Scalar Visualization – Part I

Lisa Avila
Kitware, Inc.

Overview

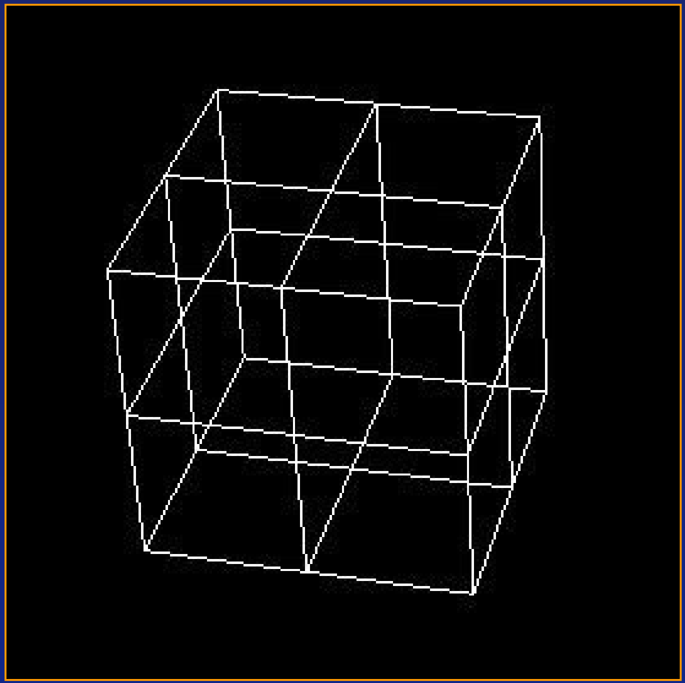
Topics covered in Part I:

- Color mapping
- Cutting
- Contouring
- Image processing

Topics covered in Part II:

- Volume rendering
  - Controlling the volume appearance
  - Cropping and clipping
  - Intermixing and interactivity

