Visualizing Image & Volume Data

Image datasets, represented by the class vtkImageData, are regular in topology and geometry as shown in **Figure 6–1**. This data type is structured, meaning that the locations of the data points are implicitly defined using just the few parameters origin, spacing and dimensions. Medical and scientific scanning devices such as CT, MRI, and ultrasound scanners, and confocal microscopes often produce data of this type. Conceptually, the vtkImageData dataset is composed of voxel (vtkVoxel) or pixel (vtkPixel) cells. However, the structured nature of this dataset allows us to store the data values in a simple array rather than explicitly creating the vtkVoxel or vtkPixel cells.

In VTK, image data is a special data type that can be processed and rendered in several ways. Although not an exhaustive classification, most of the operations performed on image data in VTK fall into one of the three categories—image processing, geometry extraction, or direct rendering. Dozens of image processing filters exist that can operate on image datasets. These filters take vtkImageData as input and produce vtkImageData as output. Geometry extraction filters exist that convert vtkImageData into vtkPolyData. For example, the vtkContourFilter can extract iso-valued contours in triangular patches from the image dataset. Finally, there are various mappers and specialized actors to render vtkImageData, including techniques ranging from simple 2D image display to volume rendering.

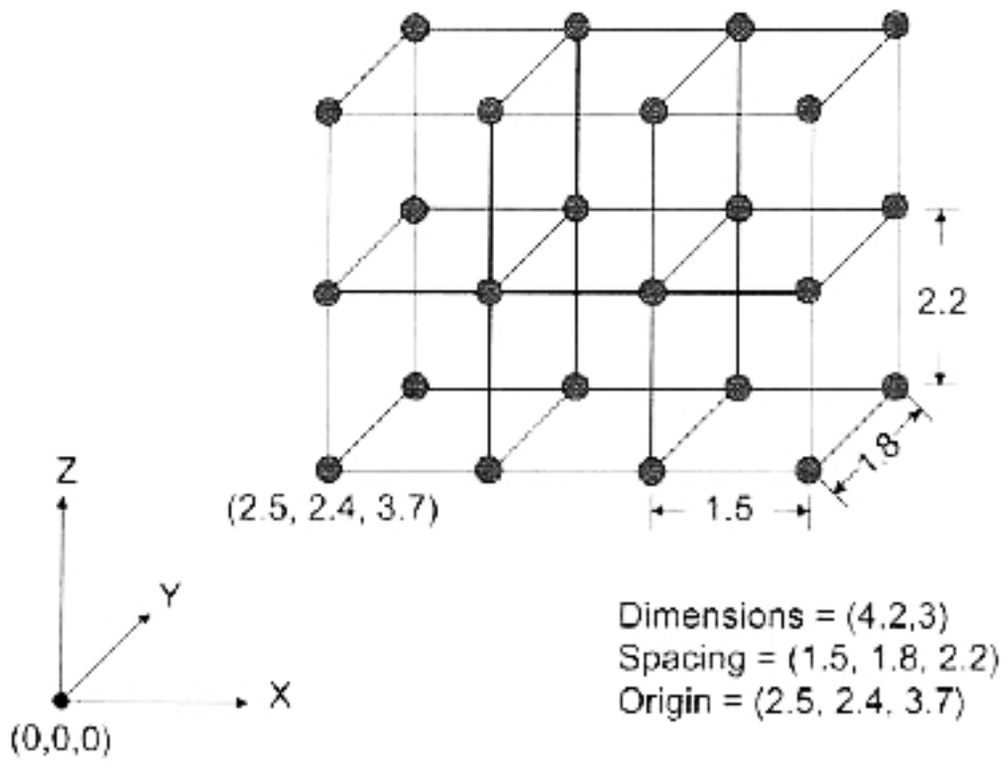


Figure 6–1 The vtkImageData structure is defined by dimensions, spacing, and origin. The dimensions are the number of voxels or pixels along each of the major axes. The origin is the world coordinate position of the lower left corner of the first slice of the data. The spacing is the distance between pixels along each of the three major axes.

In this chapter we examine some important image processing techniques. We will discuss basic image display, image processing, and geometry extraction as elevation maps. In addition, we will cover volume rendering in detail in this chapter. Other geometry extraction techniques such as contouring are covered in Chapter 5.

6.1 Historical Note on vtkStructuredPoints

Initially, VTK contained only the class vtkStructuredPoints for representing image datasets. When extensive image processing functionality was added to VTK, the class vtkImageData was developed. This class was more flexible than vtkStructuredPoints, allowing for parallel processing and streaming of data. Over the next several releases of VTK, the image and graphics pipelines were merged. This caused vtkImageData and vtkStructuredPoints to become functionally quite similar, until vtkStructuredPoints became a nearly empty subclass of vtkImageData. Although it will likely require several more releases of VTK to remove all references to "structured points," the class vtkStructuredPoints has been deprecated and should not be used.

6.2 Manually Creating vtkImageData

Creating image data is straightforward: you need only define the dimensions, origin, and spacing of the dataset. The origin is the world coordinate position of the lower left hand corner of the dataset. The dimensions are the number of voxels or pixels along each of the three major axes. The spacing is the height, length, and width of a voxel or the distance between neighboring pixels, depending on whether you view your data as homogeneous boxes or sample points in a continuous function.

In this first example we will assume that we have an array of unsigned character values pointed to by the variable data, and is size[0] by size[1] by size[2] samples. We generated this data outside of VTK, and now want to get this data into a vtkImageData so that we might use the VTK filtering and rendering operations. We will give VTK a pointer into the memory, but we will manage the deletion of the memory ourselves.

The first thing we need to do is create an array of unsigned chars to store the data. We use the SetVoidArray() method to specify the pointer to the data and its size, with the final argument indicating that VTK should not free this memory.

```
vtkUnsignedCharArray *array = vtkUnsignedCharArray::New();
array->SetVoidArray( data, size[0]*size[1]*size[2], 1);
```

The second step is to create the image data. We must take care that all values match—the scalar type of the image data must be unsigned char, and the dimensions of the image data must match the size of the data.

```
imageData = vtkImageData::New();
imageData->GetPointData()->SetScalars(array);
imageData->SetDimensions(size);
imageData->SetScalarType(VTK_UNSIGNED_CHAR);
imageData->SetSpacing(1.0, 1.0, 1.0 );
imageData->SetOrigin(0.0, 0.0, 0.0 );
```

What's important about image datasets is that because the geometry and topology are implicitly defined by the dimensions, origin, and spacing, the storage required to represent the dataset *structure* is tiny. Also, computation on the structure is fast because of its regular arrangement. What does require storage is the attribute data that goes along with the dataset.

In this next example, we will use C++ to create the image data. Instead of manually creating the data array and associating them with the image data, we will have the vtkImageData object create the scalar data for us. This eliminates the possibility of mismatching the size of the scalars with the dimensions of the image data.

```
// Create the image data
vtkImageData *id = vtkImageData::New();
  id->SetDimensions(10,25,100);
  id->SetScalarTypeToUnsignedShort();
  id->SetNumberOfScalarComponents(1);
  id->AllocateScalars();

// Fill in scalar values
unsigned short *ptr = (unsigned short *) id->GetScalarPointer();
for (int i=0; i<10*25*100; i++)
  {
  *ptr++ = i;
  }
```

In this example, the convenience method AllocateScalars() is used to allocate storage for the image data. Notice that this call is made *after* the scalar type and number of scalar components have been set (up to four scalar components can be set). Then the method GetScalarPointer(), which returns a `void*`, is invoked and the result is cast to `unsigned`

short. We can do this knowing that the type is unsigned short because we specified this earlier. (In VTK's imaging filters, many Execute() methods query the scalar type and then switch on the type into a templated function in a similar manner.)

6.3 Subsampling Image Data

As we saw in "Extract Subset of Cells" on page 102, extracting parts of a dataset is often desirable. The filter vtkExtractVOI extracts pieces of the input image dataset, and can also perform subsampling on it. The output of the filter is also of type vtkImageData.

There are actually two similar filters that perform this clipping functionality in VTK: vtkExtractVOI and vtkImageClip. The reason that there are two versions is historical—the imaging pipeline used to be separate from the graphics pipeline, with vtkImageClip working only on vtkImageData in the imaging pipeline and vtkExtractVOI working only on vtkStructuredPoints in the graphics pipeline. These distinctions are gone now, but there are still some differences between these filters. vtkExtractVOI will extract a subregion of the volume and produce a vtkImageData that contains exactly this information. In addition, vtkExtractVOI can be used to resample the volume within the VOI. On the other hand, vtkImageClip by default will pass the input data through to the output unchanged except for the extent information. A flag may be set on this filter to force it to produce the exact amount of data only, in which case the region will be copied into the output vtkImageData. The vtkImageClip filter cannot resample the volume.

The following Tcl example (taken from VTK/Examples/ImageProcessing/Tcl/Contours2D.tcl) demonstrates how to use vtkExtractVOI. It extracts a piece of the input volume, and then subsamples it. The output is fed into a vtkContourFilter. (You may want to try removing vtkExtractVOI and compare the results.)

```
# Quadric definition
vtkQuadric quadric
    quadric SetCoefficients .5 1 .2 0 .1 0 0 .2 0 0
vtkSampleFunction sample
    sample SetSampleDimensions 30 30 30
    sample SetImplicitFunction quadric
    sample ComputeNormalsOff
vtkExtractVOI extract
    extract SetInput [sample GetOutput]
    extract SetVOI 0 29 0 29 15 15
    extract SetSampleRate 1 2 3
vtkContourFilter contours
```

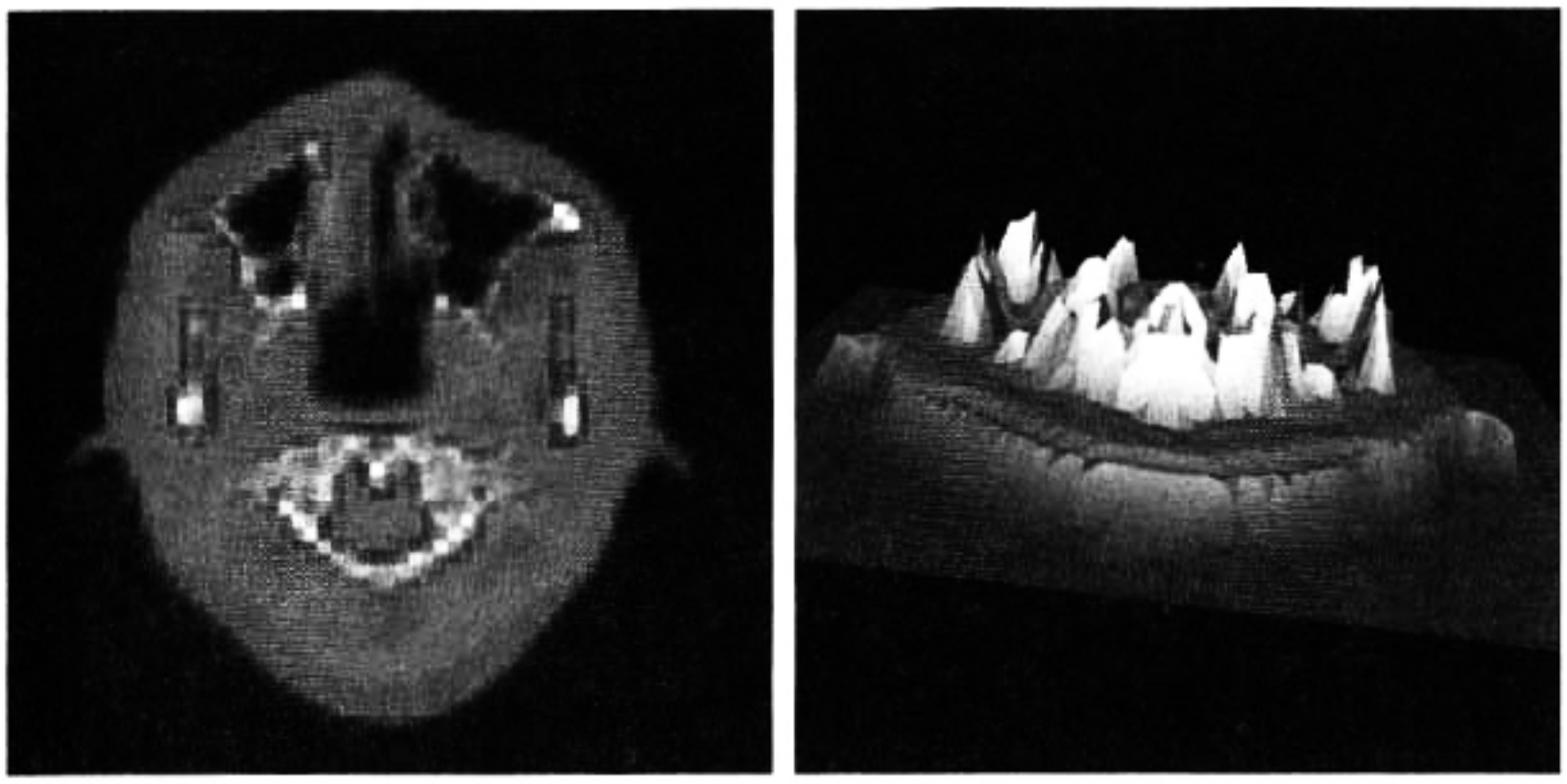


Figure 6–2 Image warped by scalar values.

```
    contours SetInput [extract GetOutput]
    contours GenerateValues 13 0.0 1.2
vtkPolyDataMapper contMapper
    contMapper SetInput [contours GetOutput]
    contMapper SetScalarRange 0.0 1.2
vtkActor contActor
    contActor SetMapper contMapper
```

Note that this script extracts a plane from the original data by specifying the volume of interest (VOI) as (0,29,0,29,15,15) (i_{min},i_{max}, j_{min},j_{max}, k_{min},k_{max}), and the sample rate is set differently along each of the *i*-*j*-*k* topological axes. You could also extract a subvolume or even a line or point by modifying the VOI specification. (The volume of interest is specified using 0-offset values.)

6.4 Warp Based On Scalar Values

One common use of image data is to store elevation values as an image. These images are frequently called range maps or elevation maps. The scalar value for each pixel in the image represents an elevation, or range value. A common task in visualization is to take such an image and warp it to produce an accurate 3D geometry representing the elevation or range data. Consider **Figure 6–2** which shows an image that has been warped based on its scalar value. The left image shows the original image while the right view shows the image after warping to produce a 3D surface.

The pipeline to perform this visualization is fairly simple but there is an important concept to understand. The original data is an image which has implicit geometry and topology. Warping the image will result in a 3D surface where geometry is no longer implicit. To support this we first convert the image to a vtkPolyData representation using vtkImage-DataGeometryFilter. The we perform the warp and connect to a mapper. In the script below you'll note that we also make use of vtkWindowLevelLookupTable to provide a greyscale lookup-table instead of the default red to blue lookup table.

```
vtkImageReader reader
reader SetDataByteOrderToLittleEndian
reader SetDataExtent 0 63 0 63 40 40
reader SetFilePrefix "$VTK_DATA_ROOT/Data/headsq/quarter"
reader SetDataMask 0x7fff

vtkImageDataGeometryFilter geometry
  geometry SetInput [reader GetOutput]

vtkWarpScalar warp
  warp SetInput [geometry GetOutput]
  warp SetScaleFactor 0.005

vtkWindowLevelLookupTable wl

vtkPolyDataMapper mapper
  mapper SetInput [warp GetPolyDataOutput]
  mapper SetScalarRange 0 2000
  mapper ImmediateModeRenderingOff
  mapper SetLookupTable wl

vtkActor actor
  actor SetMapper mapper
```

This example is often combined with other techniques. If you want to warp the image with its scalar value and then color it with a different scalar field you would use the vtkMerge-Filter. Another common operation is to reduce the number of polygons in the warped surface. Because these surfaces were generated from images they tend to have a large number of polygons. You can use vtkDecimatePro to reduce the number. You should also consider using vtkTriangleFilter followed by vtkStripper to convert the polygons (squares) into triangle strips which tend to render faster and consume less memory.

