeader.tracr, trHeader.fldr,
    trHeader.tracf, trHeader.cdp  , trHeader.cdpt );

  int *pField = &(trHeader.tracl);
  for (int j = 0; j < nFields; ++j) {
    if (*pField < minValue[j]) minValue[j] = *pField;
    if (*pField > maxValue[j]) maxValue[j] = *pField;
    pField++;
  }
}
```

Section 1.4 – Querying SEGY characteristics

One final technique that may be useful is to query the SEGY reader directly to see what it believes are the characteristics of the data. This is particularly useful when modifying byte positions and/or data formats. Using these query methods we can experiment until the correct / expected characteristics are found. And we can do that without wasting time trying to load the 3D volume (the actual trace data) using incorrect characteristics.

First we'll look at the range (min and max values) and the step size for inlines, crosslines and time (Z). For Waha8.sgy, we know from the file's text header that the inline range should be 720 to 978 by steps of 3. In fact by default we get 1 to 87 by steps of 1. The problem, as mentioned earlier, is that the SEGY reader expects to find the inline number at byte 5 (tracr) by default. For Waha8.sgy, we know from the file's text header that it's actually stored at byte 9 (fldr). Using the spreadsheet technique in the previous section, we could also easily see from the pattern of numbers in the trace headers that this value is stored at byte 9. In the next section we will see how to adjust byte positions. Given the correct range and step size we can compute the actual number of inlines and crosslines.

```
// Get range and step size
int ilFrom, ilTo, ilStep;
int clFrom, clTo, clStep;
int  zFrom,  zTo,  zStep;
reader.getInlineRange   ( ilFrom, ilTo, ilStep );
reader.getCrosslineRange( clFrom, clTo, clStep );
reader.getZRange        (  zFrom,  zTo,  zStep );

// Compute number of Lines/Crosslines
int numIlines = ilTo - ilFrom + 1;
```

```
if (ilStep > 0)
  numIlines = (int)ceil((float)numIlines / ilStep);
int numXlines = clTo - clFrom + 1;
if (clStep > 0)
  numXlines = (int)ceil((float)numXlines / clStep);
```

The *getDataChar* (get data characteristics) method returns the characteristics of the 3D data volume that the reader will extract from the SEGY file. *volSize* is the geometric extent of the volume in 3D space, in other words the bounding box. By default this is an arbitrary bounding box computed from the voxel dimensions of the volume. We will discuss in other sections how to use the actual survey coordinates and how to scale the time axis to give a more useful display. *dataType* indicates the type of data values in the volume. For Waha8.sgy, the data type should be SIGNED_BYTE, corresponding to SEGY format 8. *volDim* is the dimensions of the volume in voxels. For Waha8.sgy, the volume dimensions should be 876 x 67 x 87. Remember that the SEGY reader loads traces into contiguous memory, so this is samples x crosslines x lines.

```
// Get the characteristics of the 3D volume
SbBox3f volSize;
SbVec3i32 volDim;
SoVolumeData::DataType dataType;
reader.getDataChar(  volSize, dataType, volDim );
float xmin,ymin,zmin,xmax,ymax,zmax;
volSize.getBounds( xmin, ymin, zmin, xmax, ymax, zmax );
```

The *getP1P2P3P4Coordinates* method returns the coordinates of the corners of the survey as double precision 2D vectors. It also returns a boolean value indicating if the survey should be interpreted in a left-handed or right-handed coordinate system. In this case we expect P2 to be approximately 1130370,604625 (based on the textual header). But by default the X value of P2 is returned as 1233779735. The problem, as mentioned earlier, is that the SEGY reader expects all trace header values to be integers (as stated in the spec) and these values are actually stored as floating point. In the next section we will see how to adjust the data format.

```
// Get the coordinates of the corners of the survey
SbVec2d p1, p2, p3, p4;
SbBool rightHanded = reader.getP1P2P3Coordinates( p1, p2, p3, p4 );
```

Finally, we might want to check what byte positions the reader actually used to determine the inline and crossline numbers. The *getSegyTraceHeaderBytePosition* method returns an SoVRSegyTraceHeaderBytePosition object which contains the byte positions of all the trace header fields. Use the *getBytePosition* method to get the position of a specific header field. As expected, we see that by default the reader is expecting the inline number at byte 5 and the crossline number at byte 21.

