

Updated: March 10, 2015

Introduction:

The goal of this chapter is to provide a "quick start" and a "cook book" for software developers who are evaluating, or beginning to use, the Open Inventor™ toolkit in a medical or life science application. Open Inventor is a large, complicated and powerful toolkit, but also surprisingly easy to use. We will not explain every feature of Open Inventor here. We will try to "connect the dots" between typical medical application requirements and Open Inventor features, so you can find the relevant classes more quickly. The presentation is not intended to be strictly sequential. You should be able to go directly to whatever topic you need. If you are new to Open Inventor, we recommend reading the other chapters in the programming guide, *The Open Inventor Mentor*. Volume 1 covers the features of the core Open Inventor classes. We recommend reading at least the first three chapters for the basic concepts in Open Inventor. If you're really in a hurry, you could just read the *Getting Started Guide* (in the HowTos pull-down) for a quick introduction to Open Inventor concepts. For medical image and volume data you will probably want to use the *VolumeViz* extension, in addition to core Open Inventor. Depending on your requirements you might also need the *ImageViz* extension for 2D and 3D image processing, the *RemoteViz* extension for remote visualization and/or the *MeshVizXLM* extension for surface and volume mesh visualization. We will explain briefly the value of core Open Inventor, plus each extension, in the Overview section.

Each of the extensions has its own chapter in Volume 2 of *The Open Inventor Mentor* that explains the concepts, introduces the classes and shows code fragments for some common tasks. In this document we will summarize, but try to avoid duplicating the very detailed information in those chapters. When you see a link to a specific section in one of those chapters, you can follow it for more details. The value we hope to add with this chapter is a focus on medical related visualization requirements and the features that address those requirements. We will also show example code specific to medical tasks, show how Open Inventor classes address various requirements of medical applications and also discuss how to combine classes from extensions. For complete examples please see the Open Inventor SDK and the Open Inventor Medical Examples package.

Other Resources:

- The Open Inventor Developer Zone: <http://oivdoc.vsg3d.com>
A wealth of information including:
 - Release Notes, Fixed Bugs List and Compatibility Notes for current version.
 - The class reference manual, the "Open Inventor Mentor" and various "How To" documents.
- The Open Inventor Forum: <http://www.openinventor.net>
Registration (free) required.
 - Ask questions in the main forum
 - Take advantage of code examples and utility classes in the Resources section.
- The examples provided with the Open Inventor SDK.
- The Open Inventor Medical Examples Package .

Contents

Overview	3
Concepts.....	7
Slice Visualization	16
Display a Slice	16
A 2D view of slices:	18
The other 2D slices:	19
A 3D view of slices:	21
Arbitrary planar slice:	22
Non-planar slice:	23
Thick slice (slab):	23
Isosurface Visualization	25
Display an Isosurface	25
Multiple isosurfaces:	26
3D Volume Visualization	28
Direct Volume Rendering	28
More performance/quality settings	29
Volume Cropping	30
Miscellaneous Bits.....	31
Display text on screen (e.g. DICOM info)	31
Show scene orientation (compass).....	32
Interaction.....	35
Using the Mouse Wheel	35
Querying voxels by cursor	37

Overview

Open Inventor

The Open Inventor core library provides the general tools for building a 3D visualization application. This includes the “scene graph” for organizing your data, property and rendering objects; viewers for integrating a 3D window in your user interface; general purpose geometry and image rendering nodes; support for picking (selection) and probing volumes; 3D widgets (dragers) for interacting with the 3D scene; as well as transparency, shadows, anti-aliasing, stereo, immersive VR and much more. For interaction, Open Inventor supports standard mouse, button and key events plus touch and gesture events, some 3D input devices (e.g. SpaceMouse) and tracked input for VR. The Open Inventor scene graph is a general mechanism for organizing data, property and rendering objects that define the 3D (and 2D) scene for your application. This is much more convenient for the application than directly using an immediate-mode API like OpenGL. Open Inventor also allows sequential operations on data sets to be linked into a data “pipeline”, for example to do image processing.

You may be already familiar with another API, for example VTK, that uses a pipeline architecture for everything. By comparison, a lot of repetitious calls to connect data, filter, property and rendering objects are unnecessary in Open Inventor because there are, in effect, implicit connections defined by the scene graph. The hierarchical organization of the scene graph also makes it easy to group rendering nodes that share properties or, conversely, to define rendering nodes that use the same data source but have different properties. Using the scene graph the application can organize data, property and rendering objects in a logical way using groups and hierarchy. The application can easily control the visibility of objects, or groups of objects, using SoSwitch nodes. Open Inventor provides a powerful interactive tool called IvTune that can be used to inspect and modify the scene graph. Using the scene graph, a data node may be used as a container for the actual data, but for images and volumes it will more often be a sort of “place-holder” that positions the data in the scene graph but refers to some other storage. For example, volume data managed by VolumeViz LDM or image data computed in an ImageViz pipeline.

VolumeViz

The VolumeViz extension provides tools to visualize image and volume data using slices, isosurfaces and state-of-the-art GPU ray-casting direct volume rendering. A very general Multiplanar Reformation (MPR) is supported. VolumeViz can display axis-aligned slices, arbitrary planar slices and application defined slices (e.g. curved). Additionally you can interact with your volume data, transform and filter volume data, clip and sculpt a volume and co-render multiple volumes. VolumeViz automatically takes advantage of the graphics hardware features such as programmable shaders and compressed textures to maximize performance and image quality. VolumeViz automatically manages both CPU and GPU memory, allowing large volumes to be rendered efficiently. Because VolumeViz is integrated with Open Inventor and the other extensions, you can mix volume rendering with conventional 2D or 3D geometry. VolumeViz, like Open Inventor, can be extended to meet the custom requirements of advanced applications. In addition to the Open Inventor custom node mechanism, VolumeViz provides a framework that allows applications to easily implement custom shader functions that execute on the GPU. This framework includes a library of pre-built GLSL shader functions that allows applications to extend or replace specific steps in the rendering process.

ImageViz

The ImageViz extension provides a wide range of high performance filters for 2D and 3D image processing, segmentation, registration and feature analysis (for example, to compute statistics about objects in the image or volume). ImageViz filters can be connected to make a data pipeline so that changing the input data or changing parameters of a filter automatically causes the necessary portions of the pipeline to re-execute. An ImageViz filter can load images and volumes directly from disk or memory or import images and volumes loaded using VolumeViz.

Modified images and volumes can then be fed back into Open Inventor or VolumeViz for rendering and interaction.

MeshVizXLM

The MeshVizXLM extension provides tools to manage, extract and visualize 1D, 2D or 3D mesh data, for example surfaces or tetrahedral cell models reconstructed from medical data. MeshVizXLM can extract new meshes from existing ones, for example extracting a surface mesh representing the skin of a volume mesh or a surface mesh representing an isosurface of the data. MeshVizXLM can take a VolumeViz volume as input (a volume can be considered to be a completely regular hexahedron mesh). This allows surface geometry to be extracted from a segmented data set or isosurface geometry to be extracted based on specified values.

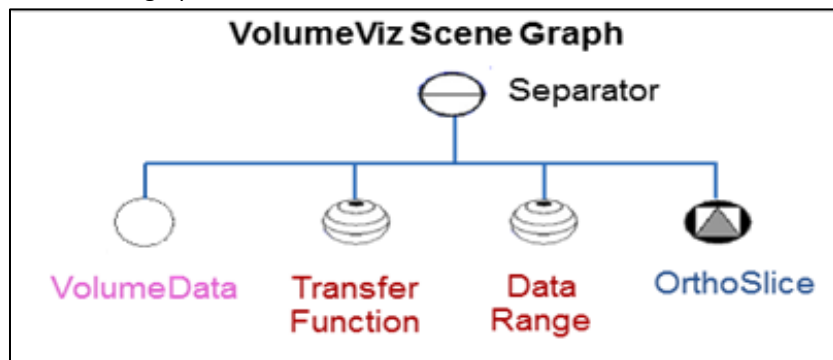
RemoteViz

The RemoteViz extension enables software developers to integrate remote 3D interaction and visualization into web-based and mobile applications. Web-based applications using Open Inventor can display on any device with an HTML5-enabled web browser (tablet, phone, laptop, or workstation). The actual 3D rendering is done on a remote server machine. The same Open Inventor scene graph code can be re-used in both desk-top and web-based versions of the application.

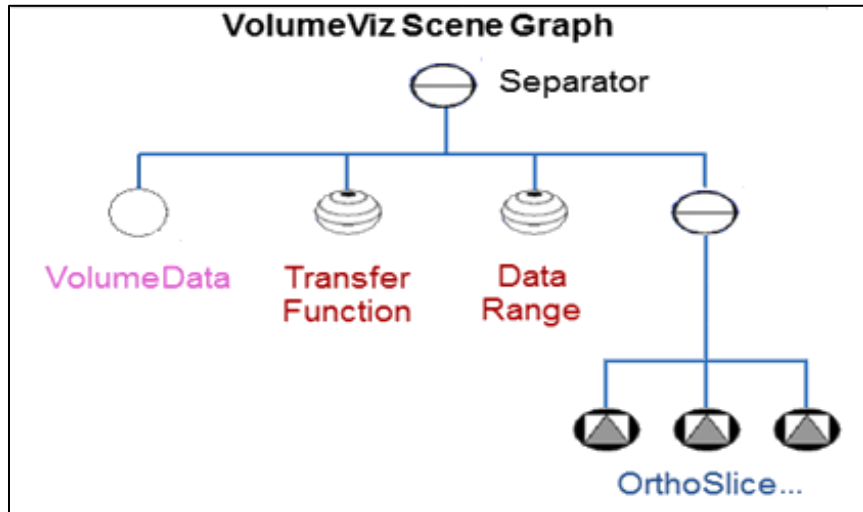
VolumeViz Features:

Just like core Open Inventor, VolumeViz has data nodes, property nodes and geometry (rendering) nodes. What you already know about using nodes, fields, actions and so on also applies to VolumeViz. A basic VolumeViz scene graph is shown here. Like any Open Inventor scene graph, there is a grouping node (*SoSeparator*) and its child nodes include a data node (*SoVolumeData*), several property nodes (e.g. *SoTransferFunction*) and a rendering node (*SoOrthoSlice*). The values set by the most recently traversed property node(s) apply to the rendering node. Like any Open Inventor scene graph we could add additional rendering nodes, for example slices, that render from the same data node but in different positions. All the slices could share the single transfer function, as shown here, or each slice could have its own transfer function. The application has tremendous flexibility to organize the scene graph in a logical way for its own requirements.

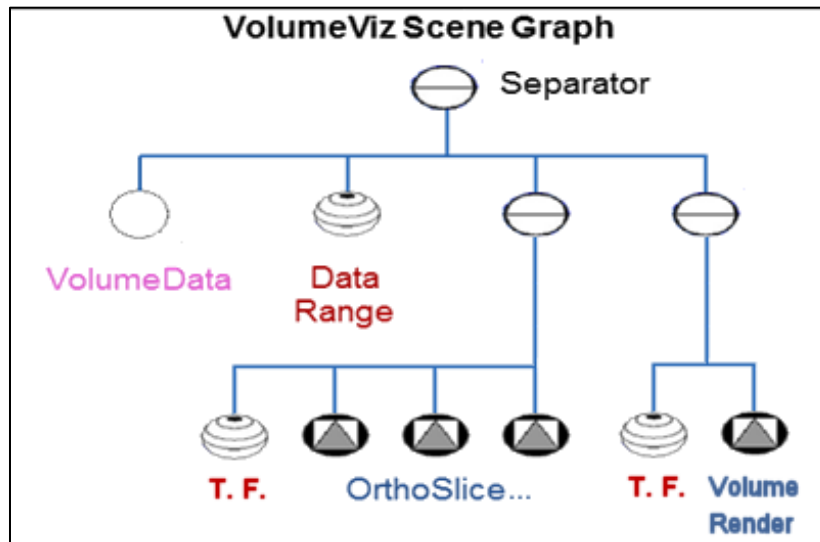
A very simple VolumeViz scene graph:



A VolumeViz scene graph with multiple slices sharing the same transfer function:



A VolumeViz scene graph with separate transfer functions for slice and volume rendering:



A summary of important VolumeViz features/classes you may find useful (there are additional features):

- **Data:**
 - *SoVolumeData*: Holds or refers to the image or volume data to be rendered.
 - VolumeViz supports non-uniform voxel spacing along one or more axes.
 - VolumeViz supports non-linear volume warping (with some limitations).
- **Properties:**
 - *SoROI*: Specifies the region of interest (sub-volume) to be rendered.
 - *SoDataRange*: Specifies the range of voxel values to be mapped into the color table.
 - *SoMaterial*: Specifies the material properties common to all voxels.
 - *SoTransferFunction*: Specifies the color and opacity lookup table.
 - *SoVolumeShader*: Specifies custom GLSL shaders for slice rendering.
 - *SoVolumeRenderingQuality*: Specifies options (including shaders) for volume rendering.