6.5 Image Display

There are several ways to directly display image data. Two methods that are generally applicable for displaying 2D images are described in this section. Volume rendering is the method for directly displaying 3D images (volumes) and is described in detail later in this chapter.

Image Viewer

vtkImageViewer is a convenient class for displaying images. It manages several objects internally—vtkImageWindow, vtkImager, vtkActor2D, and vtkImageMapper—providing an easy-to-user class that can be dropped into your application. A typical usage pattern for vtkImageViewer is to set its input, specify a *z*-slice to display (assuming that the input is a volume), and then specify window and level transfer function values. (Refer to "Image Reslice" on page 135 for an example of this class used in application.)

```
vtkImageViewer viewer
  viewer SetInput [reslice GetOutput]
  viewer SetZSlice 120
  viewer SetColorWindow 2000
  viewer SetColorLevel 1000
```

The window-level transfer function is defined as shown in **Figure 6–3**. The level is the data value that centers the a window. The slope of the resulting transfer function determines the amount of contrast in the final image. The width (i.e., window) defines the data values that are mapped to the display. All data values outside of the window are clamped to the data values at the boundaries of the window.

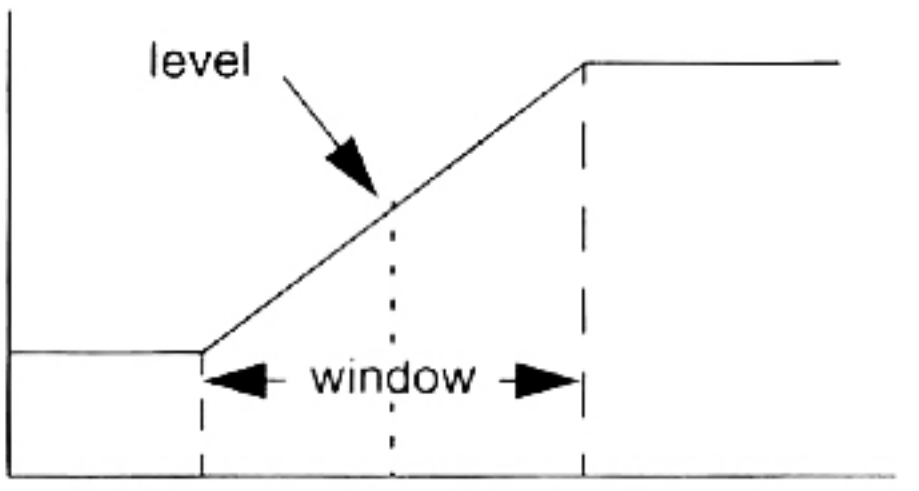


Figure 6–3 Window-level transfer function.

Image Actor

Using a vtkImageViewer is convenient when you would simply like to display the image in a window by itself, or accompanied by some simple 2D annotation. The vtkImageActor composite actor class is useful when you want to display your image in a 3D rendering

window. The image is displayed by creating a polygon representing the bounds of the image and using hardware texture mapping to paste the image onto the polygon. On most platforms this enables you to rotate, pan, and zoom your image with bilinear interpolation in real-time. By changing the interactor to a vtkInteractorStyleImage you can limit rotations so that the 3D render window operates as a 2D image viewer (see "Interactor Styles" on page 50 and "vtkRenderWindow Interaction Style" on page 303 for more information about interactor styles). The advantage to using the 3D render window for image display is that you can easily embed multiple images and complex 3D annotation into one window.

The vtkImageActor object is a composite class that encapsulates both an actor and a mapper into one class. It is simple to use, as can be seen in this example.

```
vtkPNMReader pnmReader
pnmReader SetFileName "$VTK_DATA_ROOT/Data/masonry.ppm"

vtkImageActor imageActor
imageActor SetInput [pnmReader GetOutput]
```

This imageActor can then be added to the renderer using the AddProp() method. The vtkImageActor class expects that its input will have a length of 1 along one of the three dimensions, with the image extending along the other two dimensions. This allows the vtkImageActor to be connected to a volume through the use of a clipping filter without the need to reorganize the data if the clip is performed along the X or Y axis. (Note: the input image to vtkImageActor must be if type unsigned char. If your image type is different, you can use vtkImageCast or vtkImageShiftScale to convert to unsigned char.)

6.6 Image Sources

There are some image processing objects that produce output but do not take any data objects as input. These are known as image sources, and some of the VTK image sources are described here. Refer to "Source Objects" on page 332, or the Doxygen documentation on the CD, for a more complete list of available image sources.

ImageCanvasSource2D

The vtkImageCanvasSource2D class creates a blank two-dimensional image of a specified size and type, and provides methods for drawing various primitives into this blank image. Primitives include boxes, lines, and circles, and a flood fill operation is also provided. The

following example illustrates the use of this source by creating a 512x512 pixel images and drawing several primitives into this image. The resulting image is shown in **Figure 6–4**.

```
#set up the size and type of the image canvas
vtkImageCanvasSource2D imCan
imCan SetScalarType $VTK_UNSIGNED_CHAR
imCan SetExtent 0 511 0 511 0 0

# Draw various primitives
imCan SetDrawColor 86
imCan FillBox 0 511 0 511
imCan SetDrawColor 0
imCan FillTube 500 20 30 400 5
imCan SetDrawColor 255
imCan DrawSegment 10 20 500 510
imCan SetDrawColor 0
imCan DrawCircle 400 350 80.0
imCan SetDrawColor 255
imCan FillPixel 450 350
imCan SetDrawColor 170
imCan FillTriangle 100 100  300 150  150 300

#Show the resulting image
vtkImageViewer viewer
viewer SetInput [imCan GetOutput]
viewer SetColorWindow 256
viewer SetColorLevel 127.5
```

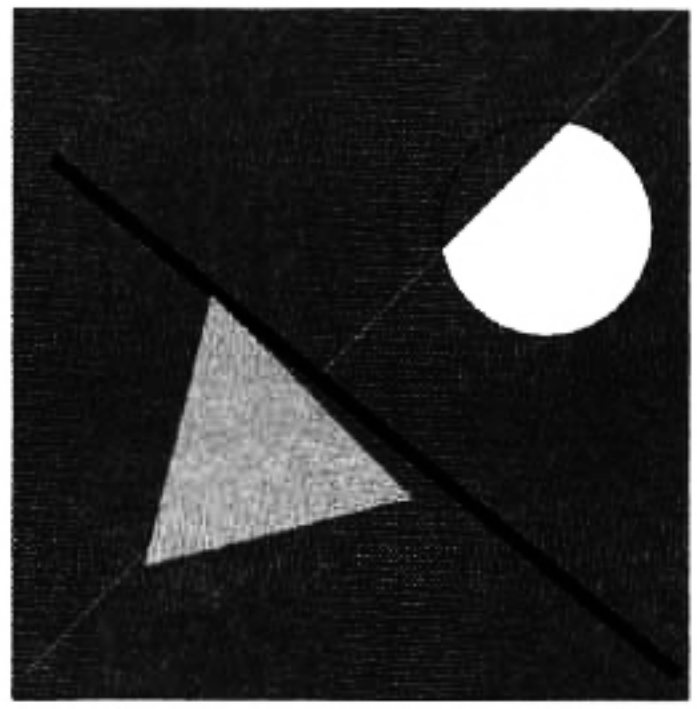


Figure 6–4 The results from a vtkImageCanvasSource2D source after drawing various primitives.

ImageEllipsoidSource

If you would like to write your own image source using a templated execute function, vtkImageEllipsoidSource is a good starting point. This object produces a binary image of an ellipsoid as output based on a center position, a radius along each axis, and the inside and outside values. The output scalar type can also be specified, and this is why the execute function is templated. This source is used internally by some of the imaging filters such as vtkImageDilateErode3D.

If you want to create a vtkImageBoxSource, for example, to produce a binary image of a box you could start by copying the vtkImageEllipsoidSource source and header files and doing a global search and replace. You would probably change the instance variable

Radius to be Length since this is a more appropriate description for a box source. Finally, you would replace the code within the templated function vtkImageBoxSourceExecute to create the box image rather than the ellipsoid image. (For more information on creating image processing filters see "How To Write An Imaging Filter" on page 295.)

ImageGaussianSource

The vtkImageGaussianSource object produces an image with pixel values determined according to a Gaussian distribution using a center location, a maximum value, and a standard deviation. The output of this image source is always floating point values.

If you would like to write your own source that produces just one type of output image, for example float, than this might be a good class to use as a starting point. Comparing the source code for vtkImageGaussianSource with that for vtkImageEllipsoidSource, you will notice that the filter implementation is in the Execute() method for vtkImageGaussianSource, whereas in vtkImageEllipsoidSource the Execute() method calls a templated function that contains the implementation.

ImageGridSource

If you would like to annotate your image with a 2D grid, vtkImageGridSource can be used to create an image with the grid pattern (**Figure 6–5**). The following example illustrates this use by blending a grid pattern with a slice from a CT dataset. The reader is a vtkImageReader that produces a 64 by 64 image.

```
imageGrid SetGridSpacing 16 16 0
imageGrid SetGridOrigin 0 0 0
imageGrid SetDataExtent 0 63 0 63 0 0
imageGrid SetLineValue 4095
imageGrid SetFillValue 0
imageGrid SetDataScalarTypeToShort

vtkImageBlend blend
blend SetOpacity 0 0.5
blend SetOpacity 1 0.5
blend SetInput 0 [reader GetOutput]
blend SetInput 1 [imageGrid GetOutput]
```

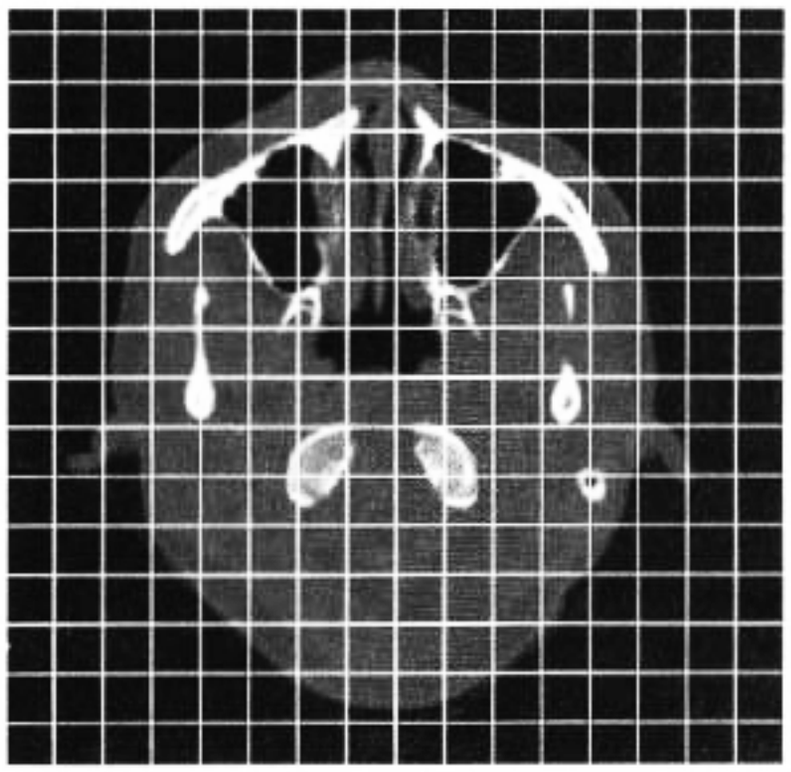


Figure 6–5 A grid pattern created by a vtkImageGridSource is overlayed on a slice of a CT dataset.

```
vtkImageViewer viewer
viewer SetInput [blend GetOutput]
viewer SetColorWindow 1000
viewer SetColorLevel 500
viewer Render
```

ImageNoiseSource

The vtkImageNoiseSource image source can be used to generate an image filled with random numbers between some specified minimum and maximum values. The type of the output image is floating point.

One thing to note about vtkImageNoiseSource is that it will produce a different image every time it executes. Normally, this is the desired behavior of a noise source, but this has negative implications in a streaming pipeline with overlap in that the overlapping region will not have the same values across the two requests. For example, assume you set up a pipeline with a vtkImageNoiseSource connected to an ImageMedianFilter which is in turn connected to a vtkImageDataStreamer. If you specify a memory limit in the streamer such that the image will be computed in two halves, the first request the streamer makes would be for half the image. The median filter would need slightly more than half of the input image (based on the extent of the kernel) to produce requested output image. When the median filter executes the second time to produce the second half of the output image, it will again request the overlap region, but this region will contain different values, causing any values computed using the overlap region to be inaccurate.

ImageSinusoidSource

The vtkImageSinusoidSource object can be used to create an image of a specified size where the pixel values are determined by a sinusoid function given direction, period, phase, and amplitude values. The output of the sinusoid source is floating point. In the image below, the output of the sinusoid source shown on the left has been converted to unsigned char

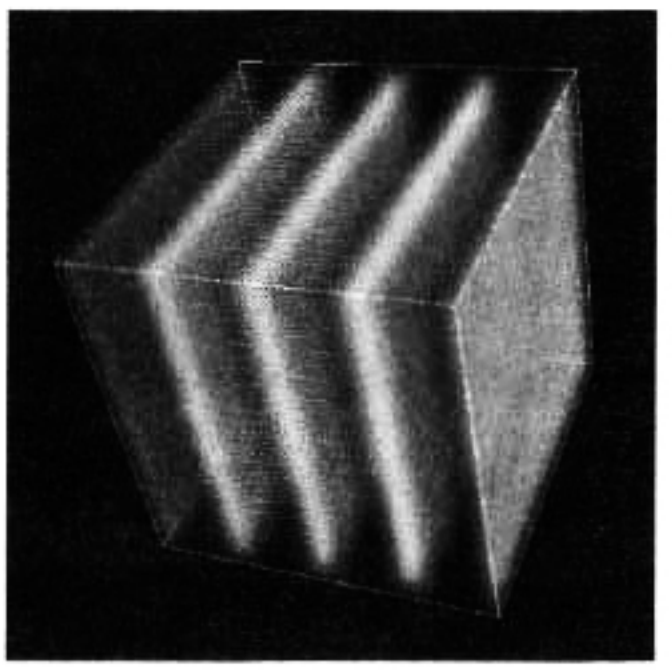


Figure 6–6 The output of the sinusoid source shown on the left has been converted to unsigned char and volume rendered.

values and volume rendered. This same output was passed through an outline filter to create the bounding box seen in the image.

```
vtkImageSinusoidSource ss
  ss SetWholeExtent 0 99 0 99 0 99
  ss SetAmplitude 63
  ss SetDirection 1 0 0
  ss SetPeriod 25
```

6.7 Image Processing

Now we will consider a few examples that process image data. This is not an exhaustive description of all filters, but it will get you started using VTK's image processing filters. You may wish to refer to the Doxygen documentation on CD, and image processing texts for more information.