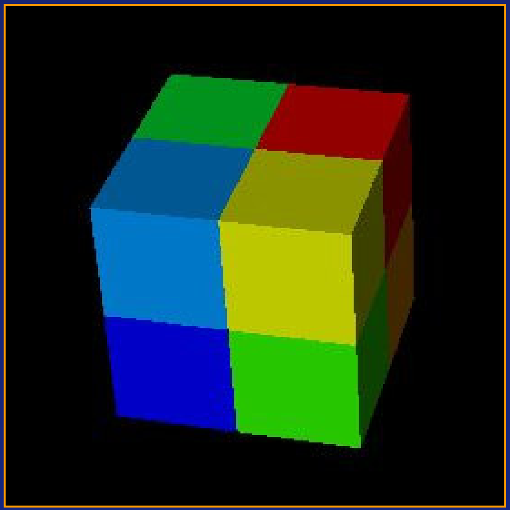
Cell Scalars and Point Scalars



vtkUnstructuredGrid
with 8 vtkVoxel cells

5

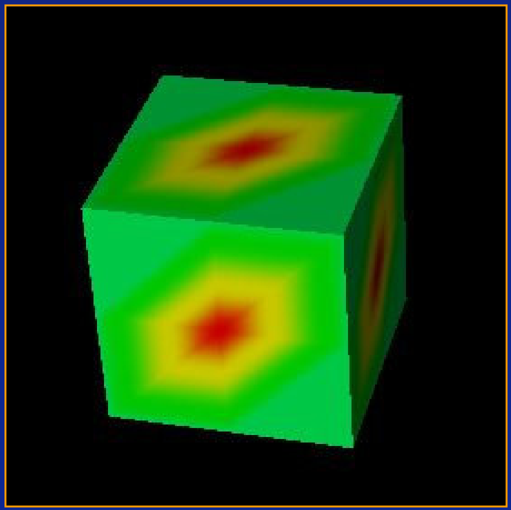
Cell Scalars



5 4

3 7

Point Scalars



Scalar Components

If type is unsigned char, scalars can map directly to color

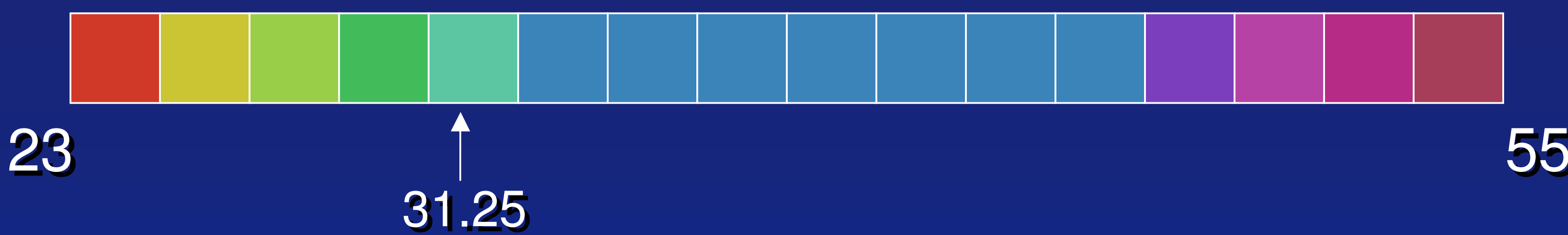
- One component: (s1, s1, s1, 1)
- Two components: (s1, s1, s1, s2)
- Three components: (s1, s2, s3, 1)
- Four components: (s1, s2, s3, s4)

```
mapper->SetColorModeToDefault();
mapper->SetColorModeToMapScalars();
```

Everything else must be mapped through a lookup table to assign color to a scalar value

Color Mapping

vtkLookupTable provides an array of color values that can be initialized using simple HSV and alpha ramps, or defined manually.