```
// Get the actual byte positions the reader used
SoVRSegyTraceHeaderBytePosition
    bytePos = reader.getSegyTraceHeaderBytePosition();
int ilineByte = bytePos.getBytePosition(
    SoVRSegyTraceHeaderBytePosition::SEGY_INLINE );
int xlineByte = bytePos.getBytePosition(
    SoVRSegyTraceHeaderBytePosition::SEGY_CROSSLINE );
```

Section 2 – Working with Slices

Section 2.1 – Overview

In this section we will discuss rendering and interacting with slices. Typically the first thing we want to do with a new seismic volume is move some slices throught it. VolumeViz provides two slice primitives, called *SoOrthoSlice* and *SoObliqueSlice*. ObliqueSlice is the more general primitive. This slice can be placed at any location in 3D space and at any orientation. An OrthoSlice is constrained to be axis aligned and positioned in voxel coordinates. This slice has better rendering performance and corresponds to the conventional notion of an inline, crossline or time slice. The *SoVolumeSkin* primitive is effectively a collection of OrthoSlices constrained to lie on the faces of the current subvolume (ROI or Region of Interest). SoVolumeSkin is very useful for rendering opaque volume probes. We will discuss this primitive separately, but much of the rendering discussion is the same as for slices.

An OrthoSlice is positioned in voxel coordinates using the *sliceNumber* field. The slice number starts at zero and goes to the volume dimension minus one along the OrthoSlice's current axis. Knowing, for example, the range of inline numbers, we can easily convert an inline number to the corresponding slice number. Just keep in mind that slices are numbered in the volume's voxel coordinate system. You specify the current axis using the *axis* field, which takes the enum values X, Y or Z. Again these correspond to the volume's voxel coordinate system.

Section 2.2 – Selecting a slice

How will we know which slice the user wants to manipulate? One way could be through the application's user interface. Typically a seismic application will provide a "tree view" of all the surveys loaded and objects created, such as slices and subvolumes. We might allow the user to make a particular slice "current" by selecting it in this tree view. However it is very important to allow users to select and manipulate objects directly in the 3D window. The user will normally select an object by pointing to it with the mouse cursor and clicking a mouse button. At the lowest level, detecting and acting upon mouse motion and button clicks is called "event handling". Determining which graphical object is underneath the mouse cursor is an intermediate level operation called "picking" in 3D graphics. Selection of an application object is a higher level operation that may occur as a result of picking. Open Inventor provides powerful and flexible tools for event handling, picking and selecting objects. You can find a lot of details about this in Chapter 10 of the *The Inventor Mentor*. For now we will just discuss a very convenient high-level tool called the *SoSelection* node.

SoSelection is derived from SoSeparator and is normally placed near the top of the scene graph. All objects below the SoSelection node will (by default) be selectable by the user. Objects that we never want the user to select, for example a rendering of the colormap, can be kept in part of the scene graph that is not below the SoSelection. Objects that must be below the SoSelection for organizational reasons, or are temporarily not selectable, can be protected from selection using an *SoPickStyle* node with the *style* field set to UNPICKABLE. When Open Inventor handles a mouse button down event, the SoSelection node automatically does picking to determine what (if any) graphical object is under the cursor. If a valid object is found, SoSelection adds the *path* to that object (see *SoPath* and Chapter 3 of the *The Inventor Mentor*) to its *selection list*. By default, if a different path is already in the list, that path is first removed from the list. You can modify this behavior if needed. The important thing is that whenever a path is added to, or removed from, the selection list, SoSelection will notify the application.

As usual in Open Inventor for C++, the notification is done by calling a callback function. Starting with our basic program that loads a SEGY volume and displays a slice, we'll first include the necessary header files and declare the callback functions. There are actually two functions. The first one will be called when an object is selected and the second one will be called when an object is "deselected" or removed from the selection list. We will also create a "global" variable to store the address of the currently selected slice. (Of course in a real application we would use something more elegant than a global variable.)

```
#include <VolumeViz/nodes/SoSelection.h>

// Selection callback functions
void selectionCB( void *pUserData, SoPath *pPath )
void deselectionCB( void *pUserData, SoPath *pPath )

static SoOrthoSlice *pCurrentSlice = NULL;
```

Next we'll create the SoSelection node and tell it what functions to call for selection and de-selection changes. Remember that the basic program builds its scene graph under an SoSeparator and tells the viewer to display that Separator (variable *root*). We'll keep most of that code the same. After building the scene graph, we'll create an SoSelection node, put the Separator under it and tell the viewer to display the SoSelection node.

```
  SoSelection *pSelRoot = new SoSelection();
  pSelRoot->addSelectionCallback  ( selectionCB   );
  pSelRoot->addDeselectionCallback( deselectionCB );
  pSelRoot->addChild( root );
  ...
  myViewer->setSceneGraph( pSelRoot );
```

Now we'll implement the selection and de-selection callback functions. The selection function is called with some optional user data, which we do not currently use, and the path to the selected node in the scene graph. The first thing we'll do is get the last node in the path, the "tail" of the path, as a generic node. This is the actual geometry node that was picked. This check is unnecessary right now because we can only get a slice, but the scene graph will be more complicated later. If the picked node really is an SoOrthoSlice, then we can cast to the actual type. For now we'll just store this address as our current slice and print a message to confirm that the selection worked. Later we might want to determine what application object this corresponds to and, for example, highlight that slice in the tree view. The de-selection callback is similar.

```
void selectionCB( void *pUserData, SoPath *pPath )
{
  SoNode *pNode = pPath->getTail();
  if (pNode->isOfType(SoOrthoSlice::getClassTypeId())) {
    pCurrentSlice = (SoOrthoSlice *)pNode;
    std::cout << "Slice selected\n";
  }
}

void deselectionCB( void *pUserData, SoPath *pPath )
{
  SoNode *pNode = pPath->getTail();
  if (pNode->isOfType(SoOrthoSlice::getClassTypeId())) {
    pCurrentSlice = NULL;
    std::cout << "Slice DEselected\n";
```