Gradient

vtkImageGradient is the filter that computes the gradient of an image or volume. You can control whether it computes a two- or three-dimensional gradient using the SetDimensionality() method. It will produce an output with either two or three scalar components per pixel depending on the dimensionality you specified. The scalar components correspond to the x, y, and optionally z components of the gradient vector. If you only want the gradient magnitude you can use the vtkImageGradientMagnitude filter or vtkImageGradient followed by vtkImageMagnitude.

vtkImageGradient computes the gradient by using central differences. This means that to compute the gradient for a pixel we must look at its left and right neighbors. This creates a problem for the pixels on the outside edges of the image since they will be missing one of their two neighbors. There are two solutions to this problem and they are controlled by the HandleBoundaries instance variable. If HandleBoundaries is on, then vtkImageGradient will use a modified gradient calculation for all of the edge pixels. If HandleBoundaries is off, vtkImageGradient will ignore those edge pixels and produce a resulting image that is smaller than the original input image.

In the following example we use a vtkImageReader to read in the raw medical data for a CT scan. We then pass this data through a 3D vtkImageGradient and display the result as a color image.

```
vtkImageReader reader
  reader SetDataByteOrderToLittleEndian
  reader SetDataExtent 0 63 0 63 1 93
  reader SetFilePrefix "$VTK_DATA_ROOT/Data/headsq/quarter"
  reader SetDataMask 0x7fff

vtkImageGradient gradient
  gradient SetInput [reader GetOutput]
  gradient SetDimensionality 3

vtkImageViewer viewer
  viewer SetInput [gradient GetOutput]
  viewer SetZSlice 22
  viewer SetColorWindow 400
  viewer SetColorLevel 0
```

Gaussian Smoothing

Smoothing an image with a Gaussian kernel is similar to the gradient calculation done above. It has a dimensionality that controls what dimension Gaussian kernel to convolve against. It also has SetStandardDeviations() and SetRadiusFactors() methods that control the shape of the Gaussian kernel and when to truncate it. The example provided below is very similar to the gradient calculation. We start with a vtkImageReader connected to the vtkImageGaussianSmooth which finally connects to the vtkImageViewer.

```
vtkImageReader reader
  reader SetDataByteOrderToLittleEndian
  reader SetDataExtent 0 63 0 63 1 93
  reader SetFilePrefix "$VTK_DATA_ROOT/Data/headsq/quarter"
  reader SetDataMask 0x7fff

vtkImageGaussianSmooth smooth
  smooth SetInput [reader GetOutput]
  smooth SetDimensionality 2
  smooth SetStandardDeviations 2 10

VtkImageViewer viewer
  viewer SetInput [smooth GetOutput]
  viewer SetZSlice 22
  viewer SetColorWindow 2000
  viewer SetColorLevel 1000
```

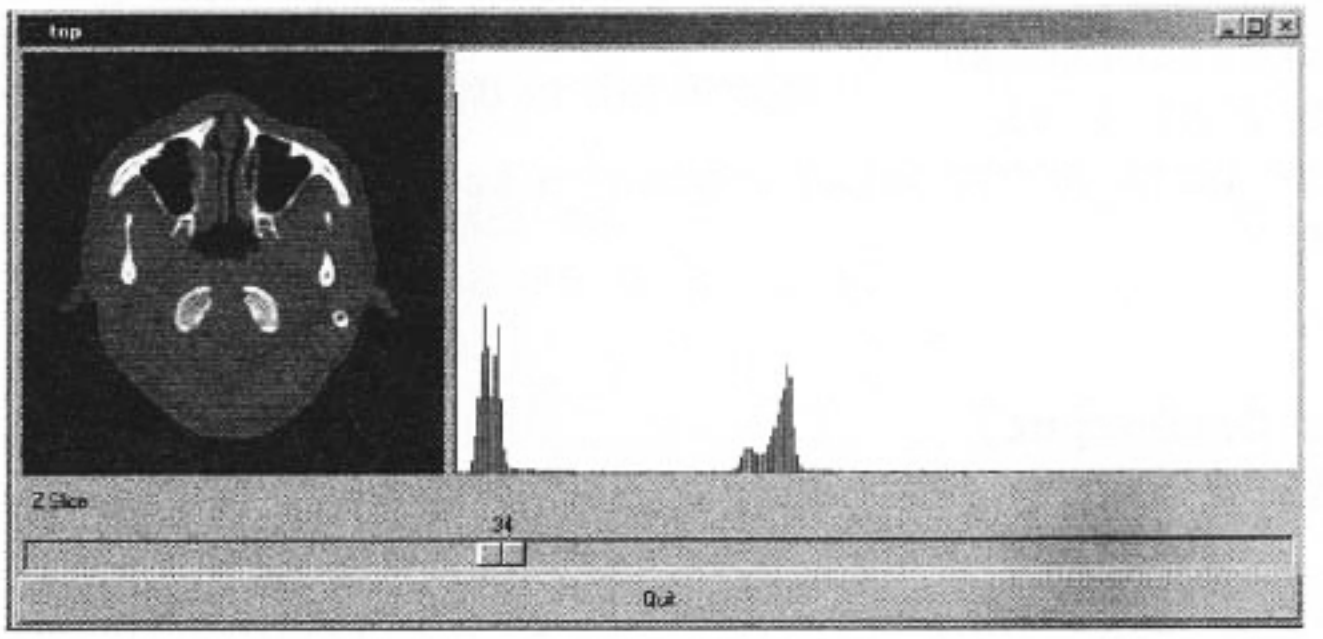


Figure 6–7 The vtkImageAccumulate class is used to generate the one dimensional histogram from the one-component input image.

Histogram

vtkImageAccumulate is an image filter that will produce generalized histograms of up to four dimensions. This is done by dividing the component space into discrete bins, then counting the number of pixels corresponding to each bin. The input image may be of any scalar type, but the output image will always be of integer type. If the input image has only one scalar component, then the output image will be one-dimensional, as shown in **Figure 6–7**. (This example is taken from `VTK/Examples/ImageProcessing/Tcl/Histogram.tcl`.)

Image Logic

vtkImageLogic is an image processing filter that takes one or two inputs and performs a boolean logic operation on them (**Figure 6–8**). Most standard operations are supported including AND, OR, XOR, NAND, NOR, and NOT. This filter has two inputs although for unary operations such as NOT, only the first input is required. In the example provided below you will notice we use vtkImageEllipseSource to generate the two input images. vtkImageEllipseSource is one of a few image sources that can procedurally generate images. Similar classes are vtkImageGaussianSource and vtkImageNoiseSource.

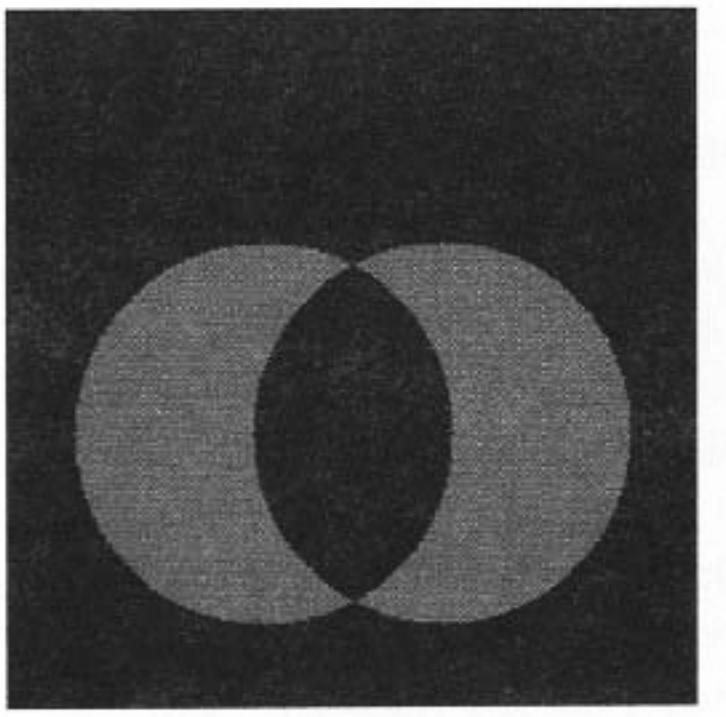


Figure 6–8 Result of image logic.

```
vtkImageEllipsoidSource sphere1
  sphere1 SetCenter 95 100 0
  sphere1 SetRadius 70 70 70

vtkImageEllipsoidSource sphere2
  sphere2 SetCenter 161 100 0
  sphere2 SetRadius 70 70 70

vtkImageLogic xor
  xor SetInput1 [sphere1 GetOutput]
  xor SetInput2 [sphere2 GetOutput]
  xor SetOutputTrueValue 150
  xor SetOperationToXor

vtkImageViewer viewer
  viewer SetInput [xor GetOutput]
  viewer SetColorWindow 255
  viewer SetColorLevel 127.5
```

Image Reslice

vtkImageReslice is a contributed class (David Gobbi is the author) that offers high-performance image resampling along an arbitrarily oriented volume (or image). The extent, origin, and sampling density of the output data can also be set. This class provides several other imaging filters: it can permute, flip, rotate, scale, resample, and pad image data in any combination. It can also extract oblique slices from image volumes, which no other VTK imaging filter can do. The following script, taken from `VTK/Examples/Imaging/Tcl/TestReslice.tcl` demonstrates how to use vtkImageReslice.

```
vtkImageReader reader
  reader ReleaseDataFlagOff
  reader SetDataByteOrderToLittleEndian
  reader SetDataExtent 0 63 0 63 1 93
  reader SetDataOrigin -32.5 -32.5 -47
  reader SetFilePrefix "$VTK_DATA_ROOT/Data/headsq/quarter"
  reader SetDataMask 0x7fff
  reader Update
vtkTransform transform
  transform RotateX 10
  transform RotateY 20
  transform RotateZ 30
vtkImageReslice reslice
  reslice SetInput [reader GetOutput]
```

```
  reslice SetResliceTransform transform
  reslice InterpolateOn
  reslice SetBackgroundLevel 1023
vtkImageViewer viewer
  viewer SetInput [reslice GetOutput]
  viewer SetZSlice 120
  viewer SetColorWindow 2000
  viewer SetColorLevel 1000
  viewer Render
```

In this example (**Figure 6–9**) a volume of size 64^2 x 93 is read. A transform is used to position a volume on which to resample (or reslice) the data. Linear interpolation is used, and a background gray level (since the volume is single-component) is set that defines the color of resample points outside the original volume. By default, the spacing of the output volume is set at 1.0, and the output origin and extent are adjusted to enclose the input volume. A viewer is used to display one *z*-slice of the resulting volume.

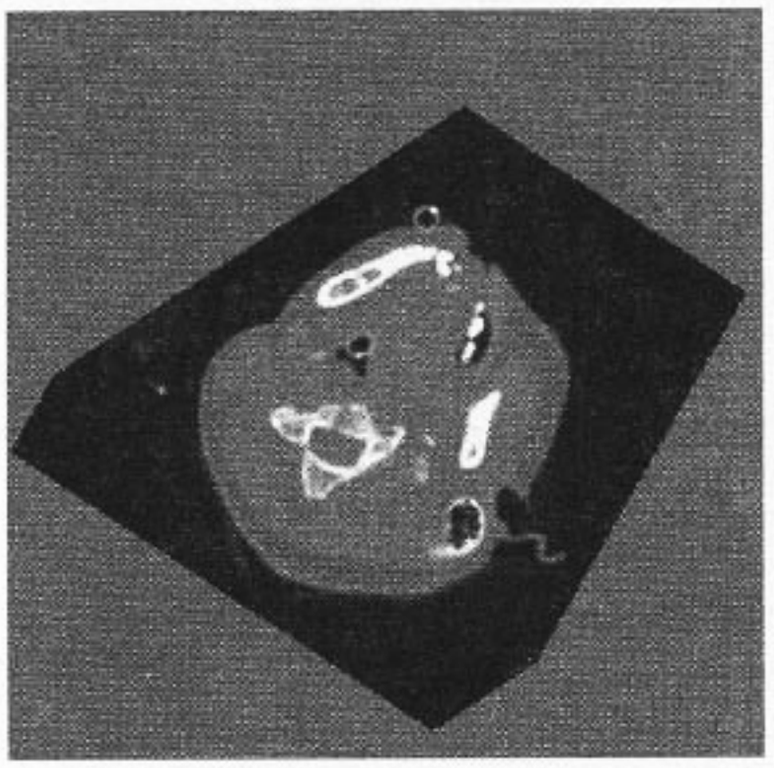


Figure 6–9 Output of vtkImageReslice with a gray background level set.

6.8 Volume Rendering

Volume rendering is a term used to describe a rendering process applied to 3D data where information exists throughout a 3D space instead of simply on 2D surfaces defined in 3D space. There is not a clear dividing line between volume rendering and geometric rendering techniques. Often two different approaches can produce similar results, and in some cases one approach may be considered both a volume rendering and a geometric rendering technique. For example, you can use a contouring technique to extract triangles representing an isosurface in a structured points dataset (see "Contouring" on page 89) and then use geometric rendering techniques to display these triangles, or you can use a volumetric ray casting technique on the image dataset and terminate the ray traversal at a particular isovalue. These two different approaches produce similar (although not necessarily identical) results. Another example is the technique of employing texture mapping hardware in order to perform composite volume rendering. This one method maybe considered a volume rendering technique since it works on image data, or a geometric technique since it uses geometric primitives and standard graphics hardware.

In VTK, volume rendering techniques have been implemented for image datasets. The SetInput() method of vtkVolumeMapper accepts a pointer to vtkImageData. Unstructured datasets cannot (at this time) be directly volume rendered. However, you can resample an unstructured grid into a regular image data format in order to take advantage of the techniques described in this section (see "Probing" on page 98).

There are three main volume rendering techniques currently implemented in VTK: ray casting, 2D texture mapping, and support for the VolumePro volume rendering hardware (if you have one of these boards installed on your system). More information on the VolumePro board can be found at the TeraRecon web site: `http://www.terarecon.com`.

We will begin this section with a simple example written using the three different volume rendering techniques. Then we will cover the objects / parameters common to all of these techniques. Next, each of the three volume rendering techniques will be discussed in more detail, including information on parameters specific to that rendering method. This will be followed by a discussion on achieving interactive rendering rates, and finally a summary of the features supported by each volume rendering approach.

A Simple Example

Consider the simple volume rendering example shown below (and illustrated in **Figure 6–10**) that is nearly the simplest Tcl script that can be written to render a volume (refer to `VTK/Examples/VolumeRendering/Tcl/SimpleRayCast.tcl`). This example is written for volumetric ray casting, but only the portion of the Tcl script highlighted with bold text is specific to this rendering technique. Following this example you will find the alternate versions of the bold portion of the script that would instead perform the volume rendering task with a 2D texture mapping approach, or using the VolumePro hardware support. You will notice that switching volume rendering techniques, at least in this simple case, requires only a few minor changes to the script.