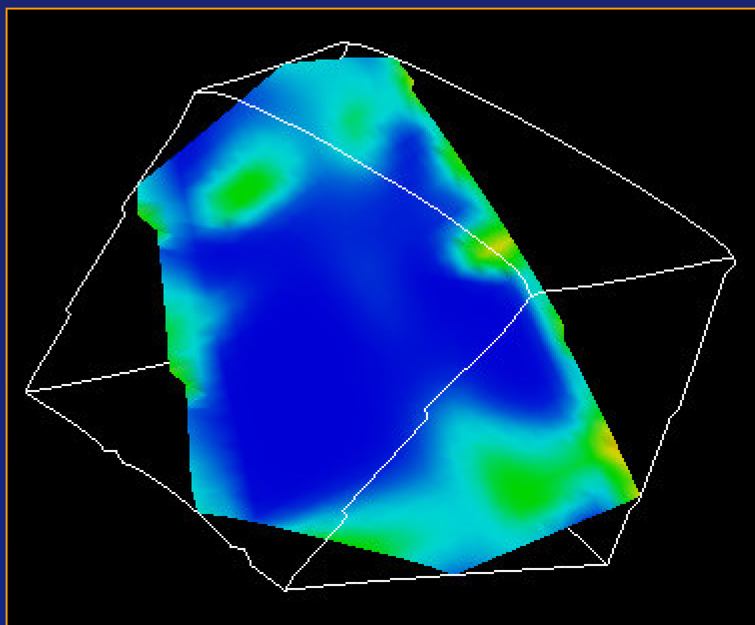


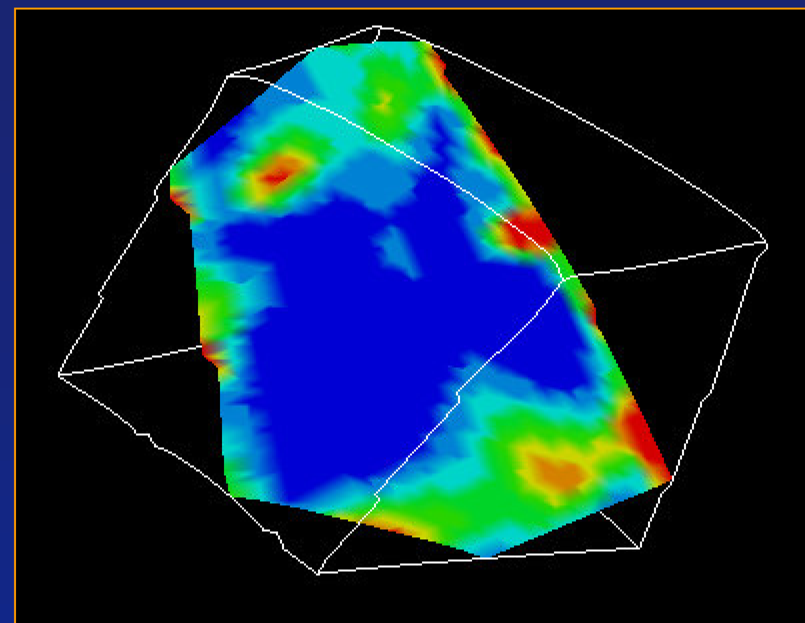
vtkColorTransferFunction provides an interface for specifying color nodes in a piecewise-linear function.

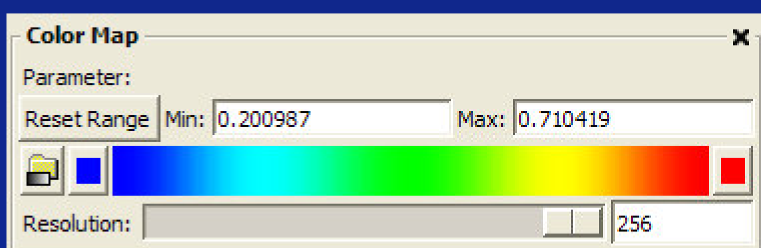
Color Mapping

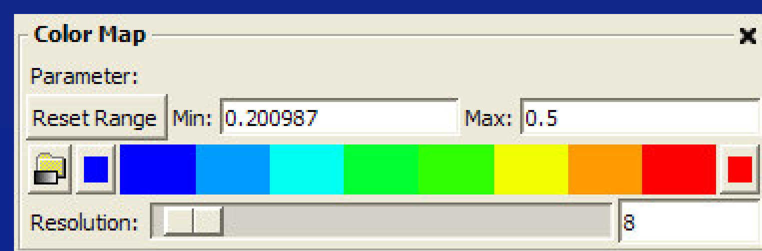
```
vtkLookupTable *lut = vtkLookupTable::New();
lut->SetNumberOfTableValues( 64 );
lut->SetHueRange( 0.0, 0.667 );

lut->SetNumberOfTableValues(4);
lut->SetTableValue( 0, 1.0, 0.0, 0.0, 1.0 );
lut->SetTableValue( 1, 0.0, 1.0, 0.0, 1.0 );
lut->SetTableValue( 2, 0.0, 0.0, 1.0, 1.0 );
lut->SetTableValue( 3, 1.0, 1.0, 0.0, 1.0 );


vtkColorTransferFunction *lut = vtkColorTransferFunction::New();
lut->AddRGBPoint(   10.0, 1.0, 0.0, 0.0 );
lut->AddRGBPoint(   11.0, 0.0, 1.0, 0.0 );
lut->AddRGBPoint( 100.0, 0.0, 0.0, 1.0 );
```