```
    }
}
```

If you build and run the example program “Slice_Selecting_a_slice”, note that you must switch the viewer into selection mode before you can select the slice. If you see the “hand” cursor, the viewer is in viewing mode. Click the “arrow” button on the right-hand side of the viewer or press the ESC key to change modes. Note that the selection callback is triggered the first time you click on the slice, but not on subsequent (redundant) clicks on the slice. However clicking anywhere not on the slice triggers the de-selection callback – the slice is no longer selected.

Section 2.3 – Moving a slice

Let’s implement a simple mechanism for moving a slice. Later we will implement a more complicated mechanism that is more convenient for the user. But this mechanism will illustrate how to move a slice in response to user events and also how to handle events in a platform independent way. Our goal is to move the currently selected slice using the up and down arrow keys. (Any keys could be used.) Specifically, when the up arrow key is pressed, we want to increment the *sliceNumber* field of the currently selected slice.

Open Inventor provides a set of platform (and window system) independent event classes derived from the *SoEvent* class. There are classes for key presses, mouse clicks and mouse motion, as well as specialized events used, for example, in immersive VR environments. In this case we are interested in the *SoKeyboardEvent* class, which represents key press and release events.

We already know that if we are using one of the viewer classes, and the viewer is in *viewing* mode (the “hand” cursor), then generally events are handled directly by the viewer. When the viewer is in *selection* mode, Open Inventor provides very flexible event handling by passing events through the scene graph. The viewer does this automatically using an *SoHandleEventAction* object. Similar to a Render action, the HandleEvent action traverses the scene graph, visiting each node in order and allowing the node to handle the event. Most nodes do not pay any attention to this action, but later we will discuss some special nodes called *draggers* that handle events and automatically modify themselves. Here we will discuss the general purpose *SoEventCallback* node that allows the application to handle events in a callback function.

We’ll start, as usual, by adding the header files for SoEventCallback and SoKeyboardEvent and declaring the callback function for handling events.

```
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/events/SoKeyboardEvent.h>

// Event handling callback function
void eventCB( void *pUserData, SoEventCallback *pNode );
```

Next we’ll create an SoEventCallback node and tell it what functions to call for various events. When we add a callback to this node, we specify both the function to be called (eventCB) and the type of event that should trigger the call. Therefore a single event callback node could call different functions for keyboard and mouse events or a single callback function could also handle many different kinds of events. We specify the event type using Open Inventor’s built-in type system. The static method *getClassTypeId* returns the type id.

```
// Create an event callback node to detect key presses
```

```
SoEventCallback *pEventCB = new SoEventCallback();
pEventCB->addEventCallback(
  SoKeyboardEvent::getClassTypeId(), eventCB );
```

Now we'll implement the event callback function. This function is called with some optional user data, which we do not currently use, and a pointer to the node that handled the event. The first thing we'll do is get the actual event that was handled (pEvent). Now we need to know several things. First, is this really a key press event (which is currently always true in this program, but may not be true later). Second, is it one of the keys that we actually care about, specifically the up or down arrow key. And third, is it a key press or release. This is important because we only want the slice to move when the arrow key is pressed. If we only checked for the correct key we would actually move the slice on the press and on the release, which would probably surprise the user. The SoKeyboardEvent class provides a convenient macro for this test that takes the event pointer and the desired key and performs all the checks we just discussed. If the event is an up arrow key press then we can use our global variable *pCurrentSlice* to get the current slice position from the *sliceNumber* field, increment that number and set it as the new slice number. As usual, Open Inventor will automatically re-render the scene when a node is modified.

```
// This function is called when an event is handled by
// an SoEventCallback node.
void eventCB( void *pUserData, SoEventCallback *pNode )
{
  const SoEvent *pEvent = pNode->getEvent();
  if      (SO_KEY_PRESS_EVENT(pEvent,UP_ARROW)) {    // Increment
    if (pCurrentSlice) {
      int sliceNum = pCurrentSlice->sliceNumber.getValue();
      pCurrentSlice->sliceNumber = sliceNum + 1;
    }
  }
  else if (SO_KEY_PRESS_EVENT(pEvent,DOWN_ARROW)) {  // Decrement
    if (pCurrentSlice) {
      int sliceNum = pCurrentSlice->sliceNumber.getValue();
      pCurrentSlice->sliceNumber = sliceNum - 1;
    }
  }
}
```

If you build and run the example "Slice – Moving_a_slice", note that the viewer must be in selection mode both to select the slice and to handle the key presses.

You may have noticed a major flaw in the above code. Specifically that we allow the user to increment and decrement the slice number beyond the valid range of slice numbers. VolumeViz clamps this value internally, so on the screen the slice simply stops moving at the limits of the volume. But we would like the value in the sliceNumber field to match the image on the screen. To do this we need access to the volume data node, or at least to the volume dimensions. We could set a pointer to this data as the callback function's "user data", but for now we'll just make it part of the global "application state" accessible from the function. Now we can test the slice number and only modify the value if it is between zero and the actual volume dimension on the appropriate axis. We can take advantage of the fact that the enum returned by the slice's *axis* field is simply the integer value 0, 1 or 2 corresponding to the X, Y or Z axis.

```
void eventCB( void *pUserData, SoEventCallback *pNode )
{
  const SoEvent *pEvent = pNode->getEvent();
  if      (SO_KEY_PRESS_EVENT(pEvent,UP_ARROW)) {    // Increment
```

```
    if (pCurrentSlice) {
      int sliceNum = pCurrentSlice->sliceNumber.getValue();
      int axis     = pCurrentSlice->axis.getValue();
      if (sliceNum < (volumeDimensions[axis]-1))
        pCurrentSlice->sliceNumber = sliceNum+1;
    }
  }
  else if (SO_KEY_PRESS_EVENT(pEvent,DOWN_ARROW)) {  // Decrement
    if (pCurrentSlice) {
      int sliceNum = pCurrentSlice->sliceNumber.getValue();
      if (sliceNum > 0)
        pCurrentSlice->sliceNumber = sliceNum-1;
    }
  }
}
```

Our simple example program is already more general than you might think. For example, if the program creates multiple slices, the selection and event handling code allows the user to select any of the slices and move the selected slice using the arrow keys. Let's add a crossline slice and try it out (see the example program "Slice – Moving_two_slices").