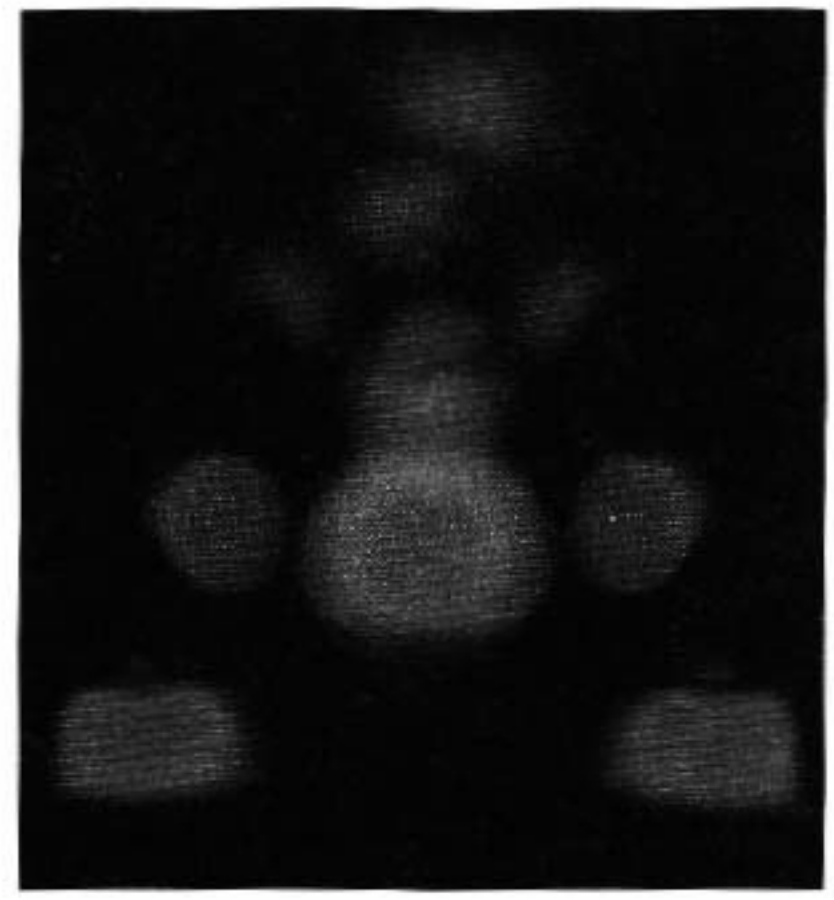


Figure 6–10 Volume rendering.

```
vtkRenderer ren1
vtkRenderWindow renWin
    renWin AddRenderer ren1
vtkRenderWindowInteractor iren
    iren SetRenderWindow renWin

# Create the reader for the data
vtkStructuredPointsReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/ironProt.vtk"

# Create transfer mapping scalar value to opacity
vtkPiecewiseFunction opacityTransferFunction
    opacityTransferFunction AddPoint  20   0.0
    opacityTransferFunction AddPoint  255  0.2

# Create transfer mapping scalar value to color
vtkColorTransferFunction colorTransferFunction
    colorTransferFunction AddRGBPoint      0.0 0.0 0.0 0.0
    colorTransferFunction AddRGBPoint     64.0 1.0 0.0 0.0
    colorTransferFunction AddRGBPoint    128.0 0.0 0.0 1.0
    colorTransferFunction AddRGBPoint    192.0 0.0 1.0 0.0
    colorTransferFunction AddRGBPoint    255.0 0.0 0.2 0.0

# The property describes how the data will look
vtkVolumeProperty volumeProperty
    volumeProperty SetColor colorTransferFunction
    volumeProperty SetScalarOpacity opacityTransferFunction

# The mapper / ray cast functions know how to render the data
vtkVolumeRayCastCompositeFunction  compositeFunction
vtkVolumeRayCastMapper volumeMapper
    volumeMapper SetVolumeRayCastFunction compositeFunction
    volumeMapper SetInput [reader GetOutput]

# The volume holds the mapper and the property and
# can be used to position/orient the volume
vtkVolume volume
    volume SetMapper volumeMapper
    volume SetProperty volumeProperty

ren1 AddProp volume
renWin Render
```

In this example we start by reading in a data file from disk. We then define the functions that map scalar value into opacity and color which are used in the vtkVolumeProperty.

Next we create the objects specific to volumetric ray casting—a vtkVolumeRayCastCompositeFunction that performs the compositing of samples along the ray, and a vtkVolumeRayCastMapper that performs some of the basic ray casting operations such as transformations and clipping. We set the input of the mapper to the data we read off the disk, and we create a vtkVolume (a subclass of vtkProp3D similar to vtkActor) to hold the mapper and property. Finally, we create the standard graphics display objects, add the volume to the renderer, and render the scene.

If you decided to implement the above script with a 2D texture mapping approach instead of volumetric ray casting, the bolded portion of the script would instead be:

```
# Create the objects specific to 2D texture mapping approach
vtkVolumeTextureMapper2D volumeMapper
```

If you have a VolumePro volume rendering board on your system, and you have compiled in the support for this device as described later in this section, you could use the following script fragment to replace the volumetric ray casting specific section with a VolumePro volume mapper:

```
# Create the objects specific to VolumePro volume rendering hardware
vtkVolumeProMapper volumeMapper
```

Why Multiple Volume Rendering Techniques?

As you can see, in this simple example the only thing that changes between rendering strategies is the type of volume mapper that is instantiated; and in the case of ray casting, a ray cast function also needs to be created. This may lead you to the following questions: why are there different volume rendering strategies in VTK? Why can't VTK simply pick the strategy that would work best on my platform? First, it is not always easy to predict which strategy will work best—ray casting may out-perform texture mapping if the image size is reduced, more processors become available, or the graphics hardware is the bottleneck to the rendering rate. These are things that can change continuously at run time. Second, due to its computational complexity, most volume rendering techniques only produce an approximation of the desired rendering equation. For example, techniques which take samples through the volume and composite them with an alpha blending function are only approximating the true integral through the volume. Under different circumstances, different techniques perform better or worse than others in terms of both quality and speed. In addition, some techniques work only under certain special conditions. For example, the VolumePro rendering board only supports parallel camera projections while ray casting

and texture mapping both support parallel and perspective camera transformations. The last portion of this section provides a summary chart of the capabilities / limitations of each of these volume rendering techniques (see **Table 6–1**).

Creating a vtkVolume

A vtkVolume is a subclass of vtkProp3D that is intended for use in volume rendering. Similar to a vtkActor (that is intended for geometric rendering), a vtkVolume holds the transformation information such as position, orientation, and scale, and pointers to the mapper and property for the volume. Additional information on how to control the transformation of a vtkVolume is covered in "Volumes" on page 63.

The vtkVolume class accepts objects that are subclasses of vtkVolumeMapper as input to SetMapper(), and accepts a vtkVolumeProperty object as input to SetProperty(). vtkActor and vtkVolume are two separate objects in order to enforce the different types of the mappers and properties. These different types are necessary due to the fact that some parameters of geometric rendering do not make sense in volume rendering and vice versa. For example, the SetRepresentationToWireframe() method of vtkProperty is meaningless in volume rendering, while the SetInterpolationTypeToNearest() method of vtkVolumeProperty has no value in geometric rendering.

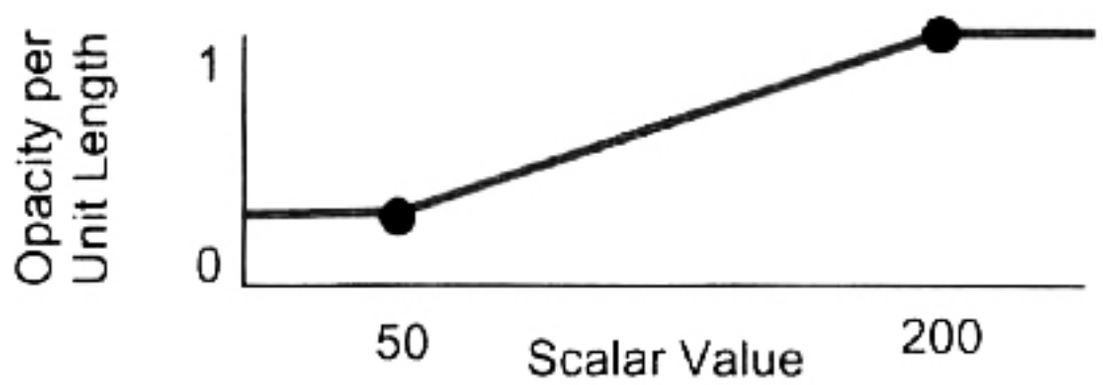
Using vtkPiecewiseFunction

In order to control the appearance of a 3D volume of scalar values, three mappings or transfer functions must be defined. The first transfer function, known as the *scalar opacity transfer function*, maps the scalar value into an opacity or an opacity per unit length value. The second transfer function, referred to simply as the *color transfer function*, maps the scalar value into a color. The third transfer function, called the *gradient opacity transfer function*, maps the magnitude of the gradient of the scalar value into an opacity multiplier. Any of these mappings can be defined as a single value to single value mapping, which can be represented with a vtkPiecewiseTransferFunction. For the scalar value to color mapping, a vtkColorTransferFunction can also be used to define RGB rather than grey-scale colors.

From a user's point of view, vtkPiecewiseFunction has two types of methods—those that add information to the mapping, and those that clear out information from the mapping. When information is added to a mapping, it is considered to be a point sample of the map-

ping with linear interpolation used to determine values between the specified ones. For example, consider the following section of a script on the left that produces the transfer function draw on the right:

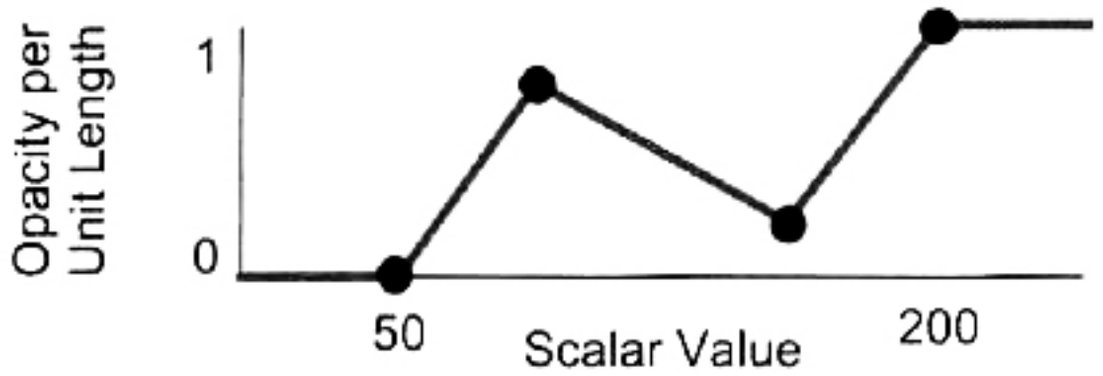
```
vtkPiecewiseFunction tfun
  tfun AddPoint  50  0.2
  tfun AddPoint 200  1.0
```



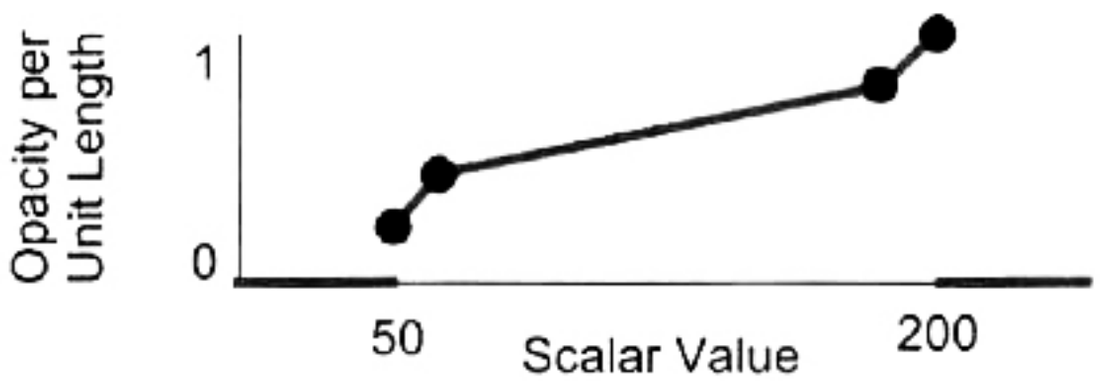
The value of the mapping for the scalar values of 50 and 200 are given as 0.2 and 1.0 respectively, and all other mapping values can be obtained by interpolating between these two values. If Clamping is on (it is by default) then the mapping of any value below 50 will be 0.2, and the mapping of any value above 200 will be 1.0. If Clamping was turned off, then out-of-range values map to 0.0.

Points can be added to the mapping at any time. If a mapping is redefined it replaces the existing mapping. In addition to adding a single point, a segment can be added which will define two mapping points and clear any existing points between the two. As an example, consider the following two modification steps and the corresponding pictorial representations of the transfer functions:

```
tfun RemovePoint  50
tfun AddPoint  50  0.0
tfun AddSegment
     100  0.8  150  0.2
```




```
tfun AddPoint  50  0.2
tfun AddSegment
     60  0.4  190  0.8
tfun ClampingOff
```



In the first step, we change the mapping of scalar value 50 by removing the point and then adding it again, and we add a segment. In the second step, we change the mapping of scalar value 50 by simply adding a new mapping without first removing the old one. We also

add a new segment which eliminates the mappings for 100 and 150 since they lie within the new segment, and we turn clamping off.

Using vtkColorTransferFunction

A vtkColorTransferFunction can be used to specify a mapping of scalar value to color using either an RGB or HSV color space. The methods available are similar to those provided by vtkPiecewiseFunction, but tend to come in two flavors. For example, AddRGBPoint() and AddHSVPoint() both add a point into the transfer function with one accepting an RGB value as input and the other accepting an HSV value as input.

The following Tcl example shows how to specify a transfer function from red to green to blue with RGB interpolation performed for values in between those specified:

```
vtkColorTransferFunction ctfun
  ctfun SetColorSpaceToRGB
  ctfun AddRGBPoint 0 1 0 0
  ctfun AddRGBPoint 127 0 1 0
  ctfun AddRGBPoint 255 0 0 1
```

Controlling Color / Opacity with a vtkVolumeProperty

In the previous two sections we have discussed the basics of creating transfer functions, but we have not yet discussed how these control the appearance of the volume. Typically, defining the transfer functions is the hardest part of achieving an effective volume visualization since you are essentially performing a classification operation that requires you to understand the meaning of the underlying data values.

For rendering techniques that map a pixel to a single location in the volume (such as an isosurface rendering or a maximum intensity projection) the ScalarOpacity transfer function maps the scalar value to an opacity. When a compositing technique is used, the ScalarOpacity function maps scalar value to an opacity that is accumulated per unit length for a homogenous region of that value. Compositing is performed by approximating the continuously changing value through the volume by a large number of small homogeneous regions, where the opacity per unit length is transformed based on the size of these regions.

The ScalarOpacity and Color transfer functions are typically used to perform a simple classification of the data. Scalar values that are part of the background, or considered "noise" are mapped to an opacity of 0.0, eliminating them from contributing to the image. The remaining scalar values can be divided into different "materials" which have different opacities and colors. For example, data acquired from a CT scanner can often be categorized as air, soft tissue, or bone based on the density value contained in the data (**Figure 6–11**). The scalar values defined as air would be given an opacity of 0.0, the soft tissue scalar values might be given a light brown color and the bone values might be given a white color. By varying the opacity of these last two materials, you can visualize the skin surface or the bone surface, or potentially see the bone through the translucent skin. This process of determining the dividing line between materials in the data can be tedious, and in some cases not possible based on the raw input data values. For example, liver and kidney sample locations may have overlapping CT density values. In this case, a segmentation filter may need to be applied to the volume to either alter the data values so that materials can be classified solely on the basis of the scalar value, or to extract out one specific material type. These segmentation operations can be based on additional information such as location or a comparison to a reference volume.