- **Rendering:**
 - *SoOrthoSlice*: Renders an axis aligned slice (axial, sagittal or coronal)
 - *SoObliqueSlice*: Renders a slice defined by an arbitrary plane in 3D.
 - *SoVolumeFaceSet*: Renders custom slice geometry, e.g. a curved slice.
 - *SoVolumelsosurface*: Renders one or more isosurfaces defined by specific data values.
 - *SoVolumeRender*: Renders the region of interest using direct volume rendering.
- **Clipping:**
 - *SoROI*: Crops volume against a “box” in voxel coordinates (including “cut away” box).
 - *SoClipPlane*: Clips slice and volume rendering against an arbitrary plane in 3D.
 - *SoVolumeClippingGroup*: Clips against a closed shape defined by polygonal geometry.
 - *SoCSGShape*: Add, subtract and intersect polygonal shapes (to refine clipping geometry).
 - *SoVolumeMask*: Clips against an arbitrary region defined by a mask volume.
- **Interactivity**
 - Picking works for image and volume data similar to other Open Inventor geometry. You can get detailed information, like the voxel value, from a pick (see for example *SoOrthoSliceDetail*).
 - VolumeViz provides specialized draggers to allow users to directly interact with volume rendering scenes. For example, *SoOrthoSliceDragger* allows dragging a slice through the volume.
 - Modifying a node automatically causes the scene graph to be re-rendered. For example changing the color map by modifying *SoTransferFunction*, or modifying parameters of a filter.
 - Image and volume data can be modified interactively using the methods in *SoVolumeData* (see *startEditing*, *finishEditing*, etc). Image/volume editing supports undo and redo.
 - For large volumes VolumeViz provides multiple options for temporarily reducing the rendering complexity while interacting with the scene. See for example the *SoInteractiveComplexity* node.
- **Custom Shaders**
 - Custom transfer functions, masking, rendering effects and custom blending are just a few of the many effects that can be implemented using an *SoVolumeShader* node and GLSL shader functions. (GLSL is the standard OpenGL shader language.)
 - VolumeViz provides a framework of prebuilt shader functions for commonly used calculations and effects. This allows applications to extend or replace stages in the rendering pipeline while still taking advantage of other VolumeViz features.
- **Multiple Data Sets**

Multiple *SoVolumeData* nodes can be inserted in the same scene graph.

 - If the volumes are independent and rendered separately, use an *SoVolumeGroup* node to manage and correctly render intersecting regions in the same scene.
 - If the volumes are different data sets for the same set of voxels, use an *SoVolumeShader* node to specify a fragment shader that combines (or chooses one of) the data values or colors.
 - If the volumes are different time steps for the same set of voxels, use an *SoSwitch* node to select each data set in sequence for rendering.

Concepts

In this section we explain how key concepts and common tasks in medical visualization are supported by Open Inventor.

Images & Volumes:

In general, image data and volume data are the same thing. Conceptually an image has two dimensions and a volume has three dimensions, but an image can also be handled as a volume whose Z dimension is exactly one. To avoid having to say “image or volume” everywhere in this document, we will generally say “volume” when talking about VolumeViz and “image” when talking about ImageViz. The discussion applies to both unless we highlight a specific difference that’s important to the current topic. VolumeViz and ImageViz have separate classes for encapsulating data, but adapter classes allow using the same data in either or both.

Volume Properties:

First we need to understand how Open Inventor describes a volume and how that relates to commonly used medical terminology. The VolumeViz extension only supports volumes with a single scalar or RGBA value per voxel. However we can use VolumeViz to render a multi-channel data set using multiple volume data nodes, one for each channel, and a data combining shader function. See the multi-channel microscopy example in the SDK. The data values in a volume can be interpreted several different ways including:

- Scalar values
 - Specify a value, e.g. intensity, for each voxel.
Can be byte, short, int or float values (signed or unsigned).
- Label values
 - Integer values that specify an “id” for each voxel.
A label volume is usually the result of doing segmentation on the original volume.
- Mask values
 - Boolean values that specify “true” or “false” for each voxel.
A mask volume can be used to clip voxels, to assign different transfer functions, etc.
- RGBA values
 - Unsigned int (4 byte) values that specify the RGBA value of each voxel.
Color/opacity lookup (transfer function) is not done for RGBA volumes.

The essential properties that define an image or a volume in Open Inventor are:

- Dimensions: The number of voxels on each axis.
- Data type : byte/unsigned byte, short/unsigned short, int/unsigned int, float or RGBA.
- Extent : The geometric extent of the volume in 3D space
(defined by two corners: minX, minY, minZ and maxX, maxY, maxZ).

If possible, the image or volume reader will get this information from the data file(s). The dimensions and data type are fixed, but the application may change the volume extent to stretch or compress the voxels. Voxels do not need to be cubes, i.e. the X, Y or Z voxel size may be different from the other axes. By default VolumeViz assumes that the voxel size is uniform along each axis, but non-uniform voxel spacing is also supported (advanced topic).

NOTE: Voxel size.

The voxel size (or spacing) is *indirectly* specified in both VolumeViz and ImageViz using the 3D ‘extent’ property. The “units” in 3D space are whatever the application wants them to be, for example millimeters. The voxel size for each axis is defined by the volume dimensions and extent. For example, if

the volume dimension in X is 512 voxels and the voxel size in X is 0.408 mm, then the DICOM volume reader will set the volume extent in X to (approximately) 209 mm. To compute the voxel size in your application, simply divide the extent by the dimension. Conversely, to compute the volume extent in your application, simply multiply the voxel size by volume dimension.

Volume Data:

In this sub-section we're talking mainly about the VolumeViz extension. Use an *SoVolumeData* node to put a reference to the volume data into the scene graph. Loading volume data is discussed in section 1.2 of the Inventor Mentor's VolumeViz chapter (<http://oivdoc.vsg3d.com/content/123-loading-file>). We will briefly summarize here. VolumeViz accesses data through an interface class called *SoVolumeReader*. So effectively *SoVolumeReader* objects are plug-ins. Most volume readers load data from a specific file format, but the data source could be memory, a network connection, etc. VolumeViz provides volume readers for some common file formats, for example DICOM. Applications can create new readers by deriving a class from *SoVolumeReader*. For convenience, if a filename is specified and VolumeViz recognizes the file extension string, an instance of the appropriate volume reader is automatically created. In this case the application can query the volume reader from the *SoVolumeData* node, for example to get DICOM header info. If the data file is a supported format, but has a non-standard file extension, the application must create an instance of the appropriate volume reader and set it in the *SoVolumeData* node.

The location of the data can be given by:

- A single string, which is:
 - The name of a data file containing the volume (or image), or
 - The name of a text file containing a list of file names to be loaded as one volume, or
 - The name of a directory containing DICOM files to be loaded as one volume.
- A list of strings
 - For DICOM or stack of images, the names of data files to be loaded as one volume.
- A volume reader object.
- A block of memory (application must specify data type, dimensions and extent).

See *SoVolumeData* or *SoVolumeReader* for the current list of supported file formats. The volume data readers you will most likely use are:

- DICOM
 - See *SoVRDicomFileReader*.
VolumeViz will automatically load files with a ".dc3", ".dic", ".dcm", or ".dicom" extension. A DICOM file may contain one slice, multiple slices or an entire volume. The order and position of each slice is defined by the DICOM file header. The volume extent is automatically set in millimeters (mm) based on the DICOM file header information.
 - The reader's *getDicomData()* method returns an *SoVRDicomData* object that allows the application to query information from the DICOM file header using group/element numbers. It also provides some convenience functions, e.g. *getSliceThicknessMM()*.
 - NOTE: The volume extent that the DICOM reader automatically computes is always centered at 0,0,0 in 3D world coordinates. For example, the min and max X values in the extent box are exactly $-\text{width}/2$ and $+\text{width}/2$, which would be -104.5 mm and +104.5 mm if the volume width is 209 mm. The DICOM volume origin (*ImagePositionPatient*) is *not* considered when setting the volume extent. This value can be queried using the *SoVRDicomData* object. The application can set the volume extent using the *SoVolumeData* node's 'extent' field.

- Stack of images
 - See *SoVRRasterStackReader*.
This class allows you to load a collection of image files (one per slice) as a volume. The image files can be any common format (JPEG, PNG, etc). The application must create (or there must exist) a meta-data file that contains keywords defining the properties of the volume followed by a list of filenames. Allows loading one channel, e.g. red, from an RGB image.
 - The order of the slices (images) is defined by the order of the filename list.
 - The volume extent in 3D can be specified in the meta-data file or set by the application.
- Image data pipeline
 - See *SoVRImageDataReader*.
This class allows you to load a volume from the output of an ImageViz image processing pipeline. For example to load a volume and apply smoothing or de-noising filters before rendering. The original input volume could be loaded into the pipeline using ImageViz (*SoFileDataAdapter*) or VolumeViz (*SoVolumeReaderAdapter*).
- Raw and miscellaneous data
 - See *SoVRGenericFileReader*.
This class allows you to load a volume from a data file as long as you know the data type, the dimensions of the volume and the offset to the beginning of the data bytes (i.e. the length of the header, if any). This allows loading “raw” files with no (or separate) header and also allows loading formats like NRRD where the data values are at the end of the file.
- Volume in memory
 - Set the ‘data’ field of the *SoVolumeData* node directly. VolumeViz will create an instance of the *SoVRMemoryReader* class internally to manage the data.

Image Data:

As mentioned previously, image data is a subset of volume data. The VolumeViz extension can be used to load and display images as well as volumes. This is the most direct approach if the initial goal is to display the image (ImageViz does not provide rendering). The ImageViz extension can also be used to load both image and volume data. This is the most direct approach if the initial goal is to apply algorithms to the image, for example smoothing and de-noising filters. You will use one of the sub-classes of *SoImageDataAdapter* to load data using ImageViz. For example, *SoFileDataAdapter* to load data from a file or *SoVolumeReaderAdapter* to access data already loaded using VolumeViz. Similar to *SoVolumeData*, the image data adapter allows you to query the voxel dimensions, voxel characteristics, actual voxel values and the 3D extent of the image. Unlike *SoVolumeData*, the image data adapter is not itself an Open Inventor node and cannot be added directly into the scene graph. If rendering is required, the output of the ImageViz processing pipeline can be linked to an *SoVolumeData* node in the scene graph using the *SoVRImageDataReader* class.

Coordinate Systems

In a medical imaging application there are typically three related coordinate systems:

- 3D world coordinates:
Also called XYZ coordinates. This is the Cartesian coordinate system in which rendering objects (volumes, geometry), cameras, lights, etc. are positioned. In Open Inventor this is a right-handed coordinate system (same as OpenGL). The default camera creates a view looking down the Z axis, i.e. +Z is toward the user, +X is to the right and +Y is up. The view can be changed by setting the camera node’s ‘position’ and ‘orientation’ fields. The position and orientation of the volume (or any other rendering object) can also

be changed by adding a transformation node (e.g. *SoTransform*) to the scene graph.

- **Voxel coordinates:**
Also called “IJK” coordinates. Each voxel’s position in the volume can be described by an IJK coordinate. In VolumeViz this is also a right-handed coordinate system, corresponding to the positive XYZ axes, i.e. in the default camera’s view K is toward the user, I is to the right and J is up. When loading data, VolumeViz usually assumes that voxels are loaded with I varying fastest, then J, then K. Thus the K axis is the “slice number”, the first voxel is the lower left back corner of the volume and slice number 0 is the furthest from the user. More complex data formats, like DICOM, may specify a different loading order. Normally in VolumeViz, the voxel “I” axis corresponds to the 3D “+X” axis, “J” corresponds to “+Y” and “K” corresponds to “+Z”. You can convert between IJK and XYZ coordinates based on the volume dimensions and extent or use the conversion methods provided in the *SoVolumeData* class, e.g. *voxelToXYZ()*.
- **Anatomical coordinates:**
The three standard viewing planes are Axial, Coronal and Sagittal. The Axial plane divides the body into Superior (head) and Inferior (foot) sections. The Coronal plane is a vertical plane dividing the body into Anterior (front) and Posterior (back) sections. The Sagittal plane divides the body longitudinally into Right and Left sections. The orientation of the patient in a medical data set is often specified using three letters to identify the primary axes, where each letter is one of the section names. For example DICOM volumes typically have LPS (Left, Posterior, Superior) orientation. In this case VolumeViz considers that the direction:
 - Toward the left side of the body (Left) is the I / +X / Sagittal axis,
 - Toward the back of the body (Posterior) is the J / +Y / Coronal axis, and
 - Toward the head (Superior) is the K / +Z / Axial axis.

In other words, the default camera view is looking at the top of the head with the body “face down”.

Memory Management

VolumeViz is responsible for managing the volume data within a specified maximum amount of CPU memory and GPU memory. These limits ensure that the application can reserve CPU and GPU memory for other needs. By default the maximum CPU memory that will be used is 50% of the total CPU memory, up to a maximum of 70% of the current free (available) memory. See the *SoCpuDevice* class to query available CPU memory. By default the maximum GPU memory that will be used is 75% of the total GPU memory. See the *SoGLDevice* class to query available GPU memory. The application can set the maximum total memory allowed for all loaded volumes (see the *SoLDMGlobalResourceParameters* class) or set the maximum memory allowed for each volume (see the ‘*ldmResourceParameters*’ field in *SoVolumeData*).

It is possible for the volume data to be much larger than the maximum available CPU or GPU memory. It is also possible for an image to be larger than the maximum GPU texture size. In both cases, some toolkits will automatically subsample the data and then display a lower resolution volume or image. VolumeViz avoids this problem using an internal data manager called “LDM” that manages the data as a multi-resolution hierarchy of “tiles”, where a tile is a fixed size region of the volume (tile size can be specified by the application). The advantage of this approach is that VolumeViz can intelligently load some (the most important) tiles at full resolution and other tiles at a lower resolution. This is managed dynamically so that “zooming in” on part of the volume automatically loads higher resolution data, somewhat like the well-known Google Earth application. One consequence of this approach is that using a small tile size (relative to the volume dimensions) creates a large number of tiles and a lot of overhead for managing them.