Color Mapping



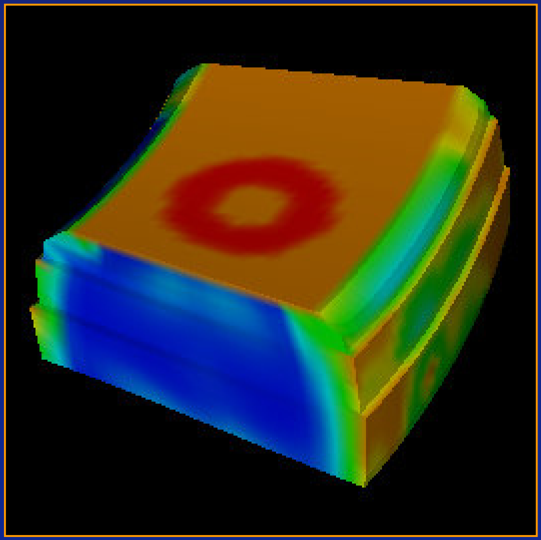




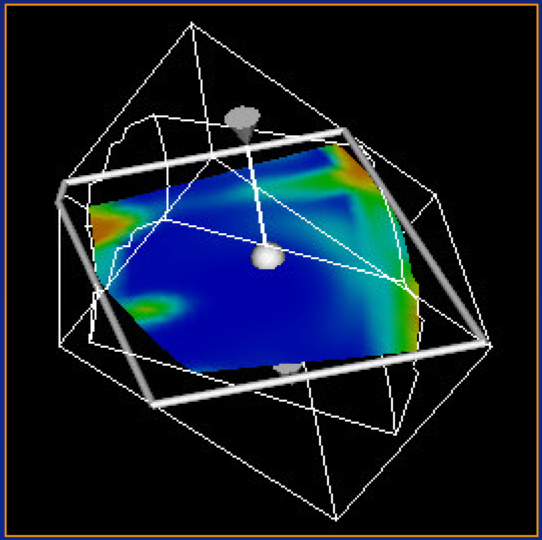


Cutting

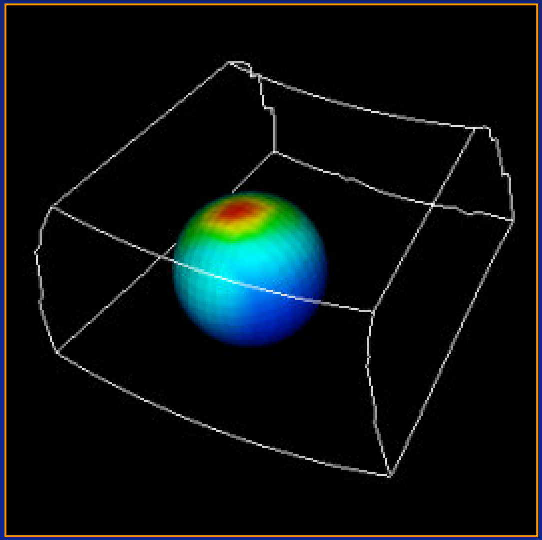
- To see a subset of the whole 3D scalar field, we can use vtkCutter to cut through the data set using an implicit function to extract a surface with interpolated scalar values.