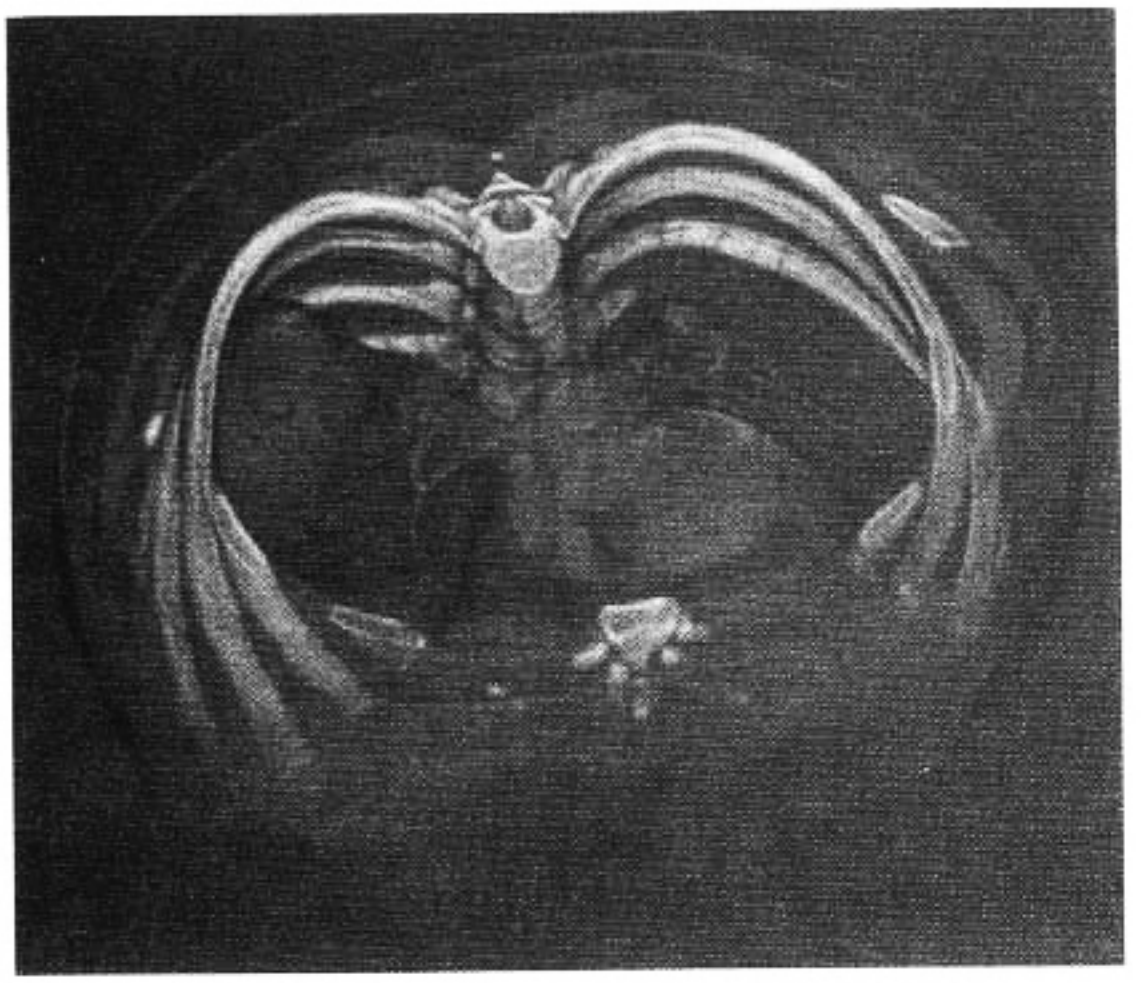


Figure 6–11 CT torso data classified using the ScalarOpacity, Color, and GradientOpacity transfer functions.

Two examples of segmenting CT data in this manner are shown here, one for a torso (**Figure 6–11**) and the other for a head study (**Figure 6–12**). In both of these examples, the third transfer function maps the magnitude of the gradient of the scalar value to an opacity multiplier, and is used to enhance the contribution of transition regions of the volume. For example, a large gradient magnitude can be found where the scalar value transitions from air to soft tissue, or soft tissue to bone, while within the soft tissue and bone regions the magnitude remains relatively small. Below is a code fragment that defines a typical gradient opacity transfer function for 8-bit unsigned data.

Figure 6–12 CT head data classified using the ScalarOpacity, Color, and GradientOpacity transfer functions.

```
vtkPiecewiseFunction gtfun
  gtfun AddPoint   0  0.0
  gtfun AddPoint   3  0.0
  gtfun AddPoint   6  1.0
  gtfun AddPoint 255  1.0
```

This function eliminates nearly homogeneous regions by defining an opacity multiplier of 0.0 for any gradient magnitude less than 3. This multiplier follows a linear ramp from 0.0 to 1.0 on gradient magnitudes between 3 and 6, and no change in the opacity value is performed on samples with magnitudes above 6. Noisier data may require a more aggressive edge detection (so the 3 and the 6 would be higher values).

There are a few methods in vtkVolumeProperty that relate to the color and opacity transfer functions. The SetColor() method accepts either a vtkPiecewiseFunction (if your color functions defines only greyscale values) or a vtkColorTransferFunction(). You can query the number of color channels with GetColorChannels() which will return 1 if a vtkPiecewiseFunction was set as the color, or 3 if a vtkColorTransferFunction was used to specify color. Once you know how many color channels are in use, you can call either GetGrayTransferFunction() or GetRGBTransferFunction() to get the appropriate function.

The SetScalarOpacity() method accepts a vtkPiecewiseFunction to define the scalar opacity transfer function, and there is a corresponding GetScalarOpacity() method that returns this function. Similarly, there are two methods for the gradient opacity transfer function: SetGradientOpacity() and GetGradientOpacity().

One other method in vtkVolumeProperty that can be used to control the color of a volume is SetRGBTextureCoefficient(). If a texture is defined in the volume mapper, and the volume mapper supports 3D texturing for the given rendering technique, then this coefficient will be considered when determining the color of a sample according to:

(color from scalar value) * (1 - RGBTextureCoefficient) +
(color from texture) * (RGBTextureCoefficient)

Setting the coefficient to 0.0 will ignore the texture and use only the ScalarColor transfer function, while setting it to 1.0 will use only the texture. Applying a texture to a volume will be discussed in more detail later in this section.

Controlling Shading with a vtkVolumeProperty

Controlling shading of a volume with a volume property is similar to controlling the shading of a geometric actor with a property (see "Actor Properties" on page 58 and "Actor Color" on page 59). There is a flag for shading, and four basic parameters: the ambient coefficient, the diffuse coefficient, the specular coefficient and the specular power. Generally, the first three coefficients will sum to 1.0 but exceeding this value is often desirable in volume rendering to increase the brightness of a rendered volume. The exact interpretation of these parameters will depend on the illumination equation used by the specific volume rendering technique that is being used. In general, if the ambient term dominates then the volume will appear unshaded, if the diffuse term dominates then the volume will appear rough (like concrete) and if the specular term dominates then the volume will appear smooth (like glass). The specular power can be used to control how smooth the appearance is (such as brushed metal versus polished metal).

By default, shading is off. You must explicitly call ShadeOn() for the shading coefficients to affect the scene. Setting the shading flag off is generally the same as setting the ambient coefficient to 1.0, the diffuse coefficient to 0.0 and the specular coefficient to 0.0. Some volume rendering techniques, such as volume ray casting with a maximum intensity ray function, do not consider the shading coefficients regardless of the value of the shading flag.

The shaded appearance of a volume depends not only on the values of the shading coefficients in the vtkVolumeProperty, but also on the collection of light sources contained in the renderer, and their properties. The appearance of a rendered volume will depend on the number, position, and color of the light sources in the scene.

If possible, the volume rendering technique attempts to reproduce the lighting equations defined by OpenGL. Consider the following example, where the creation of the renderer, render window, and interactor is left out for clarity.

```
#Create a geometric sphere
vtkSphereSource sphere
  sphere SetRadius 20
  sphere SetCenter 70 25 25
  sphere SetThetaResolution 50
  sphere SetPhiResolution 50

vtkPolyDataMapper mapper
  mapper SetInput [sphere GetOutput]

vtkActor actor
  actor SetMapper mapper
  [actor GetProperty] SetColor 1 1 1
  [actor GetProperty] SetAmbient 0.01
  [actor GetProperty] SetDiffuse 0.7
  [actor GetProperty] SetSpecular 0.5
  [actor GetProperty] SetSpecularPower 70.0

#Read in a volumetric sphere
vtkSLCReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/sphere.slc"

# Use this tfun for both opacity and color
vtkPiecewiseFunction opacityTransferFunction
    opacityTransferFunction AddSegment 0 1.0 255 1.0

# Make the volume property match the geometric one
vtkVolumeProperty volumeProperty
  volumeProperty SetColor opacityTransferFunction
  volumeProperty SetScalarOpacity tfun
  volumeProperty ShadeOn
  volumeProperty SetInterpolationTypeToLinear
  volumeProperty SetDiffuse 0.7
  volumeProperty SetAmbient 0.01
  volumeProperty SetSpecular 0.5
  volumeProperty SetSpecularPower 70.0

vtkVolumeRayCastCompositeFunction  compositeFunction
vtkVolumeRayCastMapper volumeMapper
  volumeMapper SetInput [reader GetOutput]
  volumeMapper SetVolumeRayCastFunction compositeFunction

vtkVolume volume
  volume SetMapper volumeMapper
  volume SetProperty volumeProperty
```

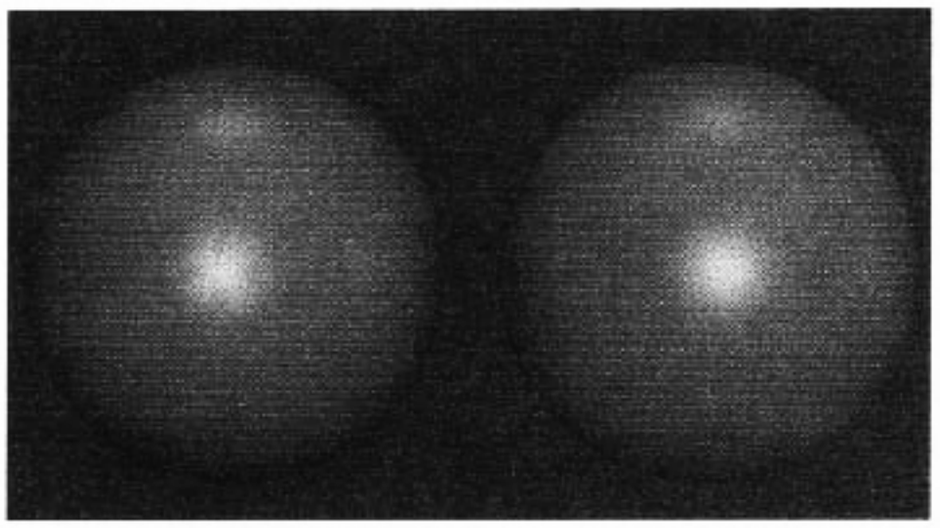


Figure 6–13 A geometric sphere (right) and a volumetric sphere (left) rendered with the same lighting coefficients.

```
# Add both the geometric and volumetric spheres to the renderer
ren1 AddProp volume
ren1 AddProp actor

# Create a red, green, and blue light
vtkLight redlight
  redlight SetColor 1 0 0
  redlight SetPosition 1000 25 25
  redlight SetFocalPoint 25 25 25
  redlight SetIntensity 0.5

vtkLight greenlight
  greenlight SetColor 0 1 0
  greenlight SetPosition 25 1000 25
  greenlight SetFocalPoint 25 25 25
  greenlight SetIntensity 0.5

vtkLight bluelight
  bluelight SetColor 0 0 1
  bluelight SetPosition 25 25 1000
  bluelight SetFocalPoint 25 25 25
  bluelight SetIntensity 0.5

# Add the lights to the renderer
ren1 AddLight redlight
ren1 AddLight greenlight
ren1 AddLight bluelight

#Render it!
renWin Render
```

In the image shown for this example (**Figure 6–13**), the left sphere is the volumetric one rendered with volumetric ray casting, and the right sphere is the geometric one rendered with OpenGL. Since the vtkProperty used for the vtkActor, and the vtkVolumeProperty used for the vtkVolume were set up with the same ambient, diffuse, specular, and specular power values, and the color of both spheres is white, they have similar appearances.

Creating a vtkVolumeMapper

vtkVolumeMapper is an abstract superclass and is never created directly. Instead, you would create a mapper subclass of the specific type desired. Currently, the choices are

vtkVolumeRayCastMapper, and vtkVolumeTextureMapper2D, or a vtkVolumeProMapper.

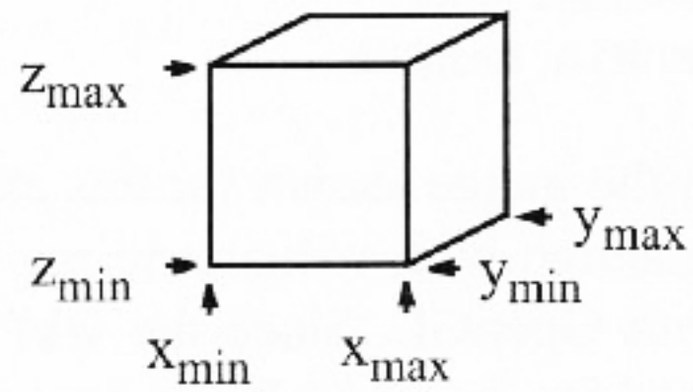
All volume mappers support the SetInput() method with an argument of a pointer to a vtkImageData object. Some rendering techniques support only certain types of input data. For example, the vtkVolumeRayCastMapper and the vtkVolumeTextureMapper2D both support only `VTK_UNSIGNED_CHAR` and `VTK_UNSIGNED_SHORT` data. The VolumePro mapper is even more restrictive in that it supports only 8 or 12 bit data, where the 12 bit data is represented as `VTK_UNSIGNED_SHORT` data with the significant bits either in the upper or lower 12 out of the 16 bits. (If your data is not of the types described above, you can use vtkImageShiftScale to scale the data to the right range and convert it to the right type.)

Be sure to set the input of the volume mapper before a volume that uses the volume mapper is rendered.

Cropping a Volume

Since volume rendered images of large, complex volumes can produce images that are difficult to interpret, it is often useful to view only a portion of the volume. The two techniques that can be used to limit the amount of data rendered are known as cropping and clipping.

Cropping is a method of defining visible regions of the volume using six planes—two along each of the major axes. These cropping planes are defined in data coordinates and are therefore dependent on the origin and spacing of the data, but are independent of any transformation applied to the volume. The most common way to use these six planes is to define a subvolume of interest as shown in the figure to the right.



To crop a subvolume, you must turn cropping on, set the cropping region flags, and set the cropping region planes in the volume mapper as shown below.

```
set xmin 10.0
set xmax 50.0
set ymin 0.0
set ymax 33.0
```

```
set zmin 21.0
set zmax 47.0

vtkVolumeRayCastMapper mapper
  mapper CroppingOn
  mapper SetCroppingRegionPlanes $xmin $xmax $ymin $ymax $zmin $zmax
  mapper SetCroppingRegionFlagsToSubVolume
```

Note that the above example is shown for a vtkVolumeRayCastMapper, but it could have instead used any concrete volume mapper since the cropping methods are all defined in the superclass.