TIP: Tile size.

The default tile size is only 64x64x64. We recommend setting the tile size to a power-of-2 approximately equal to the largest volume dimension, but no larger than 512. For example, tile size 512 works well with typical CT data sets where each image (slice) in the stack is 512x512 pixels. Set the tile size in the volume data node using the 'ldmResourceParameters' field (see also *SoLDMResourceParameters*). Setting the tile size should be done immediately after setting the filename or reader.

Performance vs Image Quality

VolumeViz can temporarily reduce the image quality in various (configurable) ways to maintain performance while the user is interacting with the scene. The most common interaction is navigating around the scene (panning, zooming and rotating). Effectively this means changing the fields of the camera node. In fact interaction means changing the field values of any node in the scene graph. Reducing image quality is usually not necessary for slice rendering, but can be critical for the user to have an interactive experience with your application when using high quality volume rendering. There are several ways to increase performance temporarily including:

- Reducing the number of samples along each ray.
- Reducing the number of rays cast into the volume.
- Turning off expensive computations, e.g. cubic interpolation.

NOTE: Interactive settings.

VolumeViz will automatically apply these changes when the user begins interacting, but it is the application's responsibility to decide how much to reduce the number of samples and/or number of rays. VolumeViz does not currently adjust these values automatically to maintain a constant frame rate.

See the 'lowResMode' field in *SoVolumeRender* and the *SoInteractiveComplexity* node. Also see the [Performance/Quality Settings](#) section later in this chapter for a brief discussion of these options.

Data Range and Transfer Function

In VolumeViz the mapping of a voxel's data value to a gray scale or color/opacity value is controlled by two nodes: *SoDataRange* and *SoTransferFunction*. This is a little different than toolkits where data range is part of the transfer function. In VolumeViz, the *SoTransferFunction* node specifies a gray/opacity or color/opacity lookup table and the *SoDataRange* node specifies the range (minimum and maximum) of data values that will be mapped onto the lookup table. Values less than the minimum range value are mapped to the first value in the lookup table and values greater than the maximum range value are mapped to the last value in the lookup table.

By default VolumeViz maps the entire range of the voxel data type onto the lookup table. For example, a typical DICOM volume contains signed short values. The default data range for signed short is -32768 to +32767, which is much larger than necessary (or useful). It is possible to query the actual smallest and largest data values in the volume using *SoVolumeData* method `getMinMax()`. However if you know the typical range of values for your data it's better to use those values directly. First, because it takes extra time to search the volume for the actual min/max values (unless the data format stores these values explicitly in its header, like VolumeViz's native LDM file format). And second because the data may contain actual min/max values that are outside the typical range. For example, to assign an intensity ramp to the typical range for CT values calibrated in Hounsfield units, add these nodes to the scene graph:

```
// Set range of data values to visualize.
SoDataRange* volRange = new SoDataRange();
volRange->min = -1000;
volRange->max = 3000;
```

```

volSep->addChild( volRange );

// Load opaque intensity ramp
SoTransferFunction* volTF = new SoTransferFunction();
volTF->predefColorMap = SoTransferFunction::INTENSITY;
volSep->addChild( volTF );

```

The default color lookup table is called GRAY and contains a linear opacity ramp with constant gray values (all white). This can be useful for volume rendering. For slice rendering, it's more useful to use the predefined INTENSITY map, which is a linear gray scale ramp with constant opacity values (all opaque). Needless to say, the "rainbow" color map (STANDARD) should not be used for medical imaging. :) The GLOW map contains an opaque "heated-object" color scale that can be useful. The *SoTransferFunction* method `loadColormap()` can read Amira™ and Avizo™ format color lookup tables. Some useful volume rendering color tables, for example "volrenRed.col.am", can be found in the VolumeViz data directory in the SDK. Finally, you can create your own lookup table using the 'colorMapType' and 'colorMap' fields (see the *SoTransferFunction* page for an example).

Custom Transfer Functions:

It is also possible to completely replace the default color/opacity lookup with essentially any algorithm, using a custom GLSL shader function. The VolumeViz shader framework allows an application to override specific functions in the rendering pipeline while still using all the other rendering features. In this case the application would provide a custom version of the `VVizComputeFragmentColor()` GLSL function using an *SoVolumeShader* node for slice rendering and an *SoVolumeRenderingQuality* node for volume rendering. This can be used, for example, to implement a "2D" transfer function using both the voxel's value and gradient magnitude to assign color and opacity. Note that both voxel value and gradient are available to the shader function, so it is not necessary to pre-compute gradients. You can find more details about using custom shaders with VolumeViz in section 1.8 of the VolumeViz chapter of the Open Inventor Mentor (<http://oivdoc.vsg3d.com/content/18-shaders>).

Window and Level:

For gray scale medical imaging, it is common to use the concept of "window center" and "window width" (sometimes just called "window" and "level") to adjust the brightness and contrast of the image. In VolumeViz this is just a different way of specifying the data range that should be mapped onto the lookup table. In the previous example the data range is -1000 to 3000, so effectively the window center is 1000 and the window width is 4000. You can convert window center/width values to *SoDataRange* min and max values like this:

```

volRange->min = windowCenter - (windowWidth / 2);
volRange->max = windowCenter + (windowWidth / 2);

```

Piece-wise linear transfer functions:

For color mapped medical imaging, it is common to use a "piece-wise linear" function to assign colors to different ranges of data values. For example, an application might want to assign one color to the range of values representing "skin" (perhaps 500 to 1000) and a different color to the range of values representing bone (perhaps 1150 and over). Using VolumeViz, you will need some helper code to convert the linear function to a lookup table, then set the data range in the *SoDataRange* node and the lookup table in the *SoTransferFunction* node. Alternatively you can implement any custom transfer function using GLSL shader functions (see the *SoVolumeShader* class and the `VVizGetData()` and `VVizComputeFragmentColor()` GLSL functions).

Gradient Magnitude Modulation:

VolumeViz has several options to automatically adjust illumination and opacity based on the gradient magnitude. One use is for noisy data that contains gradient vectors with small magnitudes and random directions that scatter light randomly rather than showing material changes in the volume. The *SoVolumeRenderingQuality* node provides several options to either ignore small gradients (see the 'gradientThreshold' field) or adjust lighting based on the gradient magnitude (see the 'surfaceScalarExponent' field).

The 'boundaryOpacity' option modulates opacity based on the gradient magnitude. This is useful to highlight the shape of a lower density material surrounding a higher density material, for example soft tissues surrounding bone in medical data. You can find more details about all the volume rendering effects in section 1.5.4 of the VolumeViz chapter of the Open Inventor Mentor (<http://oivdoc.vsg3d.com/content/154-volume-rendering-effects>).

Voxel Material:

An *SoMaterial* node can be used to specify the material properties for the voxels in the volume. This feature has several different uses and applies to different rendering objects in slightly different ways. For isosurface rendering, the material node directly specifies the material properties of each isosurface exactly like it does for polygonal geometry. For slice and volume rendering, the *SoMaterial* specifies the "base" material of all voxels in the volume. The color and opacity assigned to the voxel by the transfer function are combined with the base material to compute the final color and opacity. The default material is an opaque "light gray" (RGB values 0.8, 0.8, 0.8). This material is useful when lighting is enabled because the light source will increase the brightness of the voxels. To get exactly the gray or color value assigned by the lookup table, set the material node's 'diffuseColor' field to full white (RGB value 1, 1, 1). To give the slice or volume a blue-green tint, for example, set the red component of the 'diffuseColor' field to a smaller value, e.g. RGB 0.5, 1, 1. When lighting is not enabled, only the 'diffuseColor' and 'transparency' fields are used. When lighting is enabled, the material node's other fields (except 'emissiveColor') are taken into account. The default specular color is black [0, 0, 0], so specular highlights are not rendered. To render specular highlights it's better to use the default diffuseColor [0.8, 0.8, 0.8] and set the specularColor to a small white value, e.g. [0.2, 0.2, 0.2].

For volume rendering, the 'transparency' field can be used to apply a global scale factor to all opacity values (effectively shifting the opacity curve up or down). Note this is a "transparency" value, not an "alpha" value. The scale factor applied to the alpha values is "1 – transparency". For example, a transparency value of 0.95 effectively means to apply a scale factor of 0.05 to all opacity values. This works whether or not lighting is enabled.

Lighting:

For slice and isosurface rendering, lighting is enabled by default, as it is for polygonal rendering. For volume rendering, lighting is disabled by default and must be enabled using an *SoVolumeRenderingQuality* node.

VolumeViz supports two lighting algorithms for volume rendering. First is a classical gradient based lighting algorithm enabled by the *SoVolumeRenderingQuality* 'lighting' field. In this case a gradient vector is computed for each sample along the ray and used as the "surface normal vector" in the classical lighting equation. You can select the algorithm used to compute gradient vectors using the 'gradientQuality' field and control small gradient artifacts using the 'gradientThreshold' or 'surfaceScalarExponent' fields. The second algorithm computes lighting based on the final image depth buffer instead of using the data gradients. This option is enabled by the 'deferredLighting' field. Deferred lighting is much faster than using data gradients, does not have problems with small/random gradients and is well suited for data that has well defined surfaces (like bones).

Light sources are specified using one or more Open Inventor *SoDirectionalLight* nodes. Light color (but not intensity) is taken into account. If you use one of the Open Inventor viewer classes in your application, the viewer

will automatically create a “headlight”. This is a directional light that always shines in the direction the camera is looking (it automatically rotates with the camera). VolumeViz deferred lighting supports up to 8 light sources. Gradient lighting only supports a single light source.

Lighting effects:

VolumeViz supports shadow casting for all rendering nodes and this feature is integrated with Open Inventor geometry. Geometry can cast shadows on voxels and, conversely, opaque voxels can cast shadows on both geometry and other voxels. See the *SoShadowGroup* node. Shadows require lighting to be enabled.

For volume rendering, VolumeViz also supports a global lighting effect called “ambient occlusion” that takes into account the attenuation of light due to neighboring (relatively opaque) voxels. Enable this option using an *SoVolumeRenderingQuality* node. The ambient occlusion effect is independent of whether lighting is enabled.

Transparency:

VolumeViz rendering is fully integrated with the advanced transparency algorithms available in Open Inventor. If the application is using opaque geometry and opaque slices, then it is not necessary to explicitly set the transparency algorithm (or at least it is sufficient to set one of the basic algorithms like `DELAYED_BLEND`). In this case VolumeViz will automatically handle transparency for volume rendering.

Note that the Open Inventor algorithm `SORTED_PIXELS_BLEND` can handle complex combinations of transparent objects that are not correctly rendered by some toolkits. For example, Open Inventor can correctly render multiple intersecting transparent slices, as well as transparent geometry embedded inside a volume rendering. The application must explicitly set the transparency algorithm using the `setTransparencyType()` method in the render area class, viewer class or the *SoGLRenderAction* class.

Special rendering modes:

VolumeViz supports some special rendering modes, for example “voxelized” rendering, sometimes called “lego rendering” or “sugar cube” rendering. In this mode each voxel is rendered as a box, optionally with edges (outline) drawn. Voxelized rendering is useful for precise editing of the volume and for displaying label volumes. This option is enabled using the ‘`voxelizedRendering`’ field of *SoVolumeRenderingQuality*.

GPU Data:

VolumeViz caches the volume’s actual data values in CPU memory then downloads data to the GPU for rendering. If the volume contains RGBA (4 byte) values, these values are sent to the GPU “as is”. For scalar values it’s a little more complicated. GPU memory is typically much smaller than CPU memory so, by default, VolumeViz only loads 8-bit values on the GPU. That means that for 8-bit data the actual data values are loaded on the GPU, but for larger data types the values are scaled before downloading. There is some loss of precision, but the default color lookup tables only have 256 values, so there are only 256 different gray or color values available for rendering. When you query the value of a voxel, VolumeViz always returns the value from CPU memory, so the size of the values on the GPU doesn’t matter. If you have 12 or 16 bit data and need the actual data values loaded on the GPU, set the `texturePrecision` field on the *SoVolumeData* node like this:

```
volData->texturePrecision = 16;
```

To create a color lookup table with more than 256 entries, just load all the values into the ‘`actualColorMap`’ field of the *SoTransferFunction* node. The limit is the maximum OpenGL texture dimension. This value depends on the GPU, but usually at least 4096 for current hardware. If necessary you can work around this by writing a custom

color lookup shader (see *SoVolumeShader*).

NOTE: Previously we said that *SoDataRange* specifies the range of data values that will be mapped onto the color lookup table. That's true, but not the whole story. What *SoDataRange* really does is specify the range of data values that will be scaled to fit in the GPU voxel data type. This is particularly important to understand for label volumes.

Label Volumes:

In a label volume, each voxel value is an integer label (id) identifying the material, object, etc. that the voxel belongs to. An important difference is that a data volume is conceptually a set of discrete samples taken from a continuous scalar field. So we know the exact value at the center of each voxel and we can interpolate between those values to get the value at any position between the voxel centers. By contrast, in a label volume we normally consider each voxel to belong completely to one material, so the value is constant until we cross the boundary into the next voxel. Therefore we do not want to interpolate the label values. When rendering a label volume with volume rendering, set the *SoVolumeRender* node's 'interpolation' field to NEAREST and leave the *SoVolumeRenderingQuality* node's 'preintegrated' field set to false. When rendering isosurfaces of a label volume, set the *SoVolumeRenderingQuality* node's 'segmentedInterpolation' field to true.