```
// Create a crossline slice and position at mid-point of volume
SoOrthoSlice *pCrosslineSlice = new SoOrthoSlice();
pCrosslineSlice->axis = SoOrthoSlice::Y;
pCrosslineSlice->sliceNumber = volumeDimensions[1] / 2;
...
root->addChild( pCrosslineSlice );
```

It should be clear that we could also allow the user to interactively move a slice by simply providing a user interface control (an integer valued slider for example) and responding to changes in the control by modifying the sliceNumber field. We could even eliminate the inconvenient (for the user) selection step by providing a separate slider for each slice. However the preceding code illustrates some generally useful techniques in Open Inventor, including handling events and referring back to the volume data node to validate user input.

In fact our goal should be to allow the user to manipulate the slice position directly. By that we mean that the slice itself should be viewed as a visualization tool, not simply a result of the visualization. As a corollary the user should not be required to switch focus away from the slice in order to manipulate it. Both of the previously discussed solutions have this flaw. The user has to select with the mouse then modify with the keyboard or else look away from the slice to position the cursor over the corresponding slider. We'll talk about how to do more direct manipulation in section 3.6.

Finally, note that our example program creates a very simple scene graph structure. For example, all the OrthoSlice nodes are created at the top level of the scene graph. In a real application the structure would be more detailed to allow grouping related objects, controlling the visibility of individual objects, etc. For example, instead of creating every slice at the top level we might do something like the following code. We put an *SoSwitch* above the slice to allow the user to toggle the visibility of individual slices (we could add/remove them from the scene graph, but using a switch is more efficient). We toggle visibility by changing the switch's *whichChild* field between SO_SWITCH_ALL (-3) and SO_SWITCH_NONE (-1). It's good practice to always put an SoSeparator immediately below a switch because the SoSwitch node does not do any caching. It's also convenient if we need to group some property nodes with the slice, because SoSwitch does not save and restore traversal state. In this case, for example, we might put an instance of a shared color map (SoTransferFunction) with the slice, to allow the user to apply different

maps to different slices.

```
// Create a slice
SoSwitch *addSlice( ... slice properties ... )
{
  SoSwitch     *pSwitch = new SoSwitch;        // Visibility
  SoSeparator  *pSep    = new SoSeparator;     // Group properties
  SoOrthoSlice *pSlice  = new SoOrthoSlice();  // Actual slice
  // Initialize slice axis, position, etc
  ...
  pSwitch->addChild( pSep );
  pSep->addChild( pSlice );
  return pSwitch;
}
```

Section 2.4 – Animating slices

We discussed that an OrthoSlice is positioned in voxel coordinates using the *sliceNumber* field. Normally if your application modifies the sliceNumber field, Open Inventor will automatically re-render the scene showing the slice in its new position. (This behavior can be disabled using the viewer's *setAutoRedraw* method, but it's usually convenient to leave this enabled.) However it is important to understand that the automatic redraw does not happen *immediately* after the field is modified. That would be very inefficient if the application needed to make multiple changes to the scene graph to implement a single "logical" change. Instead modification of the field causes a redraw to be "scheduled" for later. If the application does not explicitly request a redraw (this is not normally necessary), then when control returns to the application's main event loop a timer event will be handled by Open Inventor and the redraw will be performed.

Suppose we want to animate a slice, for example we want a slice to sweep back and forth through the volume. We can extend our keyboard event callback so that pressing 'A' starts animation of the currently selected slice. If we start by just animating once through the volume, our first experiment might look something like the following code

```
  else if (SO_KEY_PRESS_EVENT(pEvent,A)) {
    if (pCurrentSlice) {
      int sliceNum = pCurrentSlice->sliceNumber.getValue();
      int axis     = pCurrentSlice->axis.getValue();
      int limit    = volumeDimensions[axis];

      for (int snum = 0; snum < limit; ++snum) {
        pCurrentSlice->sliceNumber = snum;
      }
    }
    else
      cout << "Animate: No slice selected!\n";
  }
```

First we check that a slice is selected, then get the current slice, current axis and the limit (maximum slice number on that axis). The important part is the "for" loop that just steps through all the slice positions from zero to the limit. There are a lot of little issues with this code (only animates through once, doesn't animate at a constant speed, doesn't remember / return to the starting slice position, etc). But fundamentally this code just doesn't draw the right picture! The user will only see the slice rendered at

the last position. As explained at the beginning of the section, this is because each change to the scene only *schedules* a redraw. The automatic redraw cannot occur until control returns to the event loop.

There is a "brute force" solution to the problem. At any time we can force the viewer to re-render the scene graph by calling its *render()* method. We'll add a pointer to the viewer object to the global "application state" (OK, it's just another static variable) and use it in the callback like this:

```
static SoXtViewer *spViewer = NULL;

...