outer surface



cut with plane



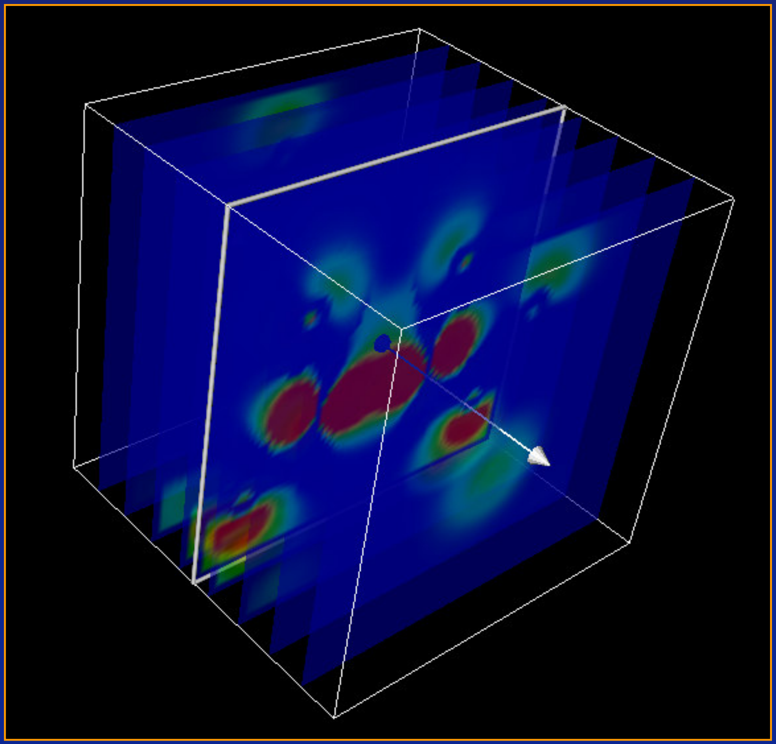
cut with sphere

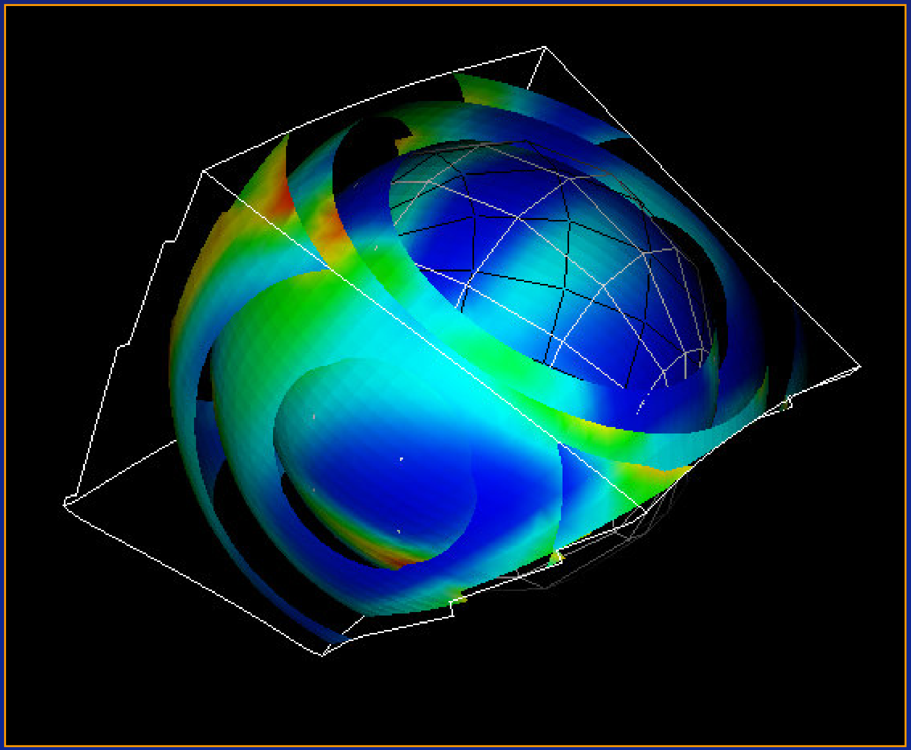
Cutting

```
// Create the implicit function
vtkPlane *plane = vtkPlane::New();
plane->SetOrigin( 0.25, 1.05, -4.27 );
plane->SetNormal( -0.287, 0.0, 0.9579 );

// Create the cutter and assign the input and function
vtkCutter *planeCut = vtkCutter::New();
planeCut->SetInput( reader->GetOutput() );
planeCut->SetCutFunction( plane );
```

Cutting

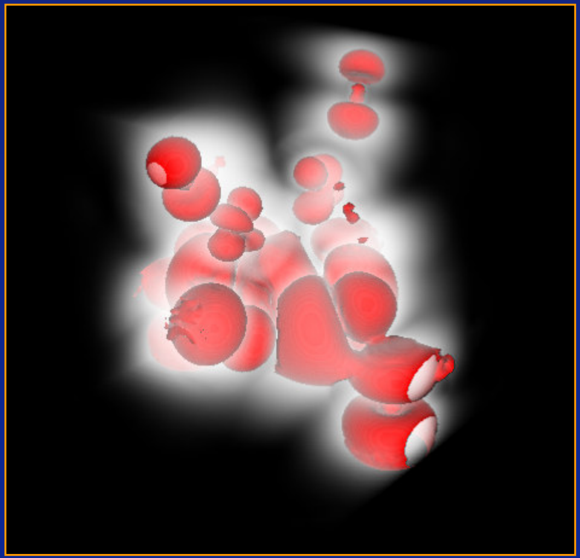
Implicit functions generate a scalar at each point in space. We can create multiple cut surfaces by cutting at set of scalar value locations, not just 0.