It is also important to set the data range, texture precision and color map size carefully. In general the number of values in the data range and the number of entries in the color map should be exactly equal and a "power of 2" (8, 16, 32, etc.). For example, if you have 6 label values, you might logically set the data range to 0..5 (min = 0 and max = 5) and create a label color map with 6 color values. That will work most of the time, but in some cases round-off in the mapping arithmetic can cause a data value to map to the wrong color. In the previous example, with 6 label values, we recommend setting the data range to 0..7 (i.e. 8 values) and creating a color map with 8 color values. The additional values (6 and 7) do not appear in the volume so the associated color values do not matter.

Writing Data:

In some cases you may need to write image or volume data back to disk after filtering, segmenting, etc. Image data (or a single slice of a volume) can be saved in any of the common formats (PNG, TIFF, etc.) using one of the sub-classes of *SoRasterImageRW*. See the example code in the Reference Manual entry for this class. The output of an ImageViz filter can be conveniently saved using the *saveToFile()* method of *SbImageDataAdapterHelper*. Volume data can be saved in VolumeViz's native LDM format (.ldm) using the *SoLDMWriter* class. To save in other formats it is necessary to extract the data from the volume using the *SoLDMDataAccess* class and write the data using other tools. For example, toolkits like GDCM or DCMTK can be used to save data in DICOM format.

Slice Visualization

In this chapter the goal is to load a volume data set, for example a DICOM volume, and display one or more slices from that volume. Remember that a “volume” could be a single image. And any image can be considered to be a volume with the third dimension equal to one. We will start with a traditional 2D slice display and then discuss displaying slices in 3D.

Display a Slice

The code to create a basic VolumeViz scene graph for slice rendering is shown below. Note that this is not the “minimal” scene graph to display a slice. That could be just a few nodes. The code shown here is a recommended practice for good results using the current version of Open Inventor with medical imaging data. First we create an *SoSeparator* to hold all the VolumeViz nodes. It’s not required for a simple example, but it’s good practice because we don’t want the OpenGL shaders that VolumeViz automatically installs to affect rendering of regular geometry. Also as part of the setup we disable the in-memory compression feature. This is also not required for a simple example, but generally provides better performance for larger medical data sets.

We create an *SoVolumeData* node and load a DICOM data set by specifying the directory name. We also change the LDM tile size to a larger value to make data management more efficient (discussed earlier in the Concepts section). This is not required for a simple example, but generally provides better performance for larger medical data sets. Next we create an *SoDataRange* node to specify the range of data values that should be mapped to the color/opacity lookup table. This is normally unnecessary for 8-bit (byte) data so we only do this for larger data types. The actual min and max values for the volume can be queried from the volume if necessary, but typically we’ll have some idea about the interesting range of values already. In this case we assume we’re loading a calibrated DICOM data set. Next we create an *SoTransferFunction* node and set the predefined INTENSITY color map. This is a gray scale ramp with constant opacity. Next we create an *SoMaterial* node to specify material properties of the voxels. This is not necessary for a simple example, but the default diffuse color value in Open Inventor is 0.8 gray, so we change that to full white in order to use the full range of intensities on the display screen. Finally we create an *SoOrthoSlice* (axis aligned slice) node, set the axis to the Z axis (the K axis of the volume) and set the slice number to position the slice at the center of the volume (based on the voxel dimensions of the volume which we queried from the *SoVolumeData* node). As discussed in the Coordinate Systems section, this will be an Axial slice. We also set the interpolation mode. This is not required for a simple example, but multi-sampling produces much higher image quality than the default linear interpolation.

Not shown here are the “#include” (C++), “import” (Java) or “using” (C#) statements to access these nodes. This works the same as for core Open Inventor nodes. If you’re not sure about the include path or assembly name, just look up the node in the reference manual. Also not shown are the initialization and cleanup statements (C++ only) or the creation and initialization of a viewer to render the scene graph. These steps are essentially the same as for any Open Inventor application. Later we will discuss viewers and how to add interactivity to your application.

Slice rendering scene graph:

```
// Root of scene graph
SoSeparator* root = new SoSeparator();

// Default setting can be a performance bottleneck
SoPreferences::setBool( "LDM_USE_IN_MEM_COMPRESSION", FALSE );

// Keep volume viz separate from geometry
SoSeparator* volSep = new SoSeparator();
root->addChild( volSep );

// Load volume data
```



```
SoVolumeData* volData = new SoVolumeData();
volData->fileName = "DicomDirectory";
volData->ldmResourceParameters.getValue()->tileDimension(512,512,512);
volSep->addChild( volData );

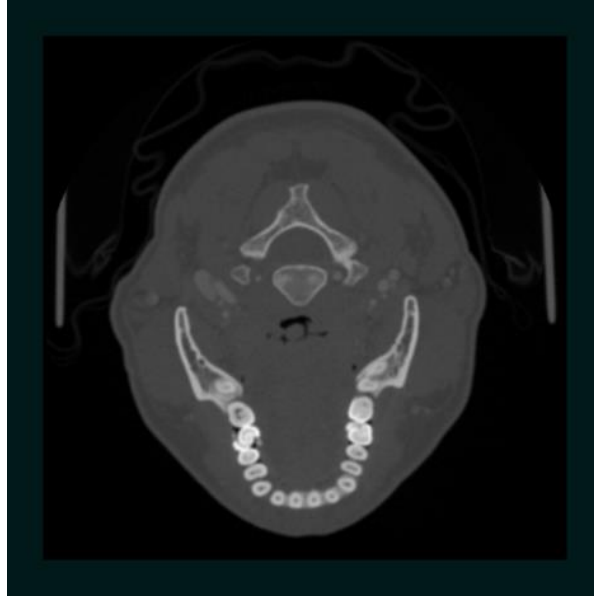
// Set range of data values to visualize.
SoDataRange* volRange = new SoDataRange();
if (volData->getDatumSize() > 1) {
    volRange->min = -1000;
    volRange->max = 3000;
}
volSep->addChild( volRange );

// Load opaque intensity ramp
SoTransferFunction* volTF = new SoTransferFunction();
volTF->predefColorMap = SoTransferFunction::INTENSITY;
volSep->addChild( volTF );

// Display slice at full intensity
SoMaterial* volMat = new SoMaterial();
volMat->diffuseColor.setValue( 1, 1, 1 );
volSep->addChild( volMat );

// Display an Axial (Z axis) slice at center of volume
SoOrthoSlice* slice = new SoOrthoSlice();
SbVec3i32 voldim = volData->data.getSize(); // Dimension in voxels
slice->axis = SoOrthoSlice::Z;
slice->sliceNumber = voldim[2] / 2;
slice->interpolation = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( slice );
```

The resulting image will look something like this:



Slices are 3D objects:

It's important to understand what we've just done. We loaded a volume data set into our 3D world coordinate space and displayed one Axial slice. Even though it looks like a 2D rendering here, Open Inventor is a 3D visualization toolkit and we're always in a 3D world. That has some important implications, including:

- Each slice is really oriented in its corresponding plane in 3D.
If we now want to display, for example, a Coronal slice, we need to do two things. First, set the *SoOrthoSlice* node's axis field to Y and second, rotate the camera to look down the Y axis.
- Each slice is actually positioned at its corresponding location in 3D.
We can display a different slice by simply setting the 'sliceNumber' field of the *SoOrthoSlice* node, but that slice is at a slightly different position along the (in this case) Z axis. If we used the default perspective camera, the slice would visually be slightly larger or slightly smaller. However we can give the user a "2D view" of the slice by using an *SoOrthographicCamera* node. We'll discuss cameras in the next section.

A 2D view of slices:

As we just discussed, for a 2D view of slices, we will normally use an orthographic camera that does not apply perspective. In a 3D view, many medical applications will also use an orthographic camera, but we might also want to use a perspective camera in some cases. We also need to think about the viewing, or navigation, model that the user will have. Open Inventor provides several prebuilt viewer classes with slightly different behaviors. (Note the exact class name depends on the language, platform and window system you are developing for. See the documentation for more details.) The *PlaneViewer*, e.g. *SoWinPlaneViewer*, allows the user to pan and zoom (but not rotate) using the left mouse button. This is the viewer we'll most often use for 2D views. The *ExaminerViewer*, e.g. *SoWinExaminerViewer*, allows the user to pan and zoom plus rotate (orbit) around the data set using the left mouse button. This is the viewer we'll most often use for 3D views. You can use an orthographic camera or a perspective camera with any viewer class. It's an independent choice.

Now let's look at the rest of the code that we need to create a 2D view of this slice. We already know how to create the scene graph, so we just need to add a camera to the scene graph, create the viewer and set some

parameters. Most importantly we need to tell the viewer what scene graph to display using the `setSceneGraph()` method. We may want to set the background color to something slightly different from black, so the user can see where the edges of the slice are. We'll also want to set the initial camera position so the whole slice is visible in the window. We can do this manually by setting the parameters of the camera node, but an easier way is to call the `viewAll()` method and let the viewer compute the initial camera settings. Then we call `saveHomePosition()` so the user can get back to the initial settings using the viewer's user interface (which you can replace). In many cases it is not necessary to explicitly create a light node to illuminate the scene. The viewer will automatically create a "headlight", a directional light that always shines in the direction the camera is looking. If necessary, the application can query this light node (see the `getHeadlight()` method) and modify its fields. The application can also explicitly create one or more lights using the *SoDirectionalLight* node.

2D Axial Slice View

```
// Root of scene graph
SoSeparator* root = new SoSeparator();

// Create orthographic camera
SoOrthographicCamera* camera = new SoOrthographicCamera();
root->addChild( camera );

// Keep volume viz separate from geometry
SoSeparator* volSep = new SoSeparator();
root->addChild( volSep );

// Create slice rendering scene graph as in previous example
. . .

// Display an Axial (Z axis) slice at center of volume
SoOrthoSlice* slice = new SoOrthoSlice();
slice->axis          = SoOrthoSlice::Z;
slice->sliceNumber   = 256;
slice->interpolation = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( slice );

// Create and initialize viewer
SoWinPlaneViewer* viewer = new SoWinPlaneViewer( parentWindow );
viewer->setSceneGraph( root );           // What to display
viewer->setBackgroundColor( SbColor(0,0.1f,0.1f) ); // Optional
viewer->viewAll();                       // Adjust camera
viewer->saveHomePosition();              // Save camera
```

Note that although it is often convenient to use the prebuilt Open Inventor viewer classes, especially for prototyping, it is not required. You can use the viewer's parent class, *RenderArea*, e.g. *SoWinRenderArea*, to create a basic 3D drawing window and manage the scene graph, then implement your own viewing model by handling events and modifying the camera node. Building a custom navigation is illustrated by the code in the *ViewerComponents* example directory included with the Open Inventor SDK. A number of convenient viewing algorithms are provided in the *SoCameraInteractor* class.

The other 2D slices:

To display Coronal and Sagittal slices, the creation and setup of the viewer will be the same. We need to rotate the camera so it looks down the correct axis (typically Y for Coronal and X for Sagittal) and we need to tell the slice node to use that axis so it will extract the correct data for display. The camera node's 'orientation' field defines a rotation that is applied to the default orientation of the camera. By default the camera is positioned on the +Z

axis, looking in the $-Z$ direction (Inferior) direction. For the Coronal slice we rotate the camera 90 degrees around the X axis so we're looking in the $-Y$ direction. For the Sagittal slice we rotate the camera 90 degrees around the Y axis so we're looking in the $-X$ direction. Simply rotating the camera "in-place" is not enough, we also need to re-position the camera to be, for example, on the $+Y$ axis for Coronal. However calling the `viewAll()` method on the viewer will automatically take care of that.