      for (int snum = 0; snum < maxDim; ++snum) {
        pCurrentSlice->sliceNumber = snum;
        spViewer->render();      // <-- Force immediate rendering
      }
```

This code has the desired (visual) effect. Unfortunately it probably won't pass usability testing, because there's no way the user can stop the animation once it's started.

What we need is a real time driven animation and Open Inventor has several tools for that. We will use the *SoTimerSensor* class (also consider *SoAlarmSensor*). Like all the sensor classes, SoTimerSensor executes a callback function when it triggers. SoTimerSensor Sensor triggers the callback repeatedly at a regular interval and automatically re-schedules itself. The sensor continues running until the application calls the unscheduled() method.

First we need to add an instance of the timer and a desired animation speed (how many slice positions to move per second) to the global application state:

```
#include <Inventor/sensors/SoTimerSensor.h>
. . .
static SoTimerSensor *spSliceTimer    = NULL;
static double         sSliceAnimSpeed = 20;  // steps per second
```

Next we re-write the event handler that initiates the animation. Let's look at that code step by step. First we check if a slice is selected and create the timer sensor if it doesn't already exist. When we create the sensor we specify the function to call when it triggers. We'll look at the implementation of this function next.

```
  else if (SO_KEY_PRESS_EVENT(pEvent,A)) {
    if (pCurrentSlice) {
      // Create time sensor if necessary
      if (! spSliceTimer)
        spSliceTimer = new SoTimerSensor( sliceAnimCB, NULL );
```

Next, check if the animation is already running. We can detect this by checking if the timer sensor is already scheduled (meaning that it is active). If the animation is currently running, stop it by calling the sensor's unschedule method:

```
      // If time sensor already running, stop it
      if (spSliceTimer->isScheduled()) {
        spSliceTimer->unschedule();
        cout << "  Stop animation\n";
      }
```

Else the animation is not currently running, so first convert the desired animation speed to a time interval in seconds (this will just be the inverse of slice positions per second). Set the sensor's baseTime to "now" and set the desired time interval between invocations of the callback function. Start the sensor running by calling its schedule method.

```
      // Else recompute animation speed and start time sensor
      else {
        cout << "Begin animating...\n";
        double timeInterval = 1 / sSliceAnimSpeed;
        spSliceTimer->setBaseTime( SbTime(0.) );
        spSliceTimer->setInterval( SbTime(timeInterval) );
        spSliceTimer->schedule();
      }
    }
    else {
      cout << "Animate: No slice selected!\n";
    }
```

In the slice animation callback function we get the slice number, axis and limit as before and increment the slice number. Since we are implementing the same "once through" behavior as our first prototype then if the slice is at the limit we stop the animation by calling unschedule, else we update the sliceNumber field. This is sample program "Slice_animate_timer1".

```
void sliceAnimCB(void *data, SoSensor *sensor)
{
  // Get current slice number, axis and maximum slice number
  int sliceNum = pCurrentSlice->sliceNumber.getValue();
  int axis     = pCurrentSlice->axis.getValue();
  int limit    = volumeDimensions[axis];

  // If slice at limit, stop animation, else update slice number
  sliceNum++;
  if (sliceNum >= limit)
    spSliceTimer->unschedule();
  else
    pCurrentSlice->sliceNumber = sliceNum;
}
```

This seems to work, so let's complete our original objective and sweep back and forth through the volume. Here is the modified animation callback, after fetching the slice number, axis and limit. We have a static variable that remembers whether we're animating forward or backward through the slice range. Increment or decrement the slice number appropriately and if we've hit the limit, reverse direction for the next invocation. We skip one slice position so the animation won't seem to "pause" at each end of the range. In this version the animation will not stop until the user presses 'A' again, which causes the event handler to call unschedule.

```
  static bool sliceAnimForward = true; // Initially animating up

    if (sliceAnimForward) {       // Animating forward
      sliceNum++;                 // Increment sliceNum
      if (sliceNum >= limit) {
        sliceNum  = limit - 2;    // At limit, start
        sliceAnimForward = false; // animating backward
```