Contouring

- Contouring is the process of extracting isosurfaces from 3D input data, or isolines from 2D input data.
- VTK has a generic contour filter that operates on any data set: vtkContourFilter
- VTK also has two contour filters in the Patented kit: vtkMarchingCubes and vtkSynchronizedTemplates



Contouring Example

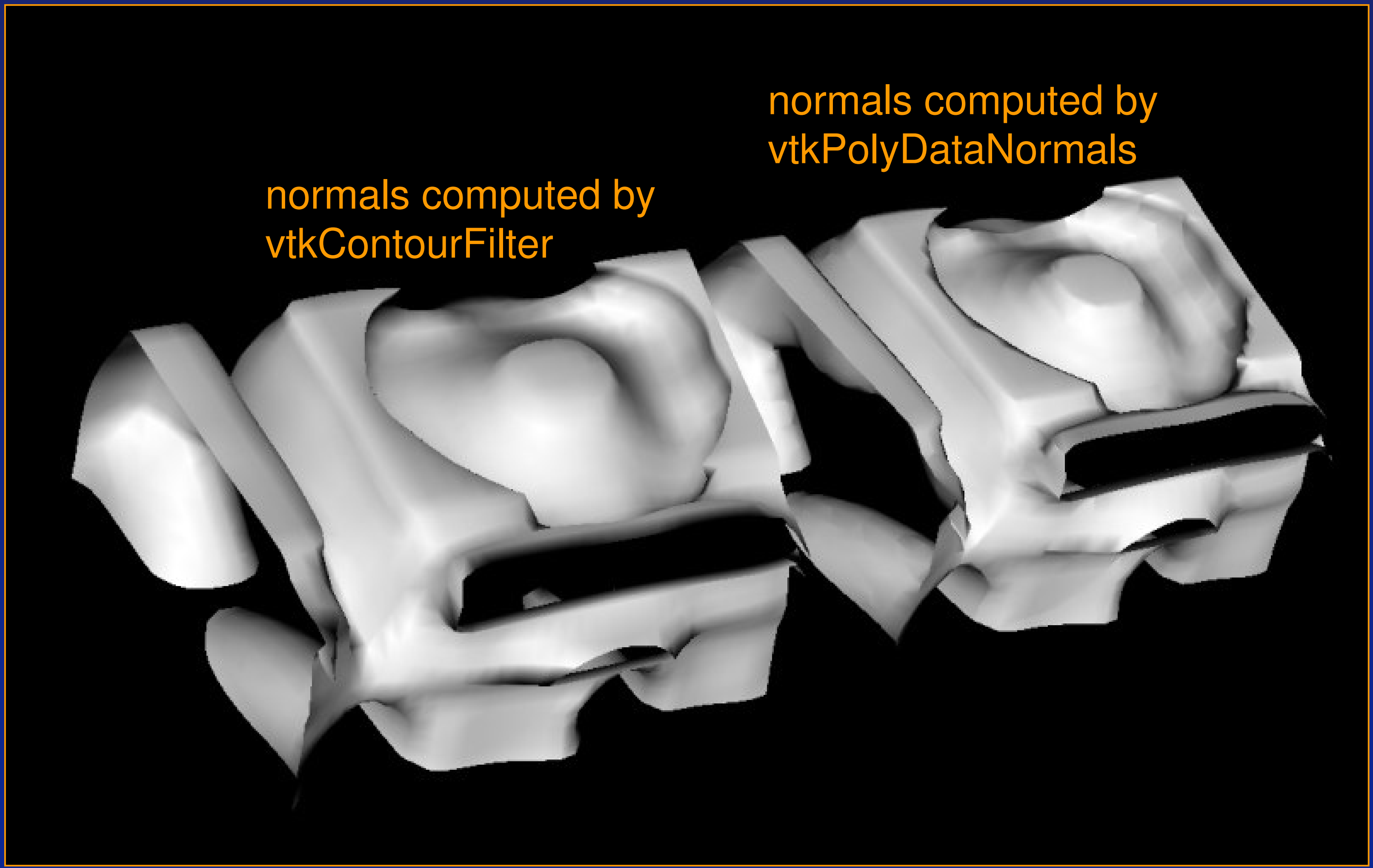
```
vtkContourFilter *skin = vtkContourFilter::New();
skin->SetInput( reader->GetOutput() );
skin->SetValue( 0, 500 );

vtkPolyDataMapper *mapper = vtkPolyDataMapper::New();
mapper->SetInput( skin->GetOutput() );
mapper->ScalarVisibilityOff();

vtkActor *actor = vtkActor::New();
actor->SetMapper( mapper );

renderer->AddProp( actor );
```