2D Coronal Slice View

```
// Create orthographic camera for Coronal (Y axis) slice
SoOrthographicCamera* camera = new SoOrthographicCamera();
camera->orientation.setValue( SbVec3f(1,0,0), (float)M_PI/2 ); // Axis, angle
root->addChild( camera );

. . .

// Display a Coronal (Y axis)
SoOrthoSlice* slice = new SoOrthoSlice();
slice->axis          = SoOrthoSlice::Y;
slice->sliceNumber    = 256;
slice->interpolation  = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( slice );
```

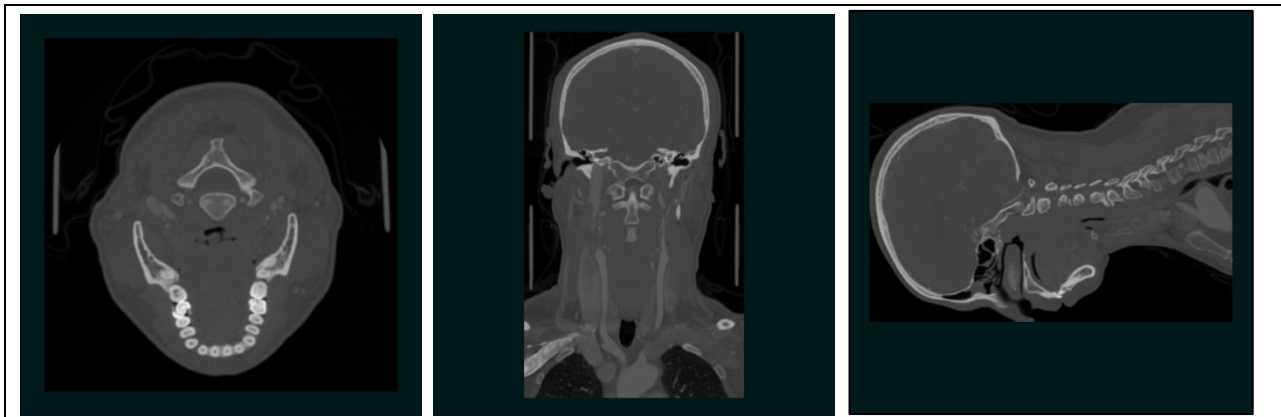
2D Sagittal Slice View

```
// Create orthographic camera for Sagittal (X axis) slice
SoOrthographicCamera* camera = new SoOrthographicCamera();
camera->orientation.setValue( SbVec3f(0,1,0), (float)M_PI/2 ); // Axis, angle
root->addChild( camera );

. . .

// Display a Sagittal (X axis)
SoOrthoSlice* slice = new SoOrthoSlice();
slice->axis          = SoOrthoSlice::X;
slice->sliceNumber    = 256;
slice->interpolation  = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( slice );
```

Axial, Coronal and Sagittal slices:



A 3D view of slices:

To display the slices in 3D is straightforward. We don't need to rotate the camera (we'll let the user do that at run-time), but we might want to create a perspective camera instead of the orthographic camera. We will add three slices to the scene graph, each with a different axis. This is an example of using the same volume data node to render multiple objects. Finally, we will create an ExaminerViewer instead of the PlaneViewer because it provides a more natural navigation model for data in 3D.

```
// Root of scene graph
SoSeparator* root = new SoSeparator();

// Create camera
SoPerspectiveCamera* camera = new SoPerspectiveCamera();
root->addChild( camera );

// Create slice rendering scene graph
. . .

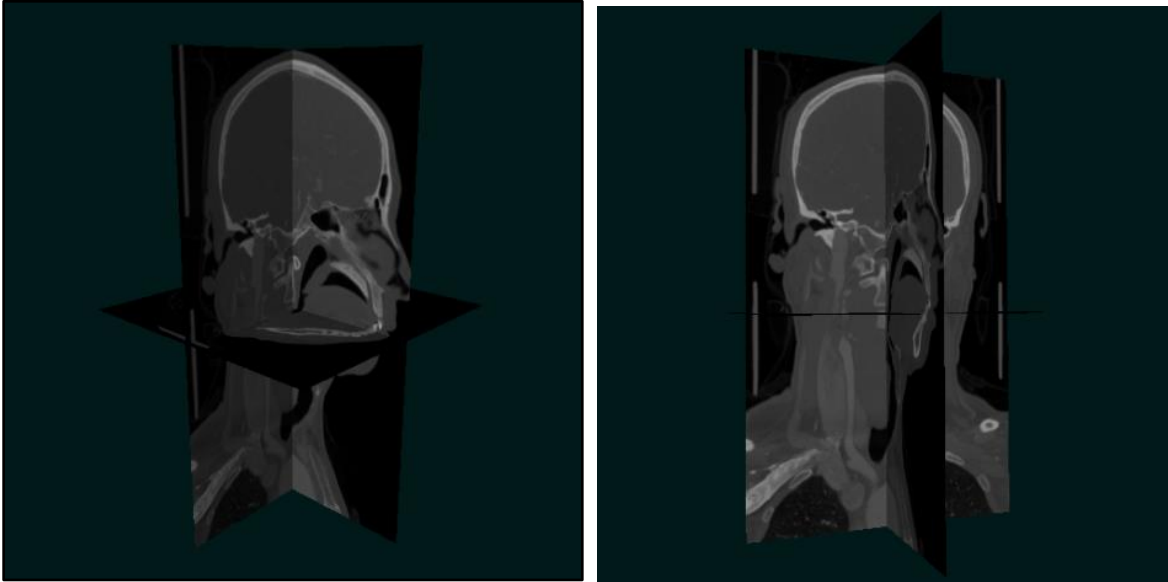
// Display Axial slice at center of volume
SoOrthoSlice* Aslice = new SoOrthoSlice();
Aslice->axis          = SoOrthoSlice::Z;
Aslice->sliceNumber   = 256;
Aslice->interpolation = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( Aslice );

// Display Coronal slice at center of volume
SoOrthoSlice* Cslice = new SoOrthoSlice();
Cslice->axis          = SoOrthoSlice::Y;
Cslice->sliceNumber   = 256;
Cslice->interpolation = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( Cslice );

// Display Sagittal slice at center of volume
SoOrthoSlice* Sslice = new SoOrthoSlice();
Sslice->axis          = SoOrthoSlice::X;
Sslice->sliceNumber   = 256;
Sslice->interpolation = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( Sslice );

// Create and initialize viewer
SowinExaminerViewer* viewer = new SowinExaminerViewer( parentWindow );
viewer->setSceneGraph( root );           // What to display
viewer->setBackgroundColor( SbColor(0,0.1f,0.1f) ); // Optional
viewer->viewAll();                       // Adjust camera
viewer->saveHomePosition();              // Save camera
```

Axis aligned slices in 3D:



Interaction:

We will discuss event handling and interaction in more detail in a later section. For now, just note that Open Inventor provides a dragger (3D widget) class *SoOrthoSliceDragger* specifically for interacting with ortho slices in 3D. When this dragger is attached to a slice the user can click on the slice and interactively drag it along its axis using the mouse or touch input. This provides a natural way of manipulating objects directly in the 3D scene instead of using (only) a 2D user interface object like a “slider”. *SoOrthoSliceDragger* is a specialization of the *SoTranslate1Dragger* class.

```
// Display Sagittal slice at center of volume
SoOrthoSlice* Sslice = new SoOrthoSlice();
Sslice->axis          = SoOrthoSlice::X;
Sslice->sliceNumber    = 256;
Sslice->interpolation  = SoOrthoSlice::MULTISAMPLE_12;
volSep->addChild( Sslice );

// Create path to slice node
// Note: Can be a partial path but must include the slice node.
// Here we start at the volSep Separator.
SoPath* Spath = new SoPath( volSep );
Spath->append( Sslice );

// Create and initialize dragger.
SoOrthoSliceDragger* Sdragger = new SoOrthoSliceDragger();
Sdragger->orthoSlicePath  = Spath;
Sdragger->volumeDimension = volData->data.getSize();
Sdragger->volumeExtent    = volData->extent.getValue();
volSep->addChild( Sdragger );
```

Arbitrary planar slice:

In a 3D view we can also create a planar slice with arbitrary orientation, using the *SoObliqueSlice* class. This could be useful to align a slice with the primary direction of some structure inside the volume. The scene graph setup

and usage is essentially the same as for *SoOrthoSlice* so we will not go into the details here. The big difference is that the position and orientation of the slice is defined by a mathematical plane in 3D space (see the *SbPlane* class). That means working in XYZ coordinates, not voxel coordinates. There are conversion methods in *SoVolumeData*, e.g. *voxelToXYZ()*, that may be helpful. The plane can be specified by its normal vector and distance from the origin or by three points that define a plane. See the example programs in the SDK.

Interaction:

It is also possible to allow the user to directly manipulate an oblique slice in 3D. One solution is to use the standard *SoJackDragger* class. However this dragger is not very intuitive for users that are not familiar with Open Inventor. Another solution is to use the *ClipPlaneDragger* class available in the Resources section of the Open Inventor Forum (www.openinventor.net). Despite the name, this dragger is convenient for manipulating any object defined by a plane and has a simpler, more intuitive appearance.

Non-planar slice:

We can also create non-planar slices, for example a curved slice that follows the path of some structure inside the volume. This is sometimes called Curved Planar Reformation (CPR). In fact, using standard polygonal geometry such as triangles, we can create a “slice” of almost any shape and *VolumeViz* will extract the values of the voxels intersected by the shape. This feature is called “volume geometry” in Open Inventor. See, for example, the *SoVolumeIndexedFaceSet* node. The scene graph is similar to other slices, so we will not go into the details here. See the example programs in the SDK.

Thick slice (slab):

Finally, we can also do “thick slice” or slab rendering using Maximum Intensity Projection (MIP) and similar techniques. There is no specific class for thick slice rendering, but it is straightforward to implement using the *SoVolumeRender* and *SoROI* nodes. Both of these nodes are discussed in the volume rendering section, so we will be brief here. *SoVolumeRender* is a rendering node that does direct volume rendering using a ray-casting algorithm. Using the ‘renderMode’ field, you can select Maximum Intensity Projection (MIP), Minimum Intensity Projection (MinIP), Sum Intensity Projection (SumIP) or Average Intensity Projection (AvgIP). *SoROI* is a property node that specifies an axis aligned “region of interest” in voxel IJK coordinates, essentially cropping the volume. We’ll use the *SoROI* node to control the thickness of the slice. We’ll set the ‘box’ field to the full extent of the volume along the two perpendicular axes. Along the slice axis we’ll set it to a small region centered on the thick slice position. Similar to rendering a normal ortho slice, we’ll create an orthographic camera and orient it to look down the slice axis.

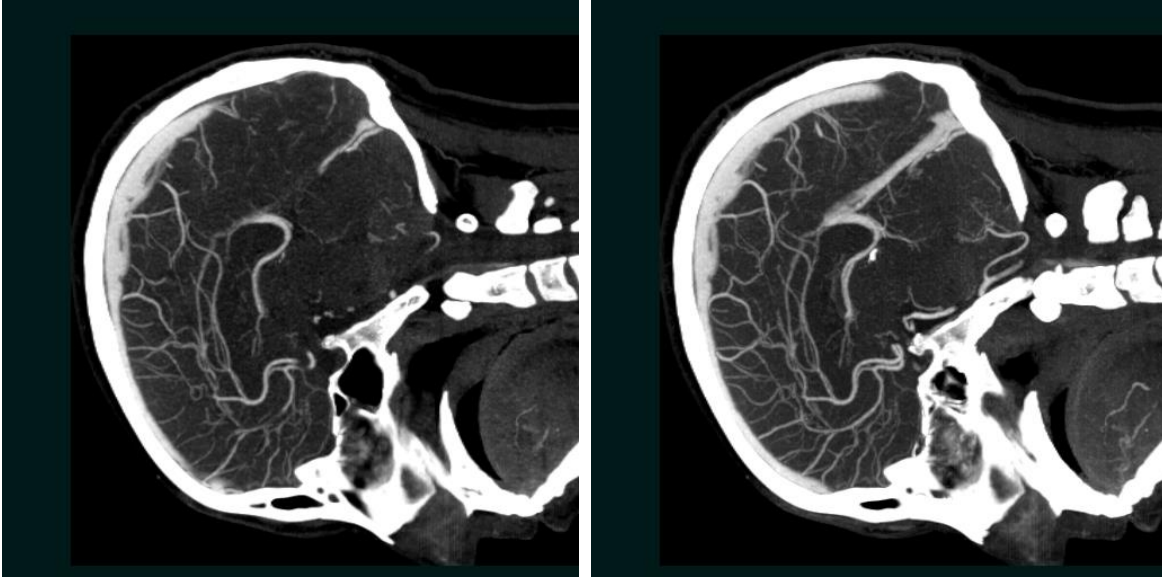
```
// Create the usual 2D slice scene graph with orthographic camera.
// The rendering node (below) will be different.
. . .

// Create ROI node to crop the volume to a Sagittal (X axis) thick slice.
// Assume the volume is 512 x 512 x 512.
// The ROI box is defined by: xmin,ymin,zmin, xmax,ymax,zmax.
// If X voxel spacing is 0.408 mm, 6 slices is approximately a 2.5 mm slab.
SoROI* volROI = new SoROI();
volROI->box.setValue( 259,0,0, 265,511,511 );
volSep->addChild( volROI );

// Create volume rendering node using MIP rendering.
SoVolumeRender* volRend = new SoVolumeRender();
volRend->renderMode = SoVolumeRender::MAX_INTENSITY_PROJECTION;
```

```
volSep->addChild( volRend );
```

A 2.5 mm slab and a 10 mm slab (Sagittal) using Maximum Intensity Projection:



As with the other slices, the thick slice is really a 3D rendering. We're just looking at it in a 2D view in this example. Exactly the same technique can be used to render an axis-aligned "slab" in a 3D view. In this case it is also possible to allow the user to click and drag the slab through the volume. See the *SoROIManip* class.

Finally, it is also possible to render a slab aligned with an arbitrary plane, effectively a thick oblique slice. This can be done using an *SoUniformGridClipping* node to clip the volume and an *SoTransform* node to orient the clipping plane. This is an advanced topic and not explained here.

Isosurface Visualization

Display an Isosurface

VolumeViz provides the ability to render isosurfaces directly on the GPU without time consuming extraction of actual geometry. This is very powerful because isosurfaces are now an interactive tool for exploring the structures inside a volume data set by interactively changing the isosurface value. VolumeViz can even render multiple isosurfaces in one pass and each isosurface can have its own color and opacity value.