```
    }
  }
  else {                        // Animating backward
    sliceNum--;                 // Decrement sliceNum
    if (sliceNum < 0) {
      sliceNum  = 1;            // At limit, start
      sliceAnimForward = true;  // animating forward
    }
  }
  pCurrentSlice->sliceNumber = sliceNum;
```

If you build and run this version of the program ("Slice_animate_timer2") you will see that the slice animates back and forth continuously but also starts and stops when the 'A' key is pressed. Because the event loop is executed between slice positions, the application continues to respond to the user. For example you can use the viewer to move the camera around while the slice is animating. This still isn't quite "production quality" code, but the basics are in place.

Of course the timer interval and the frame rate will only be exactly the same if rendering takes no time at all. That's not a realistic assumption and rendering time may depend on the volume size, rendering effects (e.g. shader complexity), window size, graphics hardware performance and other factors. It may be better to give the user a speed control labeled "Slow" and "Fast" rather than specific speeds. Also note that, as written, our animation is guaranteed to render every slice, no matter how long it takes. In many cases this is the desired result (we don't want to skip any slices). We could also imagine an "adaptive" animation callback that calculates where we are, or should be, in the animation cycle and "jumps" to the appropriate slice.

Section 2.5 – Moving slices interactively

As mentioned previously our goal should be to allow the user to manipulate the slice position directly. By that we mean that the slice itself should be viewed as a visualization tool, not simply a result of the visualization. As a corollary the user should not be required to switch focus away from the slice in order to manipulate it. Both of the previously discussed solutions have this flaw. The user had to select with the mouse then modify with the keyboard or else look away from the slice to position the cursor over the corresponding slider bar. In this section we will consider how to implement direct manipulation of slices.

To accomplish this we will use one of the most powerful and flexible tools in Open Inventor, called a *dragger* (see SoDragger). What makes the dragger such a powerful tool is:

- Draggers convert 2D motion into 3D motion.
  We don't normally have a 3D input device, so we have to improvise with the mouse. The mouse can only move along 2 axes (up-down and left-right), but we need to move objects in the scene in arbitrary directions in 3D. Draggers convert mouse motions into 3D motions in an intuitive and convenient way.

- Draggers have *constraints* that make manipulation easier and more precise.
  You can easily see this by using the middle mouse button (or shift-left button) in the viewer to move the scene. It's quite difficult to move the scene exactly in a straight line! Using draggers we can constrain motion in a way that is appropriate for the user's task – on a straight line, in a plane, around a circle, around a sphere, and so on.

What makes draggers so flexible is two things:

- "Simple" draggers can be combined to make complex "composite" draggers.
  For example the TabBoxDragger, commonly used to manipulate volume probes in seismic applications. This dragger combines six instances of the TabPlaneDragger.

- The geometry displayed by the dragger can be changed to anything you want.
  In addition to replacing the dragger's default geometry with different shape nodes, you can tell the dragger to use any shape already in the scene graph as a "proxy" geometry. The proxy geometry does not become part of the dragger, but any input events on the proxy are handled "as if" that geometry was part of the dragger. For example, we will use this feature to make the actual SoOrthoSlice node be the proxy geometry for an *SoTranslate1Dragger*, allowing the user to directly click and drag the slice.

To experiment with the standard behavior of the Translate1Dragger: start the SceneViewer (pre-built on Windows, otherwise found in the SDK under src/Inventor/Demos) or any simple Inventor program and load the file "translate1Dragger.iv". You should see a horizontal bar with an arrowhead (cone) on each end. This is the default geometry for the Translate1 dragger, called the "translator" part in the nodekit. Switch to selection mode (press the ESC key) and click on the dragger. On mouse-down the dragger changes from white to yellow. This is actually a second, separate geometry called the "translatorActive" part, which the dragger displays when it is being actively manipulated. If you move the mouse while holding the button down, the dragger will move from side to side. Notice that the dragger is *constrained* so it can only move along a single axis, hence the name Translate1Dragger.

What we would like to implement we might call an "SoOrthoSliceManip" node. Remember that in Open Inventor a *manipulator* is a derived class that combines a standard node with a built-in dragger, allowing the user to directly manipulate some property of the node. For example, an SoTrackballManip is derived from SoTransform and includes an SoTrackballDragger. The complete implementation of an ortho slice manipulator is a little too in-depth for this document, so we'll focus on creating the tools needed and demonstrating the functionality. Specifically we'll create an ortho slice *dragger* and "manually" connect it to the slice.

[NOTE: Open Inventor 8.0 now includes a pre-built *SoOrthoSliceDragger* node, however the following discussion is still useful to better understand how to use draggers in your application.]

First we'll need a way for the user to indicate which slice should be dragged. One way to do that is to use the selection mechanism discussed in previous sections. In the selection callback function we could connect our custom dragger to the newly selected slice. However this turns out to be "awkward" for the user because it effectively requires two clicks to drag the slice: one click to trigger the selection callback, then another click to initiate dragging. We still need the selection mechanism so the user can select a slice in order to change its properties, but we will take a different approach for dragging. We will use a special feature of draggers so that when the user clicks and drags on a slice, it will respond immediately.

We will always drag ortho slices (inline, crossline and time) along a single axis, so the *SoTranslate1Dragger* is an appropriate starting point. We want to use the actual slice as the "proxy" geometry for the dragger, so we'll need to use this method:

```
setPartAsPath( const SbName &partName, SoPath *proxyPath )
```

from *SoInteractionKit* (a parent class of SoTranslate1Dragger). As you can see we need a *path* to the proxy geometry. This is not as complicated as it might sound. When events are handled in the scene

graph, the dragger just checks that the specified proxy path is *part of* the pick path. In other words we don't need a path all the way from the top of the scene graph. In fact a trivial path just containing the slice node itself will usually work here, since we don't expect the slice to ever be instanced more than once in the scene graph.

All the setup and specific behaviors we need from our dragger can be done using the standard SoTranslate1Dragger class. However it will be convenient to encapsulate this code in a new class "orthoSliceDragger" which is derived from SoTranslate1Dragger. Here are the main issues we need to address in this class:

- Dragger direction.
  The translate1 dragger is constrained to only move along the X axis, but we need to drag slices along whichever axis the slice is oriented to. Luckily it is more correct to say that the translate1 dragger is constrained to only move along its *local* X axis. So we can rotate the dragger to *appear* to slide along any arbitrary axis in 3D. In fact we can do this very conveniently by initializing the dragger's *motionMatrix* with a rotation. See the setDraggerDirection method in the custom class.

- Dragger position versus slice number.
  As the dragger moves it updates its position, but this position is in 3D units and the slice is positioned in voxel units (slice number). Therefore we cannot simply connect the dragger position to a slice node, like we can to a transform node. We will implement utility functions to convert from position to slice number and the reverse. This is fairly straightforward because we know the extent of the volume in 3D space. See the synchronize and valueChangedCB methods in the custom class.

Here is the declaration of the slice dragger class (see orthoSliceDragger.h):