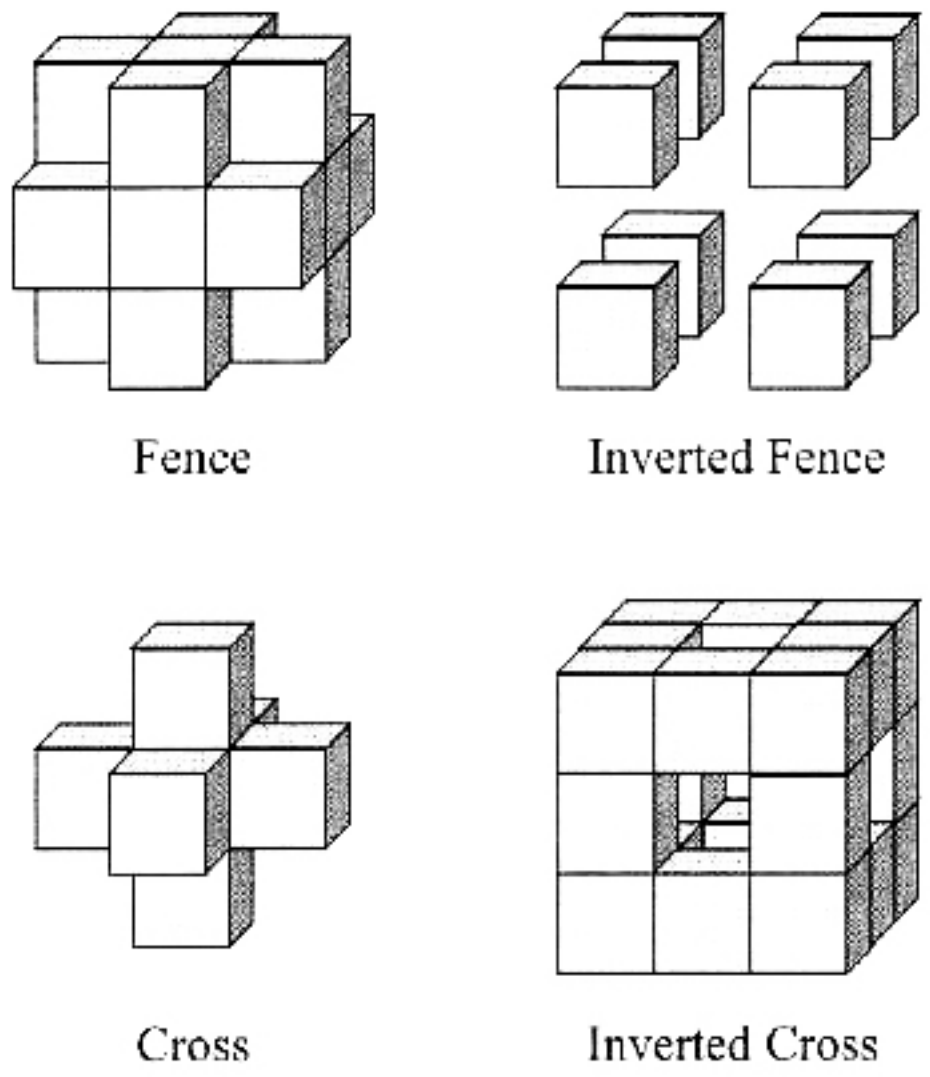


Figure 6–14 Cropping operations.

The six planes that are defined by the x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, and z_{max} values break the volume into 27 regions (a 3x3 grid). The CroppingRegionFlags is a 27 bit number with one bit representing each of these regions, where a value of 1 indicates that data within that region is visible, and a value of 0 indicating that data within that region will be cropped. The region of the volume that is less than x_{min}, y_{min}, and z_{min} is represented by the first bit, with regions ordered along the x axis first, then the y axis and finally the z axis.

The SetCroppingRegionFlagsToSubVolume() method is a convenience method that sets the flags to 0x0002000—just the center region is visible. Although any 27 bit number can be used to define the cropping operation, in practice there are only a few that are used. Four additional convenience methods are provided for setting these flags: SetCroppingRegionFlagsToFence(), SetCroppingResgionFlagsToInvertedFence(), SetCroppingRegionFlagsToCross(), and SetCroppingRegionFlagsToInvertedCross(), as depicted in **Figure 6–14**.

Clipping a Volume

In addition to the cropping functionality supplied by the vtkVolumeMapper, arbitrary clipping planes are provided in the vtkAbstractMapper3D. For subclasses of vtkAbstractMapper3D that use OpenGL to perform the clipping in hardware such as vtkOpenGLPolyDataMapper, and vtkOpenGLVolumeTextureMapper2D, an error message will be displayed if you attempt to use more than six clipping planes. Software rendering techniques such as vtkVolumeRayCastMapper can support an arbitrary number of clipping planes. The vtkVolumeProMapper does not support these clipping planes directly, although the class does contain methods for specifying one clipping box using a plane and a thickness value.

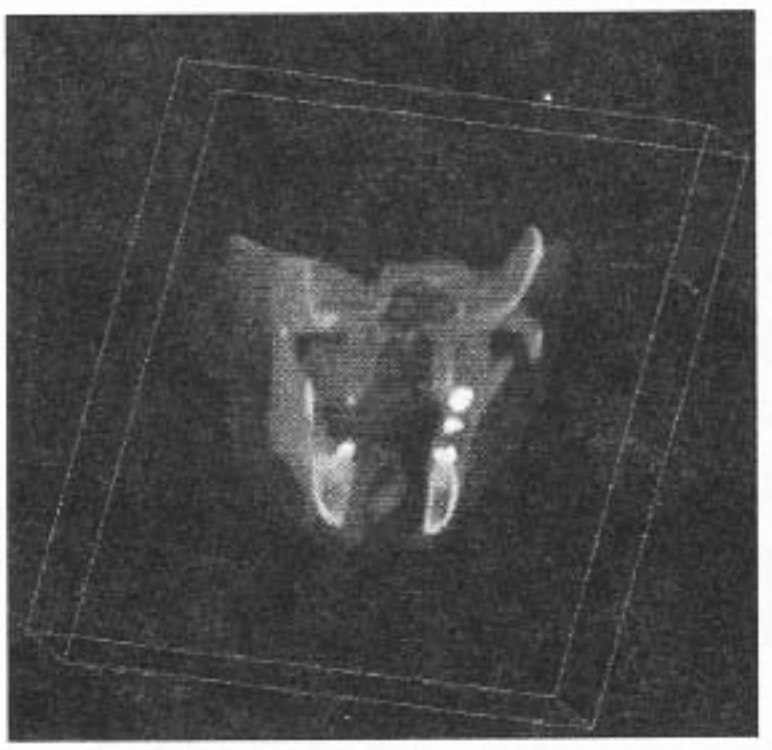


Figure 6–15 Clipping planes are used to define a thick slab.

The clipping planes are specified by creating a vtkPlane, defining the plane parameters, then adding this plane to the mapper using the AddClippingPlane() method. The most common use of these arbitrary clipping planes in volume rendering is to specify two planes parallel to each other in order to perform a thick reformatting operation. An example of this applied to CT data is shown in **Figure 6–15**.

Applying 3D Texture to a Volume

The volume mapper accepts data of the same type as SetInput() to the SetRGBTextureInput() method: a pointer to a vtkImageData object. In the current implementation, the texture input must have three components which represent R, G, and B, and each component must be an unsigned char. Not all volume rendering techniques make use of the RGBTextureInput. In the current implementation, the only volume rendering technique that does make use of the texture is the vtkVolumeRayCastMapper when a vtkVolumeRayCastIsosurfaceFunction is used as the ray cast function. An example script is

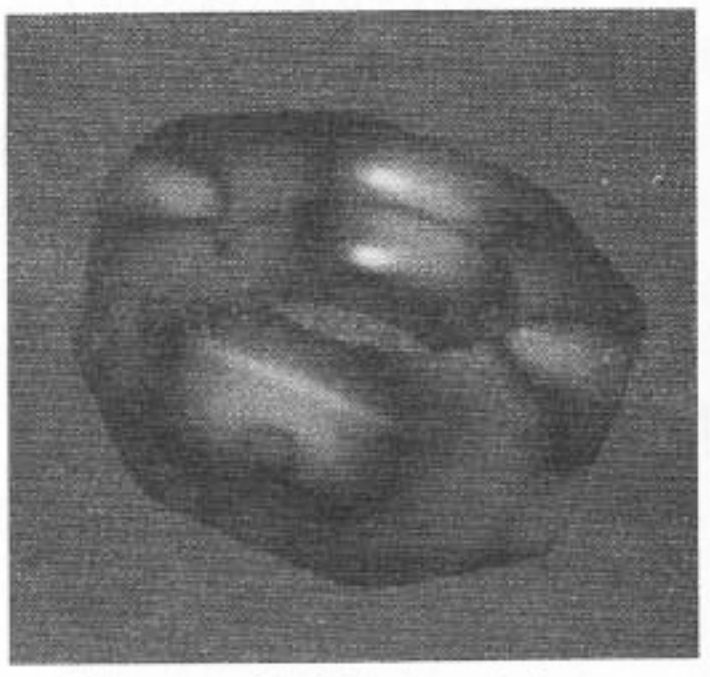


Figure 6–16 Applying texture to a volume.

shown below (refer to **Figure 6–16**). Note that the creation of the renderer, render window, and interactor have been left out for clarity.

```
# A simple example of using a texture
# on a volume rendered with ray casting
# and an isosurface ray cast function.
# The texture data was created from two
# other volume data sets - with one data
# set duplicated to define R and B, and
# the other used to define G.

# Read the input data from disk
vtkSLCReader reader
    reader SetFileName "$VTK_DATA_ROOT/Data/nut.slc"

# Read the texture input from disk
vtkStructuredPointsReader rgbreader
    rgbreader SetFileName "$VTK_DATA_ROOT/Data/hipipTexture.vtk"

# Create transfer function for opacity - the surface is opaque
vtkPiecewiseFunction opacityTransferFunction
    opacityTransferFunction AddSegment 0 1.0 255 1.0

# Create the property - all color comes from the texture
vtkVolumeProperty volumeProperty
    volumeProperty SetScalarOpacity opacityTransferFunction
    volumeProperty ShadeOn
    volumeProperty SetInterpolationTypeToLinear
    volumeProperty SetRGBTextureCoefficient 1.0

# Create the isosurface function
vtkVolumeRayCastIsosurfaceFunction  isoFunction
    isoFunction SetIsoValue 128.0

# Create the mapper - set the input and texture
vtkVolumeRayCastMapper volumeMapper
    volumeMapper SetInput [reader GetOutput]
    volumeMapper SetRGBTextureInput [rgbreader GetOutput]
    volumeMapper SetVolumeRayCastFunction isoFunction

# Create the volume
vtkVolume volume
    volume SetMapper volumeMapper
    volume SetProperty volumeProperty
```

```
# Add the volume and render
ren1 AddProp volume
renWin Render
```

Controlling the Normal Encoding

The standard illumination equation relies on a surface normal in order to calculate the diffuse and specular components of shading. In volume rendering, the gradient at a location in the volumetric data is considered to point in the opposite direction of the "surface normal" at that location. A finite differences technique is typically used to estimate the gradient, but this tends to be an expensive calculation, and would make shaded volume rendering prohibitively slow if it had to be performed at every sample along every ray.

One way to avoid these expensive computations is to precompute the normals at the grid locations, and to use some form of interpolation in between. If done naively, this would require three floating point numbers per location, and we would still need to take a square root to determine the magnitude. Alternatively, we could store the magnitude so that each normal would require four floating point values. Since volumes tend to be quite large, this technique requires too much memory, so we must somehow quantize the normals into a smaller number of bytes.

In VTK we have chosen to quantize the normal direction into two byes, and the magnitude into one. The calculation of the normal is performed by a subclass of vtkEncodedGradientEstimator (currently only vtkFiniteDifferenceGradientEstimator) and the encoding of the direction into two bytes is performed by a subclass of vtkDirectionEncoder (currently only vtkRecursiveSphereDirectionEncoder). For mappers that use normal encoding (vtkVolumeRayCastMapper and vtkVolumeTextureMapper2D), these objects are created automatically so the typical user need not be concerned with these objects. In the case where one volume dataset is to be rendered by multiple mappers into the same image, it is often useful to create one gradient estimator to set in all the mappers. This will conserve space since otherwise there would be one copy of the normal volume per mapper. An example fragment of code is shown below:

```
# Create the gradient estimator
vtkFiniteDifferenceGradientEstimator GradientEstimator

# Create the first mapper
vtkVolumeRayCastMapper volumeMapper1
  volumeMapper1 SetGradientEstimator GradientEstimator
```

```
  volumeMapper1 SetInput [reader GetOutput]

# Create the second mapper
vtkVolumeRayCastMapper volumeMapper2
  volumeMapper2 SetGradientEstimator GradientEstimator
  volumeMapper2 SetInput [reader GetOutput]
```

If you set the gradient estimator to the same object in two different mappers, then it is important that these mappers have the same input. Otherwise, the gradient estimator will be out-of-date each time the mapper asks for the normals, and will regenerate them for each volume during every frame rendered. In the above example, the direction encoding objects were not explicitly created, therefore each gradient estimator created its own encoding object. Since this object does not have any significant storage requirements, this is generally an acceptable situation. Alternatively, one vtkRecursiveSphereDirectionEncoder could be created, and the SetDirectionEncoder() method would be used on each estimated to associate the encoder with the estimator.

Volumetric Ray Casting

The vtkVolumeRayCastMapper is a volume mapper that employs a software ray casting technique to perform volume rendering. It is generally the most accurate mapper, and also the slowest on most platforms. The ray caster is threaded to make use of multiple processors when available.