The code to create a basic VolumeViz scene graph for isosurface rendering is shown below. It's quite similar to the code for creating a slice rendering scene graph except we use an *SoMaterial* node to specify the color and opacity of each isosurface and an *SoVolumeRender* node to do the rendering. Normally *SoVolumeRender* would do direct volume rendering, but a special property node called *SoVolumelsosurface* sets the isosurface rendering mode and specifies the isosurface values. (In this case the 'renderMode' field that we used for thick slices is ignored.) As before, we create an *SoSeparator* to hold all the VolumeViz nodes, an *SoVolumeData* node to load a DICOM data set and an *SoDataRange* node to specify the range of data values. Even though we're not using a color lookup table, we need the *SoDataRange* node to specify the range of data values that will be scaled into the GPU voxel values. Only isosurface values within this range are meaningful.

Previously the next step was to create an *SoTransferFunction* node. In this case it's unnecessary. We create an *SoMaterial* node to specify the color and opacity of the isosurface, an *SoVolumelsosurface* node to enable isosurface rendering and specify the isosurface values and finally, an *SoVolumeRender* node to do the actual rendering. This is a 3D rendering technique, so we will typically use the camera and viewer setup that we used for 3D slice rendering previously. In fact we can freely mix slice and isosurface rendering if we wish (as well as other kinds of rendering).

Notice that we set some other properties on the *SoVolumeRender* node to non-default values. These are not required, but are useful to help manage the balance between interactive performance and image quality. The 'numSlicesControl' field (despite its historical name) specifies how VolumeViz should choose the number of samples along each ray. The AUTOMATIC setting allows VolumeViz to manage this setting, in particular to automatically reduce the number of samples while the user is interacting. The 'lowResMode' and 'lowScreenResolutionScale' specify how VolumeViz should manage the number of rays cast into the volume. For static rendering (not interacting), VolumeViz will basically cast one ray per (relevant) pixel in the window. For interactive rendering VolumeViz will (in this example) only cast one ray every two pixels, significantly reducing the amount of computation needed to render the volume at the cost of a slight blurring of the image.

Isosurface rendering scene graph:

```
// Keep volume viz separate from geometry
SoSeparator* volSep = new SoSeparator();
root->addChild( volSep );

// Load volume data
SoVolumeData* volData = new SoVolumeData();
volData->fileName = "DicomSample";
volData->ldmResourceParameters.getValue()->tileDimension(512,512,512);
volSep->addChild( volData );

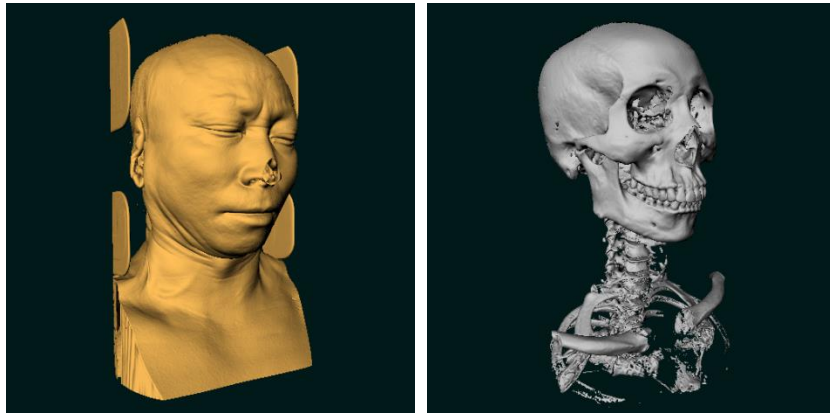
// Set range of data values to visualize.
SoDataRange* volRange = new SoDataRange();
volRange->min = -1000;
volRange->max = 3000;
volSep->addChild( volRange );
```

```
// Define color/opacity for isosurface
SoMaterial* volMat = new SoMaterial();
volMat->diffuseColor.setValue( 1, 0.75f, 0.35f ); // Skin
volSep->addChild( volMat );

// Isosurface settings
SoVolumeIsosurface* volIso = new SoVolumeIsosurface();
volIso->isovalues = -300; // Skin
volSep->addChild( volIso );

// Render isosurface
SoVolumeRender* volRend = new SoVolumeRender();
// Let Open Inventor compute best number of slices
volRend->numSlicesControl = SoVolumeRender::AUTOMATIC;
// Optional: Use lower screen resolution while moving.
volRend->lowResMode = SoVolumeRender::DECREASE_SCREEN_RESOLUTION;
volRend->lowScreenResolutionScale = 2;
volSep->addChild( volRend );
```

Isosurfaces at values -300 and 500 in this data set look like this:



Multiple isosurfaces:

Although we've been using them as single value fields up to now, the 'diffuseColor' and 'transparency' fields in the *SoMaterial* node, as well as the 'isovalues' field in the *SoVolumeIsosurface* node, are actually multi-value fields (*SoMFVec3f* and *SoMFFloat* respectively). Multi-value fields are "smart" container objects that automatically manage memory to hold as many values as the application specifies. Notice that we use the `set1Value()` method now and the first parameter is the index of the value to set. To render multiple isosurfaces, just add the desired number of iso-values in the *SoVolumeIsosurface* node's 'isovalues' field. Then in the *SoMaterial* node set the diffuseColor and transparency for each isosurface. All isosurfaces share the first specularColor and shininess values (these are usually the same for all materials). If there are not enough color values, the last color value is re-used for all the remaining isosurfaces.

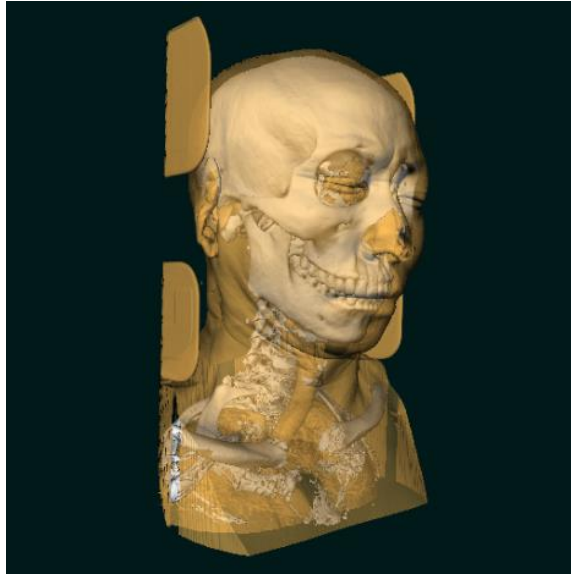
TIP: If you're using transparent isosurfaces, always specify the opaque isosurface(s) first.

```
// Define color/opacity for each isosurface
SoMaterial* volMat = new SoMaterial();
volMat->diffuseColor.set1Value( 0, 1, 1, 1 ); // Bone
volMat->transparency.set1Value( 0, 0 );
```

```
volMat->diffuseColor.set1Value( 1, 1, 0.75f, 0.35f ); // Skin
volMat->transparency.set1Value( 1, 0.4f );
volSep->addChild( volMat );

// Isosurface settings
SoVolumeIsosurface* volIso = new SoVolumeIsosurface();
volIso->isovalues = 500; // Bone
volIso->isovalues = -300; // Skin
volSep->addChild( volIso );
```

Two isosurfaces (one is partially transparent):



Extract isosurface geometry.

VolumeViz renders isosurfaces on the GPU using shaders that create an image of the isosurface directly from the volume data. The advantage of this approach is very high performance. The isosurface values can be changed interactively. However no actual geometry (e.g. triangles) is computed or stored anywhere.

If the application needs the actual isosurface geometry, you can use the MeshVizXLM extension. The “VolumeMesh” example in the MeshVizDataMapping directory of the Open Inventor SDK shows one way to do this. The efficiency and flexibility of the MeshVizZLM API is illustrated because we are able to extract the iso-value surface mesh without making a copy of the actual volume data.

3D Volume Visualization

Direct Volume Rendering

VolumeViz does high quality volume ray-casting entirely on the GPU. There are many options that affect performance and image quality. There are multiple rendering modes (compositing, MIP, MinIP, etc) and rendering effects (lighting, ambient occlusion, boundary opacity, edge coloring, voxelized rendering, etc.) that can be selected. Please see section 1.5 of the VolumeViz chapter in the Open Inventor Mentor for detailed information (<http://oivdoc.vsg3d.com/content/15-volume-rendering>).

The code to create a basic VolumeViz scene graph for volume rendering is shown below. It's quite similar to the scene graphs we've seen already for slices and isosurfaces except we introduce the *SoVolumeRenderingQuality* node for setting volume rendering options. As before, we create an *SoSeparator* to hold all the VolumeViz nodes, an *SoVolumeData* node to load a DICOM data set and an *SoDataRange* node to specify the range of data values. We create an *SoTransferFunction* node to specify the color/opacity lookup table. We create an *SoMaterial* node to specify the base material for the voxels and finally an *SoVolumeRender* node to do the actual rendering. This is a 3D rendering technique, so we will typically use the camera and viewer setup that we used for 3D slice rendering previously. In fact we can freely mix slice and isosurface rendering if we wish (as well as other kinds of rendering).

Notice that we set some other properties on the *SoVolumeRender* node to non-default values. These are not required, but are useful to help manage the balance between interactive performance and image quality. The 'numSlicesControl' field (despite its historical name) specifies how VolumeViz should choose the number of samples along each ray. The AUTOMATIC setting allows VolumeViz to manage this setting, in particular to automatically reduce the number of samples while the user is interacting. The 'lowResMode' and 'lowScreenResolutionScale' specify how VolumeViz should manage the number of rays cast into the volume. For static rendering (not interacting), VolumeViz will basically cast one ray per (relevant) pixel in the window. For interactive rendering VolumeViz will (in this example) only cast one ray every two pixels, significantly reducing the amount of computation needed to render the volume at the cost of a slight blurring of the image.

Volume rendering scene graph:

```
// Keep volume viz separate from geometry
SoSeparator* volSep = new SoSeparator();
root->addChild( volSep );

// Load volume data
SoVolumeData* volData = new SoVolumeData();
volData->fileName = "DicomSample";
volData->ldmResourceParameters.getValue()->tileDimension(512,512,512);
volSep->addChild( volData );

// Set range of data values to visualize.
SoDataRange* volRange = new SoDataRange();
volRange->min = -1000;
volRange->max = 3000;
volSep->addChild( volRange );

// Load constant intensity with alpha ramp
SoTransferFunction* volTF = new SoTransferFunction();
volTF->predefColorMap = SoTransferFunction::GRAY;
volSep->addChild( volTF );

// Display volume at full intensity
SoMaterial* volMat = new SoMaterial();
volMat->diffuseColor.setValue( 1, 1, 1 );
```

```

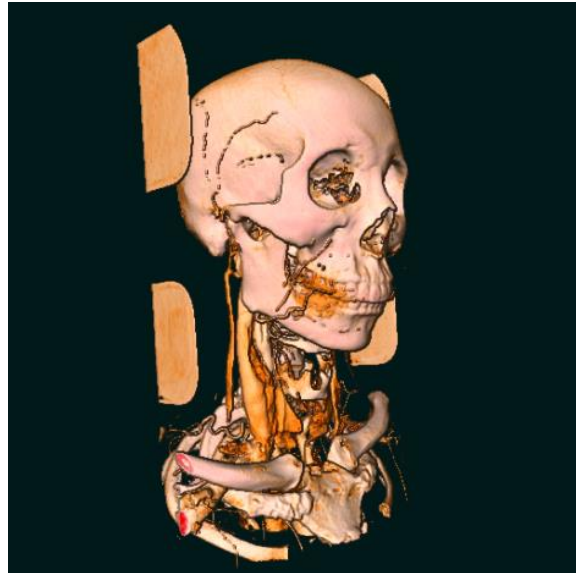
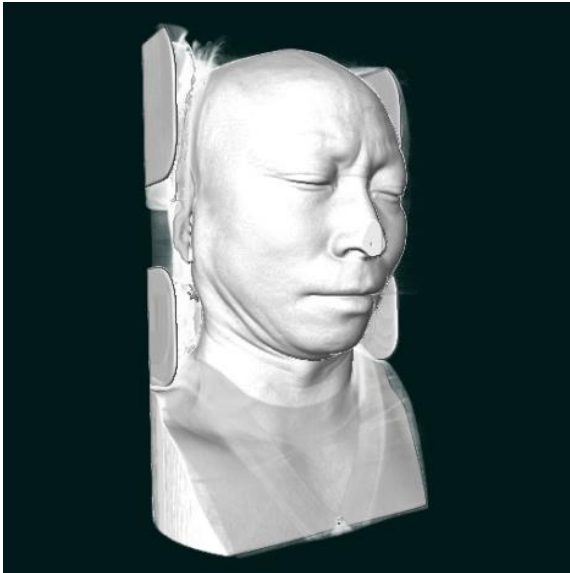
volSep->addChild( volMat );

// Volume rendering settings
SoVolumeRenderingQuality* volQual = new SoVolumeRenderingQuality();
// Higher quality rendering
volQual->preIntegrated = TRUE;
// Optional: Enable screen space lighting
volQual->deferredLighting = TRUE;
volQual->surfaceScalarExponent = 5;
volSep->addChild( volQual );

// Display volume rendering
SoVolumeRender* volRend = new SoVolumeRender();
// Let Inventor compute best number of slices
volRend->numSlicesControl = SoVolumeRender::AUTOMATIC;
// Optional: Use lower screen resolution while moving.
volRend->lowResMode = SoVolumeRender::DECREASE_SCREEN_RESOLUTION;
volRend->lowScreenResolutionScale = 2;
// Internal optimization.
volRend->subdivideTile = TRUE;
// Optimize ray-casting for surfaces in data
volRend->samplingAlignment = SoVolumeRender::BOUNDARY_ALIGNED;
// Ignore low opacity voxels
volRend->opacityThreshold = 0.1f;
volSep->addChild( volRend );