```
class orthoSliceDragger : public SoTranslate1Dragger
{
public:
  // Constructor
  orthoSliceDragger();

  // Associate with slice and volume
  void initialize( const SoPath *, const SoVolumeData * );

  // Update dragger position from slice number
  void synchronize();

  // Rotate dragger to match slice orientation
  void setDraggerDirection();

protected:
  // Update slice number from dragger position
  static void valueChangedCB( void *data, SoDragger *dragger);

  SoOrthoSlice *m_orthoSlice;
  SoVolumeData *m_volumeData;
};
```

In the slice dragger's constructor we will simply "remove" the default geometry normally used by the dragger for its inactive and active (dragging) states. We will be using the actual ortho slice as our inactive

geometry. We will not use any active geometry for now, however it could be useful to use something adapted to the slice, so the user has a visual signal that dragging is active. To remove (or otherwise modify) the dragger's geometry, use the *setPart* method and the part names shown on the dragger's help page in the "Dragger Resources" section. There needs to be something in place of those parts, so we'll just provide an empty Separator.

```
orthoSliceDragger::orthoSliceDragger()
    : m_orthoSlice(0), m_volumeData(0)
{
  setPart( "translator",       new SoSeparator );
  setPart( "translatorActive", new SoSeparator );
}
```

The initialize method is provided to associate the dragger with a specific slice and volume. If no slice is specified, then disconnect from any current slice and return. Store pointers to the slice node and the associated volume data node. Rotate the dragger to match the orientation of the slice (we'll look at this method next). Make the slice geometry the proxy for the draggers inactive part. Note that an action will be applied during this operation, which will ref and unref the dragger. This would result in the dragger being destroyed if its ref count is the default (zero). Probably the dragger is already in the scene graph and therefore ref'd by its parent node, but to be safe we temporarily increment the ref count here. Move the dragger to the current slice position. Finally add a value changed callback. The dragger will call this function when its position changes and the function will make the corresponding change to the slice position.

```
void orthoSliceDragger::initialize( const SoPath *path,
                                    const SoVolumeData *volData )
{
  // Disconnect and reset if no slice specified
  if (path == NULL || volData == NULL) {
    if (m_orthoSlice != NULL)
      setPartAsPath( "translator", NULL );
    m_orthoSlice = NULL;
    m_volumeData = NULL;
    return;
  }

  // Get ortho slice node from path and save volume data ptr
  m_orthoSlice = (SoOrthoSlice*)(path->getTail());
  m_volumeData = const_cast<SoVolumeData *>(volData);

  // Rotate dragger to match orientation of slice
  setDraggerDirection();

  // Make the slice geometry the proxy for the dragger
  ref();
  setPartAsPath( "translator", (SoPath *)path );
  unrefNoDelete();

  // Move dragger to current slice position
  synchronize();

  // Connect callback to update slice position when dragger moves
  addValueChangedCallback( (SoDraggerCB *)&valueChangedCB, NULL );
}
```

Here is how we set the orientation of the dragger. If the slice is on the X axis we don't really have to do anything. If the slice is on the Y or Z axis we set the matrix to contain the appropriate 90 degree rotation. We do this using the dragger's *motionMatrix*. The motion matrix places the dragger relative to its parent space, normally the 3D world coordinates. The dragger also updates this matrix during dragging operations, but it's better to use the dragger's fields rather than the matrix directly.

```
void orthoSliceDragger::setDraggerDirection()
{
  int sliceAxis = m_orthoSlice->axis.getValue();
  SbMatrix mat;
  switch (sliceAxis) {
  case SoOrthoSlice::X :
    mat.makeIdentity();
    break;
  case SoOrthoSlice::Y :
    mat.setRotate( SbRotation( SbVec3f(0,0,1), 1.5707963267f ) );
    break;
  case SoOrthoSlice::Z :
    mat.setRotate( SbRotation( SbVec3f(0,1,0), -1.5707963267f ) );
    break;
  }
  setMotionMatrix( mat );
}
```

Next we need a method to set the position of the dragger from the current slice number. This is straightforward arithmetic. Given the slice position in voxels and the voxel dimensions of the volume, we can compute the slice position as a "fraction" of the volume along the slice axis. Then knowing the extent of the volume, we can compute the slice position in 3D coordinates and set that value in the dragger. Note that we disable the value changed callbacks before updating the draggers position, because our value changed callback should only be called when the user changes the dragger position. That wouldn't be a problem in this simple case, but it's a good idea to always do this to avoid the possibility of an "endless loop" of callbacks.

```
void orthoSliceDragger::synchronize()
{
  // Get current dragger position
  float x, y, z;
  translation.getValue().getValue( x, y, z );

  // Get slice position / orientation
  int sliceNumber = m_orthoSlice->sliceNumber.getValue();
  int sliceAxis   = m_orthoSlice->axis.getValue();

  // Get volume extent in 3D
  SbVec3i32 volDim = m_volumeData->getDimension();
  SbBox3f volBox = m_volumeData->extent.getValue();
  SbVec3f volMin = volBox.getMin();
  SbVec3f volMax = volBox.getMax();

  // Convert slice number to fraction of volume dimension.
  // Then compute dragger position from volume extent.
  float fraction = 0;
  switch (sliceAxis) {
  case SoOrthoSlice::X :
```