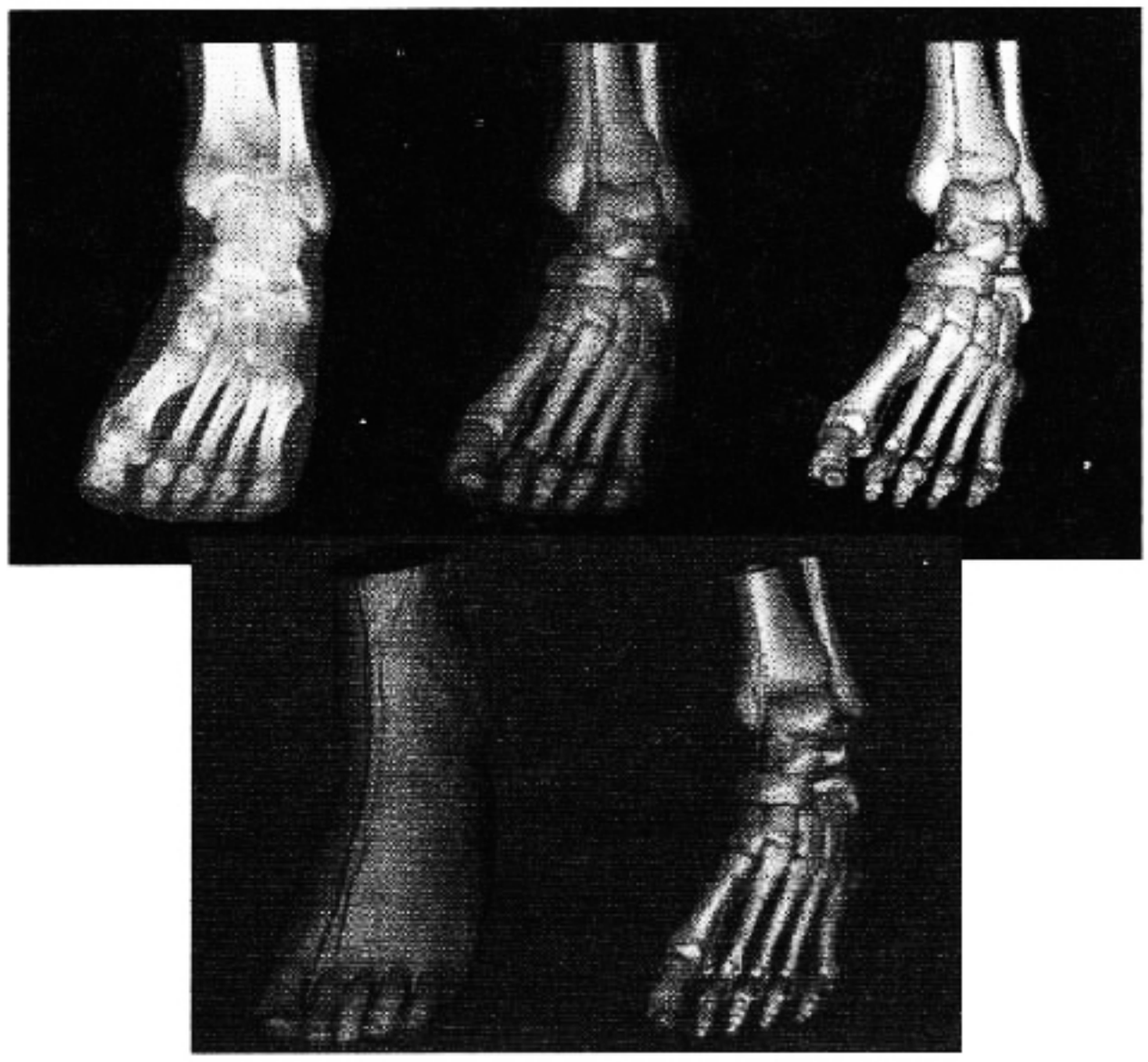
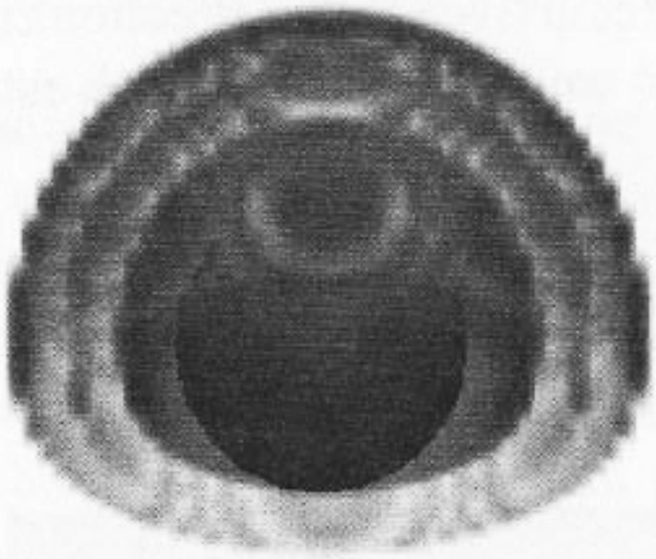


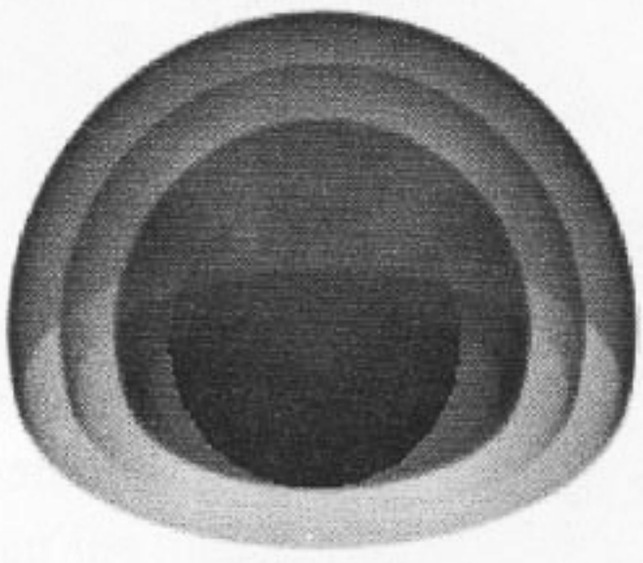
Figure 6–17 Volume rendering via ray casting.

There are a few parameters that are specific to volume ray casting that have not yet been discussed. First, there is the ray cast function that must be set in the mapper. This is the object that does the actual work of considering the data values along the ray and determining a final RGBA value to return. Currently, there

are three supported subclasses of vtkVolumeRayCastFunction: the vtkVolumeRayCast-IsosurfaceFunction that can be used to render isosurfaces within the volumetric data, the vtkVolumeRayCastMIPFunction that can be used to generate maximum intensity projections of the volume, and vtkVolumeRayCastCompositeFunction that can be used to render the volume with an alpha compositing technique. An example of the images that can be generated using these different methods is **Figure 6–17**. The upper left image was generated using a maximum intensity projection. The other two upper images were generated using compositing, while the lower two images were generated using an isosurface function. Note that it is not always easy to distinguish an image generated using a compositing technique from one generated using an isosurface technique, especially when a sharp opacity ramp is used.



Classify First



Interpolate First

Figure 6–18 The effect of interpolation order in composite ray casting.

There are some parameters that can be set in each of the ray cast functions that impact the rendering process. In vtkVolumeRayCastIsosurfaceFunction, there is a SetIsoValue() method that can be used to set the value of the rendered isosurface. In vtkVolumeRayCastMIPFunction, you can call SetMaximizeMethodToScalarValue() (the default) or SetMaximizeMethodToOpacity() to change the behavior of the maximize operation. In the first case, the scalar value is considered at each sample point along the ray. The sample point with the largest scalar value is selected, then this scalar value is passed through the color and opacity transfer functions to produce a final ray value. If the second method is called, the opacity of the sample is computed at each step along the ray, and the sample with the highest opacity value is selected.

In vtkVolumeRayCastCompositeFunction, you can call SetCompositeMethodToInterpolateFirst() (the default) or SetCompositeMethodToClassifyFirst() to change the order of interpolation and classification (**Figure 6–18**). This setting will only have an impact when trilinear interpolation is being used. In the first case, interpolation will be performed to determine the scalar value at the sample point, then this value will be used for classification (the application of the color and opacity transfer functions). In the second case, classification is done at the eight vertices of the cell containing the sample location, then the final RGBA value is interpolated from the computed RGBA values at the vertex locations. Interpolat-

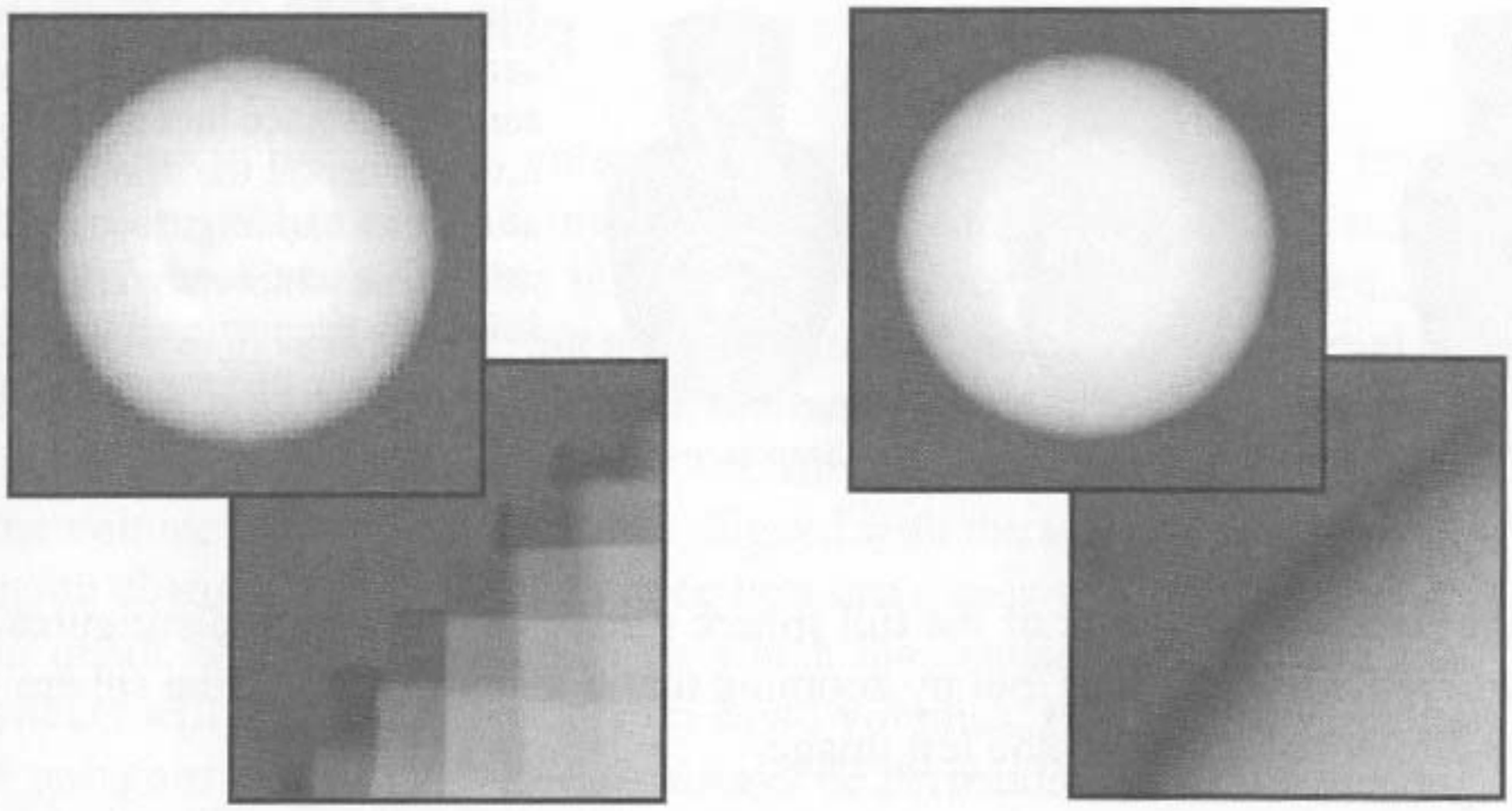


Figure 6–19 Different methods for interpolation. On the left, nearest neighbor interpolation. On the right, trilinear interpolation.

ing first generally produces "prettier" images as can be seen on the left where a geometric sphere is contained within a volumetric "distance to point" field, with the transfer functions defined to highlight three concentric spherical shells in the volume. The interpolate first method makes the underlying assumption that if two neighboring data points have values of 10 and 100, then a value of 50 exists somewhere between the two data points. In the case where material is being classified by scalar value, this may not be the case. For example, consider CT data where values below 20 are air (transparent), values from 20 to 80 are soft tissue, and values above 80 are bone. If interpolation is performed first, then bone can never be adjacent to air - there must always be soft tissue between the bone and air. This is not true inside the mouth where teeth meet air. If you render an image with interpolation performed first and a high enough sample rate, it will look like the teeth have a layer of skin on top of them.

The value of the interpolation type instance variable in the vtkVolumeProperty is important to ray casting. There are two options: SetInterpolationTypeToNearest() (the default) which will use a nearest neighbor approximation when sampling along the ray, and SetInterpolationTypeToLinear() which will use trilinear interpolation during sampling. Using the trilinear interpolation produces smoother images with less artifacts, but generally takes a bit longer. The difference in image quality obtained with these two methods is shown in **Figure 6–19**. A sphere is voxelized into a 50x50x50 voxel volume, and rendered using alpha compositing with nearest neighbor interpolation on the left and trilinear interpola-

Figure 6–20 The effects or varying sample distance along the ray. As the sample distance increases, sampling artifacts create the dramatic black and white banding. However, the cost of volume rendering increases inversely proportional to the sample size, i.e., the difference in rendering time for sample distance 0.1 is 20x faster than for 2.0.

tion on the right. In the image of the full sphere it may be difficult to distinguish between the two interpolation methods, but by zooming up on just a portion of the sphere it is easy to see the individual voxels in the left image.

Another parameter of vtkVolumeRayCastMapper that affects the image is the SampleDistance. This is the distance in world coordinates between sample points for ray functions that take samples. For example, the alpha compositing ray function performs a discrete approximation of the continuous volume rendering integral by sampling along the ray. The accuracy of the approximation increases with the number of samples taken, but unfortunately so does the rendering time. The maximum intensity ray function also takes samples to locate the maximum value. The isosurface ray function does not take samples but instead computes the exact location of the intersection according to the current interpolation function.

By default samples are taken 1 unit apart in world coordinates. In practice you should adjust this spacing based on the sample spacing of the 3D data being rendered, and the rate of change of not only the scalar values but also the color and opacity assigned to the scalar values through the transfer functions. An example is shown below of a voxelized vase with a 1x1x1 spacing between samples in the dataset. The scalar values vary smoothly in the data, but a sharp change has been introduced in the transfer functions by having the color change rapidly from black to white. You can clearly see artifacts of the "undersampling" of ray casting in the image created with a step size of 2.0. Even with a step size of 1.0 there are some artifacts since the color of the vase changes significantly within a world space distance of 1.0. If the sample distance is set to 0.1 the image appears smooth. Of course, this smooth image on the left takes nearly 20 times as long to generate as the one on the right.

2D Texture Mapping

As an alternative to ray casting, volume rendering can be performed by texture mapping the volume onto polygons, and projecting these with the graphics hardware. If your graphic board provides reasonable texture mapping acceleration, this method will be significantly faster than ray casting, but at the expense of accuracy since partial accumulation results are stored at the resolution of the framebuffer (usually 8 or less bits per component) rather than in floating point. To use 2D texture mapping, quads are generated along the axis of the volume which is most closely aligned with the viewing direction. As the viewing direction changes, the sample distance between quads will change, and at some point the set of quads will jump to a new axis which may cause temporal artifacts. Generally these artifacts will be most noticeable on small volumes. If 3D texture mapping is available, the polygons can be generated to always be perpendicular to the viewing direction, greatly reducing these artifacts. In this case, the polygons may be anything from 3 sided triangles to 6 sided hexagons. Unfortunately, 3D texture mapping is not yet widely supported across graphics cards.

The vtkVolumeTextureMapper is the superclass for volume mappers that employ hardware texture mapping as the rendering technique. Currently, the only concrete implementation is the vtkVolumeTextureMapper2D. Since this class requires access to the underlying graphics language, there is an automatically created graphics-specific object (vtkOpenGLVolumeTextureMapper2D) as well.

The current implementation of vtkVolumeTextureMapper2D supports only alpha compositing. Bilinear interpolation on the slice is used for texture mapping but since quads are only created on the data planes, there is no notion of interpolation between slices. Therefore, the value of the InterpolationType instance variable in the vtkVolumeProperty is ignored by this mapper.

Shading is supported in software for the texture mapping approach. If shading is turned off in the vtkVolumeProperty, then software shading calculations do not need to be performed, and therefore the performance of this mapper will be better than if shading is turned on.

VolumePro Rendering Hardware

Support has been included in VTK for the VolumePro rendering hardware. Both the VolumePro 500 and the VolumePro 1000 are supported in VTK 4.0 although the description

in this chapter covers mainly the VolumePro 500. Please see the latest online documentation for a description of methods and functionality specific to the VolumePro 1000.