```

Volume rendering with the predefined GRAY color map and with a custom color map:



More performance/quality settings

We've already used the 'lowResMode' option on the *SoVolumeRender* node to automatically reduce the number of rays cast while the user is interacting. More options are available using the *SoInteractiveComplexity* node. This node can automatically reduce the number of samples along each ray by automatically changing the "complexity" value (see *SoComplexity*). It can also automatically change various rendering options to less expensive values. For example, changing the interpolation mode from CUBIC (high quality, but relatively slow) to LINEAR. The *SoInteractiveComplexity* node takes a list of strings containing a node class name, a field name, an "interactive" value and a "static" value. See the reference manual for more details. Here is an example of typical use with

volume rendering (add this code to your volume rendering scene graph just before the *SoVolumeData* node):

```
// Decrease quality while moving in order to have better interactivity
SoInteractiveComplexity* interact = new SoInteractiveComplexity();
// Decrease the "number of samples" (see SoVolumeRender numSlicesControl)
interact->fieldSettings.set1Value( 0, "SoComplexity value 0.2 0.5" );
// Decrease interpolation quality.
interact->fieldSettings.set1Value( 1, "SoVolumeRender interpolation LINEAR CUBIC" );
// Don't wait before returning to full quality rendering.
interact->refinementDelay = 0;
volSep->addChild( interact );

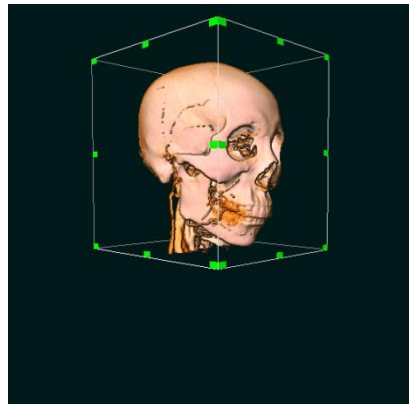
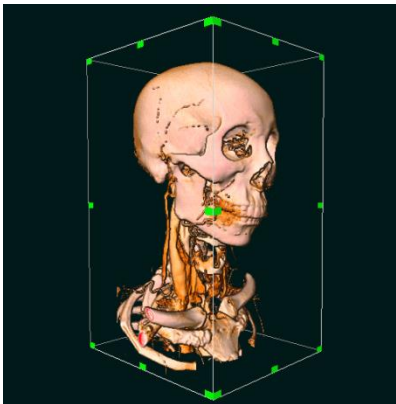
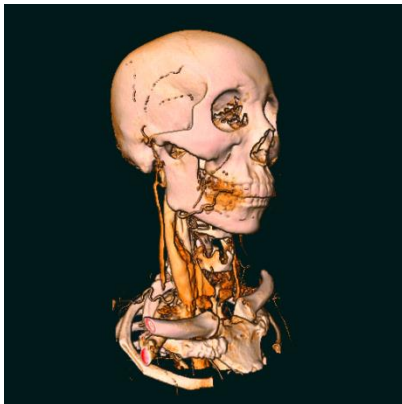
// Complexity node for the interactiveComplexity node to control.
SoComplexity* volComp = new SoComplexity();
volSep->addChild( volComp );
```

Volume Cropping

VolumeViz provides multiple ways to clip away parts of a volume (see the Overview section). A simple but very useful tool is the *SoROI* (Region of Interest) node. *SoROI* is limited to clipping with axis-aligned planes, but it is much more efficient than using traditional clip planes (although these also work with volumes, see *SoClipPlane*). For example, it might be useful to crop the volume shown above to remove the objects that are obscuring part of the skull on the left and right side. The dimensions of this volume are 512 x 512 x 557. So, by default, the region of interest is the box defined by IJK points [0, 0, 0] and [511, 511, 556]. In this case we just need to trim the X (Sagittal) dimension from 0..511 to 25..480. Insert the *SoROI* node before the *SoVolumeRender* node:

```
// Remove some voxels using Region of Interest
SoROI* volRoi = new SoROI();
// ROI box is defined by a min point and max point in IJK coordinates.
volRoi->box.setValue( 25,0,0, 480,511,556 );
volSep->addChild( volRoi );
```

The result of applying the Region of Interest is shown in the first image below. The *SoROI* node also has a subclass called *SoROIManip* that has a built-in 3D widget based on the *SoTabBoxDragger* class. *SoROIManip* can be inserted in the scene graph in place of *SoROI*. It automatically displays a user interface that allows the user to resize the Region of Interest box by clicking and dragging the green tabs. For example the second and third images below.



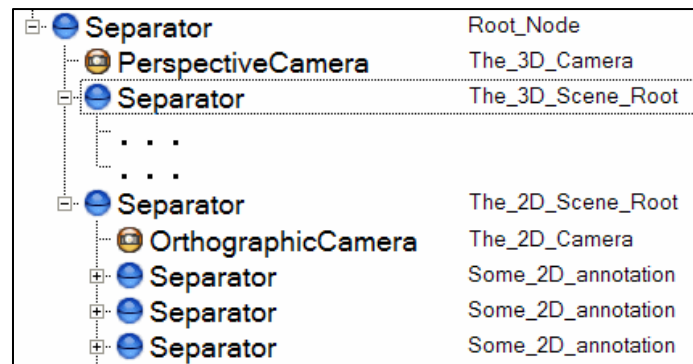
Miscellaneous Bits

In this section we show how to implement some utility features commonly used in medical visualization applications. These features are not specific to the VolumeViz or ImageViz extensions and can easily be implemented using the general purpose tools available in the core Open Inventor library.

Display text on screen (e.g. DICOM info)

3D applications often need to do a bit of 2D graphics too, particularly screen aligned annotation text. Medical applications often need to display information on screen ranging from the current slice number to various information from the DICOM header. Open Inventor provides a powerful mechanism for this by allowing multiple cameras (*SoCamera* nodes) in the scene graph. There are other uses for multiple cameras of course.

In this case we have one camera for the actual scene, usually controlled by the viewer, and a separate "2D" annotation camera (*SoOrthographicCamera*), in the scene graph at the same time. This is perfectly straightforward for Open Inventor because the camera is just a property node that happens to change the viewing and projection matrices. The viewer can only control one camera (the scene camera) at a time, so the user can move around the scene as usual and the 2D annotation camera does not change. We use the hierarchy of the scene graph to manage which geometry is viewed by which camera. The "scene" part of the scene graph will contain our application geometry and the "2D" part will contain our annotation geometry. The scene graph might look like this (this is only one possible way to organize the nodes):



By default, *SoOrthographicCamera* sets up a view volume that is -1 to 1 in both X and Y. We can consider this as a kind of normalized device coordinates (NDC). By default the camera will dynamically modify the view volume to match the aspect ratio of the viewport. But to use the view volume as NDC we need a static mapping to the viewport (i.e. stretching), so we set the camera node's 'viewportMapping' field to LEAVE_ALONE. We can use *SoFont* to set the font and text size as usual and an *SoTransform* node to position the text node as usual. Typically we'll use *SoText2* for screen annotation. Note that we can take advantage of the fact that the text node's 'string' field is a multi-value field. If we position a text node in the upper left corner of the window, we can set multiple strings in that node instead of positioning multiple text nodes.

Remember that we can query the *SoVRDicomData* object and access the DICOM header info. For example:

```

// Separator for annotation scene graph
SoAnnotation* annoSep = new SoAnnotation();
root->addChild( annoSep );

// OrthoCamera creates a -1 to 1 normalized device coordinates.
SoOrthographicCamera* annoCam = new SoOrthographicCamera();
annoCam->viewportMapping = SoCamera::LEAVE_ALONE;

```

```

annoSep->addChild( annoCam );

// Text attributes
SoFont* annoFont = new SoFont();
annoFont->name = "Arial:Bold";
annoFont->size = 16;
annoFont->renderStyle = SoFont::TEXTURE
annoSep->addChild( annoFont );

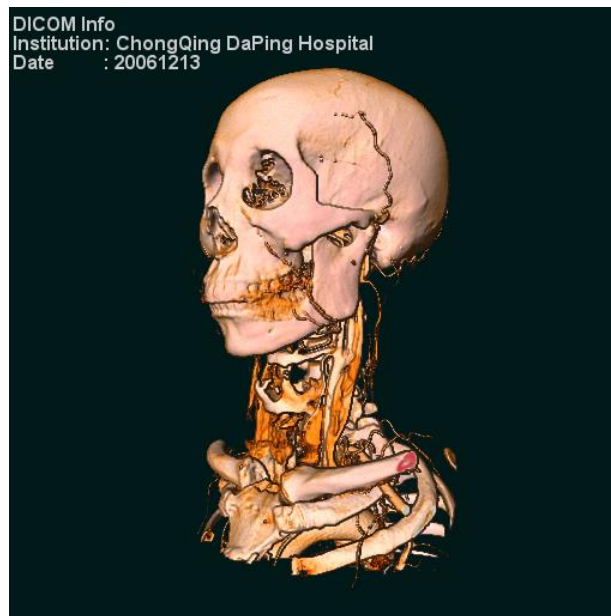
// Position text block
SoTransform* annoTran = new SoTransform();
annoTran->translation.setValue( -0.99f, 0.9, 0 );
annoSep->addChild( annoTran );

// Text block
SoText2* annoText = new SoText2();
int index = 0;
annoText->string.set1Value( index++, "DICOM Info:" );
annoSep->addChild( annoText

// Get strings from DICOM header
SoVRDicomFileReader* dicomReader = (SoVRDicomFileReader*)volData->getReader();
SbString str;
str = "Acquisition: " + dicomReader->getDicomData().getDicomInfo( 8, 128 );
annoText->string.set1Value( index++, str );
str = "Date       : " + dicomReader->getDicomData().getDicomInfo( 8, 34 );
annoText->string.set1Value( index++, str );

```

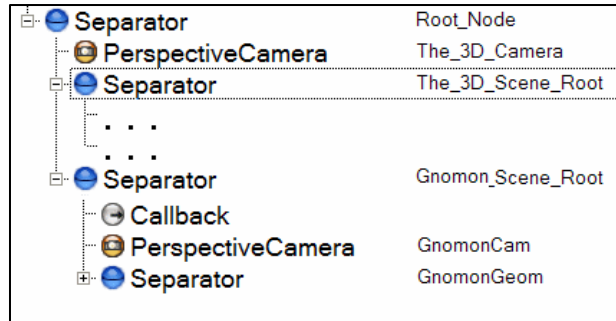
Result:



Show scene orientation (compass)

In a 3D window, medical applications typically display a small “compass” or “gnomon” in one corner of the screen to show the user the current orientation of the 3D space. This is another job for multiple cameras in the scene graph. As we did for screen annotation text (previous section), the scene graph has two sub-graphs, one for the actual 3D scene and another for the gnomon. Each sub-graph has its own camera. In this case the gnomon sub-

graph will contain some 3D geometry (not just text). Typically this is a cube with a letter on each face to indicate Anterior, Posterior, etc. To summarize, we make sure that the gnomon geometry has the same orientation in 3D as the subject, then we query the direction the main scene camera is pointing and update the gnomon camera to look in the same direction (but from a different position). The scene graph might look like this:



The key to updating the gnomon camera is adding an *SoCallback* node in the compass sub-graph. This node specifies an application function to be called when the node is traversed. The function is called with an *SoState* object that contains the current values of all the properties set during the traversal. Each property has an associated *SoElement* object. The application function can query the value of the *SoViewingMatrixElement* to get the current viewing matrix, set by the main scene camera, and set that matrix in the gnomon camera.