```
    fraction = (float)sliceNumber / (float)(volDim[0] - 1);
    x = volMin[0] + fraction * (volMax[0] - volMin[0]);
    break;
  case SoOrthoSlice::Y :
    fraction = (float)sliceNumber / (float)(volDim[1] - 1);
    y = volMin[1] + fraction * (volMax[1] - volMin[1]);
    break;
  case SoOrthoSlice::Z :
    fraction = (float)sliceNumber / (float)(volDim[2] - 1);
    z = volMin[2] + fraction * (volMax[2] - volMin[2]);
    break;
  }
  // Update dragger position (without triggering callbacks)
  enableValueChangedCallbacks( FALSE );
  translation.setValue( x, y, z );
  enableValueChangedCallbacks( TRUE );
}
```

Finally, we need to handle a change to the dragger's position. It's basically just the inverse of what we did in the synchronize method. Compute the dragger 3D position as a "fraction" of the volume's 3D extent. Compute the new slice number as that fraction of the number of slices along the appropriate axis. Omitted here (but in the example program) is the range checking code to ensure that slice number stays in the valid range.

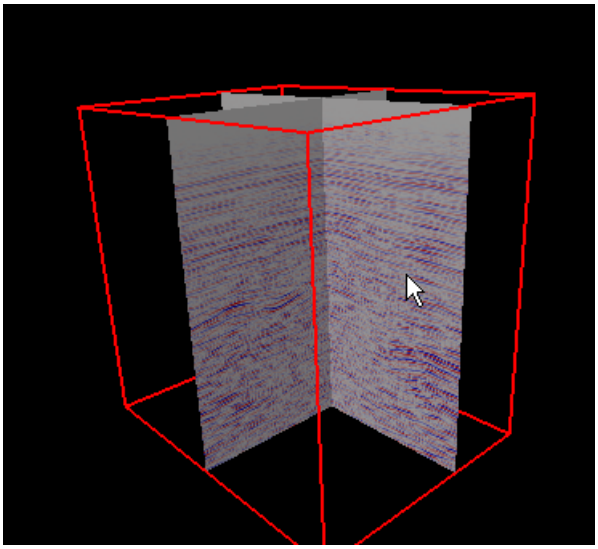
```
void
orthoSliceDragger::valueChangedCB( void *data, SoDragger *dragger)
{
  orthoSliceDragger *pThis = (orthoSliceDragger *)(dragger);

  // Get dragger info
  float x, y, z;
  (pThis->translation.getValue()).getValue(x,y,z);

  // Get slice info
  int sliceAxis  = pThis->m_orthoSlice->axis.getValue();

  // Get volume info
  SbVec3i32 volDim = pThis->m_volumeData->getDimension();
  SbBox3f volBox = pThis->m_volumeData->extent.getValue();
  SbVec3f volMin = volBox.getMin();
  SbVec3f volMax = volBox.getMax();

  // Compute dragger position as fraction of volume geometry
  float fraction;
  int   maxSlice;
  switch (sliceAxis) {
  case SoOrthoSlice::X :
    maxSlice = volDim[0] - 1;
    fraction = (x - volMin[0]) / (volMax[0] - volMin[0]);
    break;
  case SoOrthoSlice::Y :
    maxSlice = volDim[1] - 1;
    fraction = (y - volMin[1]) / (volMax[1] - volMin[1]);
    break;
  case SoOrthoSlice::Z :
    maxSlice = volDim[2] - 1;
```

```
    fraction = (z - volMin[1]) / (volMax[2] - volMin[2]);
    break;
  }
  // Convert fraction to slice number and set
  int newSliceNumber = (int)(fraction * maxSlice);
  pThis->m_orthoSlice->sliceNumber.setValue(newSliceNumber);
}
```

In the application program, we move all the code involved in creating and setting up a slice into the utility function “addSlice”. As mentioned earlier, most applications will create slices under an SoSwitch node to allow toggling the visibility of each slice separately (based on interacting with the application’s user interface). Note that the default value for the *whichChild* field is SO_SWITCH_NONE, meaning none of the children will be traversed, so we’ll change that. Also mentioned earlier, we need a *path* to initialize the ortho slice dragger, but it only needs to match part of the pick path when the event is handled, so we can use a “local” path. Initialize the path with the SoSwitch node (this will be the path’s “head” node), then add the Separator and finally the slice itself.

```
SoSwitch *addSlice( int axis )
{
  SoSwitch          *pSwitch  = new SoSwitch;
  SoSeparator       *pSep     = new SoSeparator;
  SoOrthoSlice      *pSlice   = new SoOrthoSlice();
  orthoSliceDragger *pDragger = new orthoSliceDragger();

  // Initial visibility should be ON (default for SoSwitch is OFF)
  pSwitch->whichChild = SO_SWITCH_ALL;

  // Initialize slice axis and position at middle of volume
  pSlice->axis = axis;
  pSlice->sliceNumber = spCurrentVolume->getDimension()[2] / 2;

  // Construct local scene graph
  pSwitch->addChild( pSep );
  pSep->addChild( pSlice );
  pSep->addChild( pDragger );

  // Initialize slice dragger with local path to slice
  SoPath *pPath = new SoPath(pSwitch);
  pPath->append( pSep );
  pPath->append( pSlice );
  pDragger->initialize( pPath, spCurrentVolume );
  return pSwitch;
}
```

The complete example program is provided with this document in “Slice_dragger.zip”. Switch to selection mode (press the ESC key), then click on any slice and drag it through the volume.



- End -