The version of this VolumePro 500 board implements a ray casting strategy to render volumes of up to 256x256x256 voxels (8 or 12 bits per voxel) at rates of 20 to 30 frames per second. Larger volumes can also be rendered, although at slower rates. For more information on the VolumePro product line, please visit the TeraRecon web site at http://www.terarecon.com.

In order to compile the vtkVolumeProMapper into VTK, you will need the header and library files for the board. Please visit the VTK web site at http://www.visualizationtoolkit.org in order to download the header and library files for the VolumePro 1000. If you are using the VolumePro 500, you will need to purchase the Software Development Kit from TeraRecon. To add this class to your VTK build you need to enable the `VTK_USE_VOLUMEPRO` option during setup.

If you are using a more recent version of VTK than the one that came on the CD-ROM with this book, please consult the instructions included in the comment block at the top of the file `VTK/Rendering/vtkVolumeProMapper.h` for the latest integration instructions.

Since all the work of volume rendering is performed on the special-purpose board, this is the simplest volume mapper class in VTK. The class essentially translates all the parameters of volume rendering that are stored in the vtkVolume, the vtkVolumeProperty, and the vtkVolumeMapper into vli (the library interface to the VolumePro board) classes and parameters. To create a mapper of this type, you would use the vtkVolumeProMapper class. This will automatically create the correct subclasses. Previously, the only supported version of the VolumePro hardware was the one based on the VG500 chip, so the New() method of vtkVolumeProMapper called the New() method of vtkVolumeProVG500Mapper. In addition, the results from the board are rendered to the graphics context by VTK using OpenGL, so a vtkOpenGLVolumeProVG500Mapper was the class that was finally created. In VTK 4.0, a choice will be made at either compile time (based on what headers and libraries are found) or at runtime (based on which driver is running) to select between the VolumePro 100 and VolumePro 500 versions of this class.

There are a few parameters that do not map well into vli. The arbitrary clipping planes defined in vtkAbstractMapper3D are not supported by the vtkVolumeProMapper, although thick slab clipping is supported as will be described later. The cropping region flags must be either set to off, subvolume, fence, inverted fence, cross, or inverted cross.

Specifying an arbitrary bit pattern that does not match one of these options is not supported by the vtkVolumeProMapper.

Two significant limitations of the VolumePro 500 version vtkVolumeProMapper is that it works only with parallel camera transformations, and geometry cannot be mixed within the volume. Undesirable results will occur if you attempt to use a perspective viewing transformation, or geometry is rendered within the volume. The VolumePro 1000 does support intermixed geometry although this feature may not make it into the VTK 4.0 release. Support for limited perspective projection may be added at some point for the VolumePro 1000 mapper, please check the online documentation for more details.

The VolumePro mapper does have some mapper-specific functionality. The hardware is based on a ray casting approach and supports three different ray functions. These can be selected by calling SetBlendModeToComposite(), SetBlendModeToMaximumIntensity(), or SetBlendModeToMinimumIntensity().

Since the VolumePro hardware does not support geometry that is intermixed with the volume, special functionality was added to support a 3D cursor. The following code fragment would set up a 3D cursor for a VolumePro mapper.

```
vtkVolumeProMapper volumeMapper
  volumeMapper CursorOn
  volumeMapper SetCursorTypeToCrossHair
  volumeMapper SetCursorPosition 10 20 30
```

Figure 6–21 Defining a cursor in the VolumePro hardware.

Alternatively, the cursor type could have been set to Plane, which would use three orthogonal planes instead of lines. By default the cursor type is CrossHair, and the *x*-axis line is drawn in red, the *y*-axis line is drawn in green, and the *z*-axis line is drawn in blue (**Figure 6–21**). When planes are used instead, the plane perpendicular to the *x*, *y*, or *z* axis is drawn in red, green, or blue, respectively. The cursor functionality is not supported in the VolumePro 1000 mapper since intermixed geometry is supported.

The VolumePro hardware supports supersampling in three dimensions by taking samples closer together along a ray, and casting more rays to generate an image. Supersampling can be turned on with the SuperSamplingOn() method, and the amount of supersampling can be controlled by SetSuperSamplingFactor(xfactor, yfactor, zfactor). Each factor

should be a number between 0.125 and 1.0 representing the sampling distance along that axis, where a value of 1.0 would imply no supersampling and a value of 0.125 would imply an 8x supersampling. The supersampling factor for each axis will be rounded to the nearest $1/n$ where n is an integer between 1 and 8 inclusively.

As mentioned earlier, the vtkVolumeProMapper does not support the arbitrary clipping planes that are defined in the vtkAbstractMapper3D superclass. This mapper does support one arbitrarily-oriented thick cutting plane. There are several steps to setting up a thick cut plane, as shown in the code fragment below.

```
vtkVolumeProMapper volumeMapper
  volumeMapper CutPlaneOn
  volumeMapper SetCutPlaneEquation 0.5 0.3 0.8 3.4
  volumeMapper SetCutPlaneThickness 30
  volumeMapper SetCutPlaneFalloffDistance 4
```

As shown in this example, the cut plane must be turned on, the plane equation coefficients (A, B, C, D) must be specified where $Ax + By + Cz + D = 0$, and the thickness and opacity fall-off of the cut plane must be set. The opacity fall-off distance should be 0, 1, 2, 4, 8 or 16 and represents the distance in voxel from the cut plane during which opacity drops from completely opaque to completely transparent. A distance of 0 produces a sudden transition (similar to the arbitrary cut plane functionality supported in the other mappers) while a distance of 16 will produce a blurred or fuzzy cut.

There are three flags in the vtkVolumeProMapper that indicate what visualization parameters are modulated by the magnitude of the gradient of the scalar value. In order for the opacity to be modulated according to the GradientOpacityTransferFunction set in the vtkVolumeProperty as it is in other volume mappers, the GradientOpacityModulationOn() method must be called in the vtkVolumeProMapper. In addition, the diffuse and specular lighting calculation can be altered based on the gradient magnitude by calling GradientDiffuseModulationOn() and GradientSpecularModulationOn().

Speed vs. Accuracy Trade-offs

If you do not have a VolumePro volume rendering board, many fast CPUs, or high-end graphics hardware, you will probably not be satisfied with the rendering rates achieved when one or more volumes are rendered in a scene. It is often necessary to achieve a certain frame rate in order to effectively interact with the data, and it may be necessary to

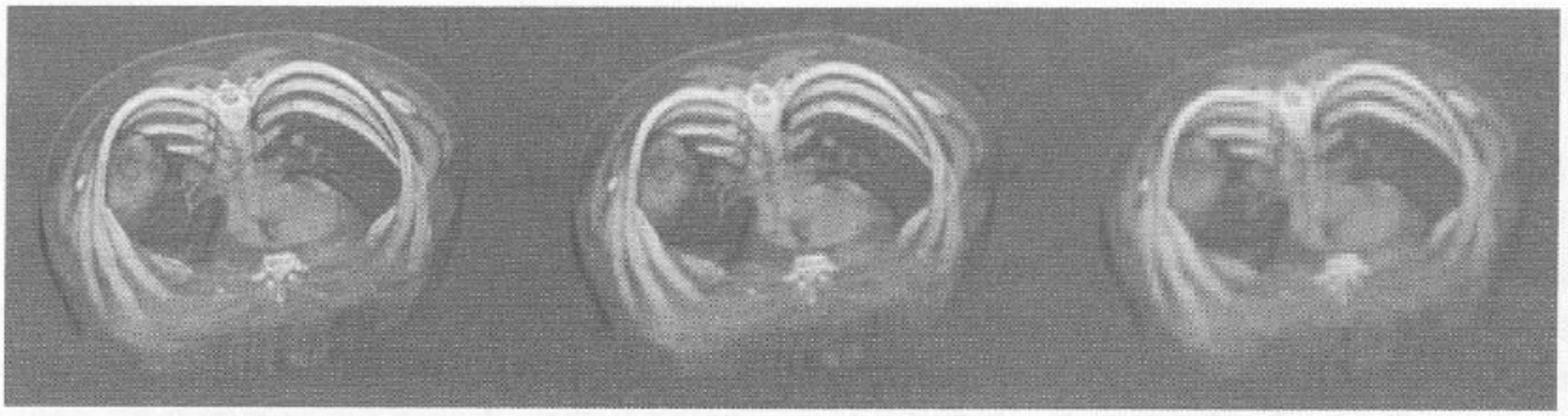
trade off accuracy in order to achieve speed. Fortunately, there are ways to do this for both ray casting and texture mapping approaches.

The support for achieving a desired frame rate for ray casting is available by default in VTK. You can set the desired update rate in the vtkRenderWindow, or the StillUpdateRate and the DesiredUpdateRate in the interactor if you are using one. Due to the fact that the time required for ray casting is mostly dependent on the size of the image, the ray caster mapper will automatically attempt to achieve the desired rendering rate by reducing the number of rays that it casts. By default, the automatic adjustment is on. In order to maintain interactivity, an abort check procedure should be specified in the render window so that the user will be able to interrupt the higher resolution image in order to interact with the data again.

There are limits on how blocky the image will become in order to achieve the desired update rate. By default, the adjustment will allow the image to become quite blocky, casting only 1 ray for every 10x10 neighborhood of pixels if necessary to achieve the desired update rate. Also by default the ray cast mapper will not cast more than 1 ray per pixel. These limits can be adjusted in the vtkVolumeRayCastMapper by setting the MinimumImageSampleDistance and MaximumImageSampleDistance. In addition AutoAdjustSampleDistances can be turned off, and the specified ImageSampleDistance will be used to represent the spacing between adjacent rays on the image plane. Results for one example are shown in **Figure 6–22**.

This technique of reducing the number of rays in order to achieve interactive frame rates can be quite effective. For example, consider the full resolution image shown on the left in **Figure 6–22**. This image may require 4 seconds to compute, which is much too slow for data interaction such as rotating or translating the data, or interactively adjusting the transfer function. If we instead subsample every other ray along each axis by setting the ImageSampleDistance to 2.0, we will get an image like the one shown in the middle in only about 1 second. Since this still may be too slow for effective interaction, we could subsample every fourth ray, and achieve rendering rates of nearly 4 frames per second with the image shown on the right. It may be blocky, but it is far easier to rotate a blocky volume at 4 frames per second than a full resolution volume at one frame every four seconds.

There are no built-in automatic techniques for trading off accuracy for speed in a texture mapping approach. This can be done by the user fairly easily by creating a lower resolution volume using vtkImageResample, and rendering this new volume instead. Since the speed of the texture mapping approach is highly dependent on the size of the volume, this will achieve similar results to reducing the number of rays in a ray casting approach.



ImageSampleDistance = 1.0 ImageSampleDistance = 2.0 ImageSampleDistance = 4.0

Figure 6–22 The effect of changing the automatic scale lower limit on image quality.

Another option is to reduce the number of planes sampled through the volume. By default, the number of textured quads rendered will be equal to the number of samples along the major axis of the volume (as determined by the viewing direction). You may set the MaximumNumberOfPlanes instance variable to decrease the number of textured quads and therefore increase performance. The default value is 0 which implies no limit on the number of planes.

Using a vtkLODProp3D to Improve Performance

The vtkLODProp3D is a 3D prop that allows for the collection of multiple levels-of-detail and decides which to render for each frame based on the allocated rendering time of the prop (see "vtkLODProp3D" on page 64). The allocated rendering time of a prop is dependent on the desired update rate for the rendering window, the number of renderers in the render window, the number of props in the renderer, and any possible adjustment that a culler may have made based on screen coverage or other importance factors.

Using a vtkLODProp3D, it is possible to collect several rendering techniques into one prop, and allow the prop to decide which technique to use. Consider the following example of creating a vtkLODProp3D:

```
vtkImageResample resampler
  resampler SetAxisMagnificationFactor 0 0.5
  resampler SetAxisMagnificationFactor 1 0.5
  resampler SetAxisMagnificationFactor 2 0.5
```

```
vtkVolumeTextureMapper2D lowresMapper
  lowresMapper SetInput [resampler GetOutput]

vtkVolumeTextureMapper2D medresMapper
  medresMapper SetInput [reader GetOutput]

vtkVolumeRayCaster hiresMapper
  hiresMapper SetInput [reader GetOutput]

vtkLODProp3D volumeLOD
  volumeLOD AddLOD lowresMapper volumeProperty 0.0
  volumeLOD AddLOD medresMapper volumeProperty 0.0
  volumeLOD AddLOD hiresMapper volumeProperty 0.0
```

For clarity, many steps of reading the data and setting up visualization parameters have been left out of this example. At render time, one of the three levels-of-detail (LOD) for this prop will be selected based on the estimated time that it will take to render the LODs and the allocated time for this prop. In this case, all three levels-of-detail use the same property, but they could have used different properties if necessary. Also, in this case all three mappers are subclasses of vtkVolumeMapper, but we could add a bounding box representation as another level-of-detail, which would have a subclass of vtkMapper for the mapper, and a vtkProperty for the property parameter.

The last parameter of the AddLOD() method is an initial time to use for the estimated time required to render this level-of-detail. Setting this value to 0.0 requires that the LOD be rendered once before an estimated render time can be determined. When a vtkLODProp3D has to decide which LOD to render, it will choose one with 0.0 estimated render time if there are any. Otherwise, it will choose the LOD with the greatest time that does not exceed the allocated render time of the prop, if it can find such an LOD. Otherwise, it will choose the LOD with the lowest estimated render time. The time required to draw an LOD for the current frame replaces the estimated render time of that LOD for future frames.

Capabilities / Limitation of the Techniques

The following table (**Table 6–1**) indicates the level of support that the three different volume mappers provide for the features that have been discussed in this section. You may wish to refer to this table to choose the best volume rendering technique for you application.

Table 6–1 Features of the basic volume rendering mapper types supported in VTK.

Feature	Ray Casting	Texture Mapping	Volume-Pro
Alpha Blending / Compositing Projection	Yes	Yes	Yes
Maximum Intensity Projection	Yes	No	Yes
Minimum Intensity Projection	No[1]	No	Yes
Isosurface Projection	Yes	No	No
Parallel Viewing Transform	Yes	Yes	Yes
Perspective Viewing Transform	Yes	Yes	No
SuperSampling in XY	No	No	Yes
SuperSampling in Z	Yes[2]	No	Yes
InterpolationType in vtkVolumeProperty is Important	Yes	No	No
Arbitrary Clipping Planes	Yes	Yes[3]	Partial[4]
Cropping Regions	Yes	Yes	Partial[6]
Intermixed with Opaque Geometry in a Scene	Yes	Yes	Partial[7]
Intermixed with Other Volumes in a Scene	Partial[8]	Partial[8]	Partial[8]
3D Cursor	No[9]	No[9]	Yes[5,9]
Modulation of Opacity by Gradient Magnitude	Yes	Yes	Yes
Modulation of Specular Lighting by Gradient Magnitude	No	No	Yes
Modulation of Diffuse Lighting by Gradient Magnitude	No	No	Yes
Automatic Speed / Accuracy Trade-off	Yes	No	No
Multiprocessor Support	Yes	No	No
Classes Included in VTK by Default	Yes	Yes	No

1. Can be achieved by reversing the opacity ramp.
2. Not applicable for isosurface projections.
3. Limited by number supported in OpenGL.
4. Only 1 arbitrary cut plane with a thickness.
5. Only for VolumePro 500.
6. Only the basic region types.
7. No intersection geometry for VolumePro 500.
8. Volumes cannot intersect each other's bounds.
9. Use of intermixed opaque geometry