Notice that the gnomon geometry is rendered down in one corner of the window. This is done by setting a different “viewport”. The viewport is the region of the window that the camera’s 3D view volume is mapped into. The default viewport is the entire window and that’s what the main scene camera will use. We can change the viewport for the gnomon using an *SoViewport* node or by setting the *SoViewportRegionElement* in the callback function. The callback function will look something like the following. Some details have been omitted for clarity. Please see the actual code in several of the medical demo package example programs:

```
// Compass sub-graph callback function
static void gnomonRenderCB( void *userData, SoAction *action )
{
    // Don't do anything if this is not a render traversal
    if (action->isOfType(SoGLRenderAction::getClassTypeId())) {
        SoState *state = action->getState();

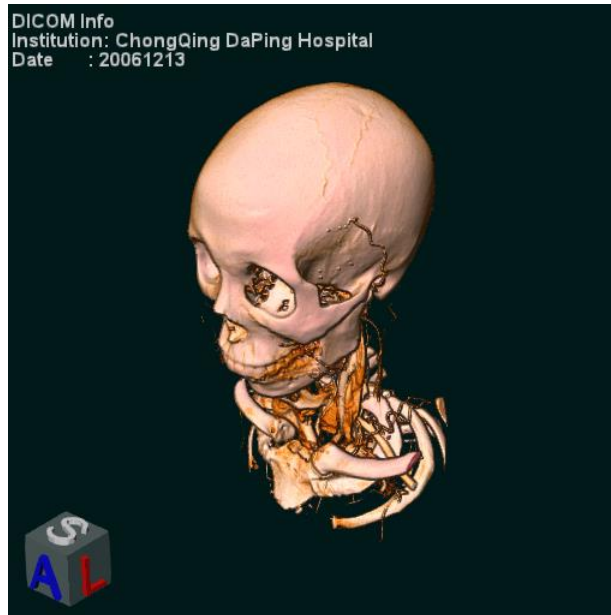
        // Set the viewport to position gnomon in the window.
        // This gnomon position is in the lower-left corner.
        SbViewportRegion viewport( 100, 100 );
        SoViewportRegionElement::set( state, viewport );

        // Get the current camera rotation from the viewing matrix.
        // Note that the viewing matrix is applied to geometry, so it's
        // actually the inverse of the matrix we need. The getTransform
        // method will return the rotation in variable cameraRotation.
        SbMatrix viewMat = SoViewingMatrixElement::get( state );
        SbVec3f tran, scale;
        SbRotation cameraRotation, orient;
        viewMat.inverse().getTransform( tran, cameraRotation, scale, orient );

        // Set the new orientation for the gnomon camera.
        SoCamera* gnomonCamera = (SoCamera*)userData;
        gnomonCamera ->orientation = cameraRotation;
    }
}
```

```
// After rotation the camera is no longer pointing at the gnomon.  
// Get the current "focal distance" (distance to pt-of-rotation)  
// and reposition the camera so it's looking at the pt-of-rotation  
// (which, in this case, we know is 0,0,0).  
float distance = gnomonCamera->focalDistance.getValue();  
SbMatrix mx;  
mx = cameraRotation;  
SbVec3f direction( -mx[2][0], -mx[2][1], -mx[2][2] );  
gnomonCamera->position = SbVec3f(0,0,0) - distance * direction;  
}  
}
```

Result:



Interaction

In this chapter we show how to implement some basic interaction features using mouse events. Open Inventor provides platform-independent handling for input events that occur inside an Open Inventor rendering window. *SoEvent* is the base class for all events. The Open Inventor render area class (parent of the viewer classes) automatically converts (relevant) raw system input events into the corresponding subclass of *SoEvent*. In addition to the usual keyboard events and mouse events, Open Inventor also supports touch events and gesture events (on a touch screen device), as well as 3D mouse events for SpaceBall/SpaceMouse type devices and tracked input events for immersive virtual reality environments. You can read more about these topics in the Open Inventor Mentor Volume 1, specifically chapter 10 (Events and Selection) and chapter 11 (Touch and Gesture Events). Open Inventor provides many other features that are useful for interaction, for example:

- **Picking**
Picking shoots a ray into the scene (typically starting at the current cursor position) and detects if any rendering objects, for example a slice or a volume rendering, are intersected. If a rendering object is intersected, Open Inventor can return information like the position in 3D (XYZ) coordinates, the position in voxel (IJK) coordinates and the value of the intersected voxel. See *SoRayPickAction* and *SoDetail*. Many rendering classes, e.g. *SoVolumeRender*, return a specific sub-class of *SoDetail* with extra information.
- **Screen drawing**
The *SoScreenDrawer* classes provide tools for interactively drawing lines, rectangles, ellipses and lassoes on top of the scene. These classes are useful for implementing measurement, area selection, etc.
- **Draggers**
The *SoDragger* classes are “3D widgets” that convert 2D input events, like dragging with the mouse, into constrained motion in 3D space. A *Translate1* dragger, for example, is constrained to move along a specified line. The *SoOrthoSliceDragger* is a specialization of this class that allows the user to click and drag slices in a 3D view. Read more about draggers in the Open Inventor Mentor chapter 17.

Events are handled by adding an *SoEventCallback* node to the scene graph. This node specifies an application function to be called when a specific type of event occurs. Typically you will add this node near the top of the scene graph. Delivering events by traversing the scene graph may sound strange at first, but it’s a powerful tool because it allows nodes with built-in event handling, like draggers, to automatically respond to user input.

Using the Mouse Wheel

Going back to our very first example, displaying an Axial or Coronal or Sagittal slice in a 2D view, we probably want to give the user one or more ways to change the slice number that is displayed. One way would be using the user interface toolkit (whatever we’re using for this application). For example, we could display an integer valued slider and change the ‘sliceNumber’ field of the *SoOrthoSlice* node when the slider value changes. This is straightforward and shouldn’t need to be explained here. But suppose we also want to allow the user to scroll through the slices by rotating the mouse wheel. That will require handling some events in the Open Inventor window. First we’ll add an *SoEventCallback* node to the scene graph and tell it to call an application function, *OnMouseWheel()* when an *SoMouseWheelEvent* is detected.

```
// Request a callback for mouse wheel events.
SoEventCallback* eventCB = new SoEventCallback;
eventCB->addEventCallback( SoMouseWheelEvent::getClassTypeId(), OnMouseWheel );
root->addChild( eventCB );
```

In the following code we assume that the *SoOrthoSlice* object is stored in a member variable named 'm_orthoSlice' in the application's state, so the callback function can access it. There are other options in Open Inventor that can be useful, especially for a test program or prototype. For example, when a node is created, you can give it a name using the *setName()* method, then later query that node using the *SoNode* class method *getByName()*. You can also use the *SoSearchAction* class to search the scene graph for a specific name or type of node. Changing the slice number by 1 could be tedious, so we check if the Shift key was pressed when this event occurred and change the slice number by 5 in that case. Note that we should be checking the slice number limits.

```
// Handler for mouse wheel events.
void OnMouseWheel( void* userData, SoEventCallback* node )
{
    // Get the specific event object and the mouse wheel "delta".
    const SoMouseWheelEvent* evt = (SoMouseWheelEvent*)node->getEvent();
    int delta = evt->getDelta();

    // Get the current slice number and increment or decrement.
    int sliceNumber = m_orthoSlice->sliceNumber.getValue();
    if (delta < 0) { // Rolling mouse wheel forward increments
        m_orthoSlice->sliceNumber = sliceNumber + (evt->wasShiftDown() ? 5 : 1);
    }
    else if (sliceNumber > 0) { // Don't let slice number go below zero.
        m_orthoSlice->sliceNumber = sliceNumber - (evt->wasShiftDown() ? 5 : 1);
    }
}
```

We'll assume that we can use annotation text (see [Display text on screen](#)) to display the slice number, so the result will look something like this:



Slice Position:

Suppose we also want to display the slice position in millimeters. If you remember the discussion about voxel size back in the [Concepts](#) section, this is straightforward. The distance from the edge of the volume to slice N is the minimum point of the volume plus the slice number times the voxel size plus one-half the voxel size. The extra

one-half is because slices are drawn at the center of the voxel. Assume, as above, that we have member variables ‘m_volData’ and ‘m_orthoSlice’ storing the *SoVolumeData* and *SoOrthoSlice* objects. In fact we could pre-compute most of the values used here, so the actual code is just a few lines, but for clarity we show the whole computation. We use the slice axis enumeration from the *SoOrthoSlice* node (0 : X, 1 : Y, 2 : Z) to index the correct values for the volume dimensions and extent.

```
// Compute slice position in millimeters.
const SbVec3i32& volDim = m_volData->data.getSize();
const SbBox3f& volExt = m_volData->extent.getValue();

int sliceNumber = m_orthoSlice->sliceNumber.getValue();
int sliceAxis = m_orthoSlice->axis.getValue();

float volumeStart = volExt.getMin()[sliceAxis];
float voxelSize = volExt.getSize()[sliceAxis] / volDim[sliceAxis];

float slicePos = volumeStart + (voxelSize * (sliceNumber + 0.5));
SbString text;
text.sprintf( "Position: %.3f mm", slicePos );
m_annoText->string.set1Value( 1, text );
```

Sensors:

This is a good place to mention another very powerful feature of Open Inventor called “sensors”. A data sensor is an “observer” that calls an application defined function when a value is changed in the scene graph. A sensor can monitor the entire scene graph, a sub-graph, a single node or even a single field. This is useful when several different parts of the application or its user interface can change a value that must be displayed to the user or must trigger changes to other values. For example, we might design the application so the ‘sliceNumber’ field in our *SoOrthoSlice* node can be changed by moving a user interface ‘slider’ or by rolling the mouse wheel or by some other user action. By attaching an *SoFieldSensor* to the ‘sliceNumber’ field, the associated function could be used to update the display of the slice number and position no matter how the slice number was actually changed. There is a short code example on the *SoDataSensor* reference manual page.

Querying voxels by pointing

Suppose that we want to investigate the specific values on the current slice by querying the value of the voxel currently under the cursor. We could use a mouse click to trigger this query, but we could also make the user’s experience more dynamic by continuously querying values as the cursor moves across the slice. Note that we can use the annotation screen text from a previous section to display the current voxel’s position and value directly in the window (probably in a different corner, but that’s a detail). Also note that querying the voxel under the cursor works in a 3D view as well, for both slice rendering and volume rendering. To get mouse motion events we register a callback for *SoLocation2Event*. In this example we create a new handler function, but we could use the same *SoEventCallback* node and/or same function to handle both mouse wheel events and mouse move events.

```
// Request a callback for mouse move events.
SoEventCallback* eventCB = new SoEventCallback;
eventCB->addEventCallback( SoLocation2Event::getClassTypeId(), OnMouseMove );
root->addChild( eventCB );
```

We’re going to use “picking” to find the node (if any) under the cursor, but in this case we don’t have to explicitly create an *SoRayPickAction*. The *SoHandleEventAction* that delivers events to the scene graph will do that for us when we call the *getPickedPoint()* method. If the pick was successful, then we first get the “path” to the picked

node (using paths allows the scene graph to handle multiple instances of the same node), then get the “tail” (last node in the path), which will be the picked node. If the picked node is an *SoOrthoSlice*, then we get the slice “detail” object and use that object to query the IJK position and value of the picked voxel.

```
// Handler for mouse move events.
void OnMouseMove( void* userData, SoEventCallback* node )
{
    // Get the event object.
    const SoLocation2Event* evt = (SoLocation2Event*)node->getEvent();

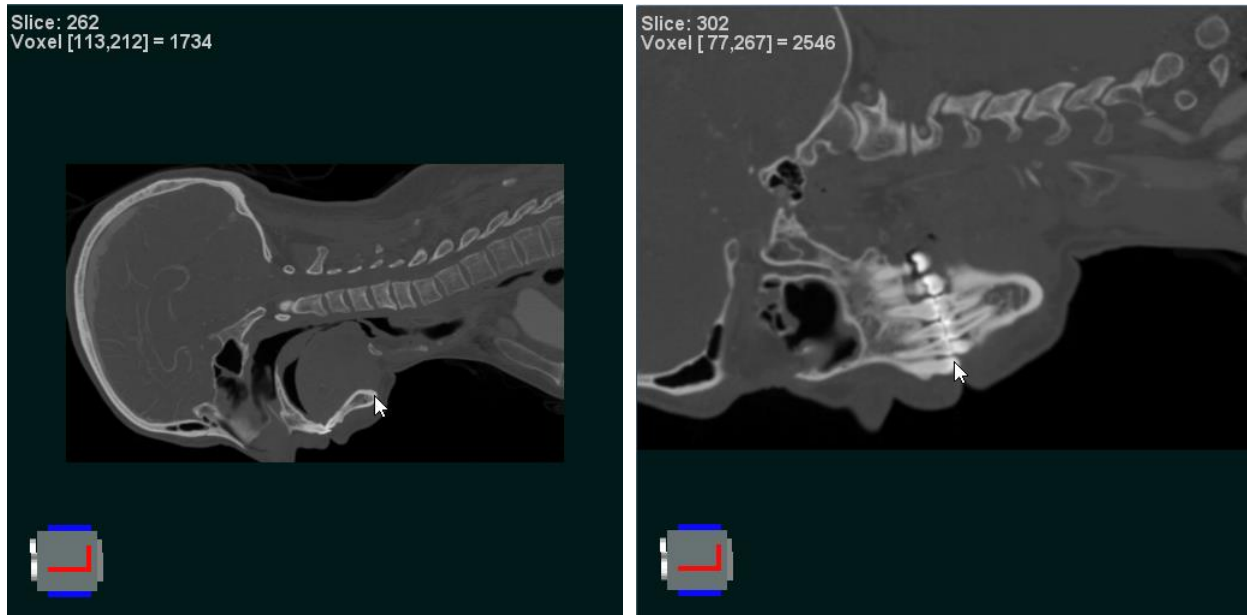
    // Check if any geometry is under the cursor
    SoHandleEventAction* action = node->getAction();
    const SoPickedPoint* pickedPt = action->getPickedPoint();
    if (pickedPt != NULL) {

        // Yes! Check if the node under the cursor is an OrthoSlice
        SoFullPath* pickedPath = (SoFullPath*)pickedPt->getPath();
        SoNode* pickedNode = pickedPath->getTail();
        if (pickedNode->isOfType(SoOrthoSlice::getClassTypeId())) {

            // Yes! Get the slice detail and query the voxel position and value.
            SoOrthoSliceDetail* detail = (SoOrthoSliceDetail*)pickedPt->getDetail();
            const SbVec3i32& ijkPos = detail->getValueDataPos();
            int64_t value = detail->getValue();

        }
    }
}
```

Using the annotation text technique presented in the previous chapter, the result looks something like this:



It should be fairly clear now how to extend this model, for example to add a handler for *SoMouseButtonEvent* to the *SoEventCallback*. By handling mouse button events and mouse move events, we could, for example, allow the user to interactively adjust the Window and Level values (window width and window center) by clicking and dragging the mouse in the Open Inventor window. In that case we would use the mouse move event position to compute new values for the ‘min’ and ‘max’ fields in the *SoDataRange* node.

x X x