Contour Normals

There are two options for generating normals for the output contours:

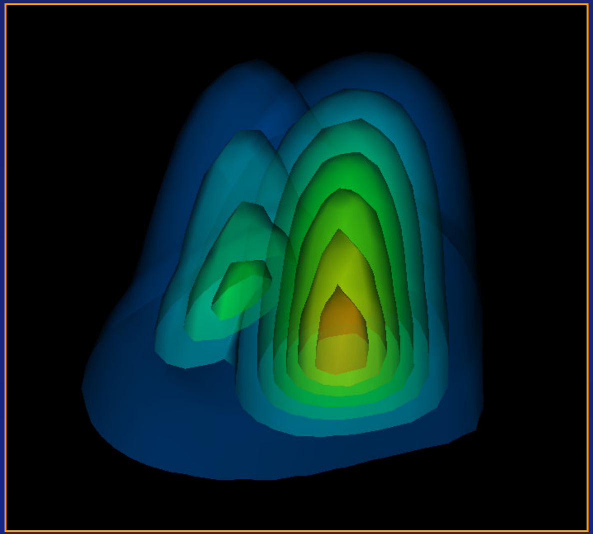
- vtkContourFilter has a ComputeNormalsOn() option – enabling this flag will cause the filter to assign normals based on the computed gradients in the input data. This will significantly increase the time required to extract a contour since gradient calculations are computationally expensive.
- The output from the vtkContourFilter can be passed through vtkPolyDataNormals to generate normals. This will be faster than computing the normals in the contour filter, but often has less visually pleasing results.

Contour Normals



Multiple Contours

- vtkContourFilter (and subclasses) can generate multiple isovalued contours in one output
- Scalar value can be used to color these contours
- Transparency can be used to display nested contours
- Use vtkDepthSortPolyData to sort the polygons for better translucent rendering



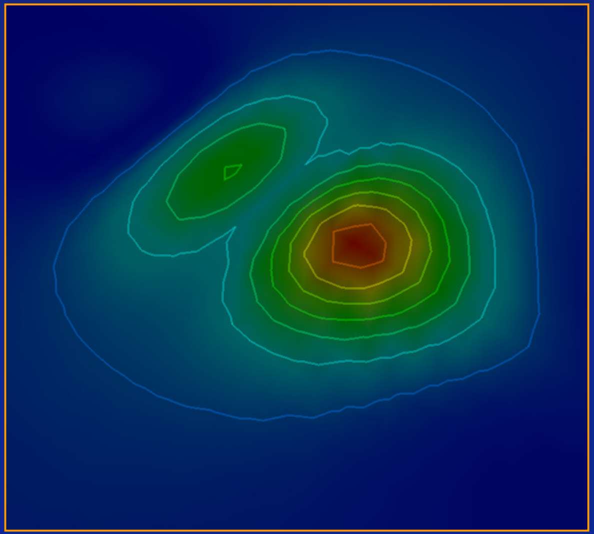
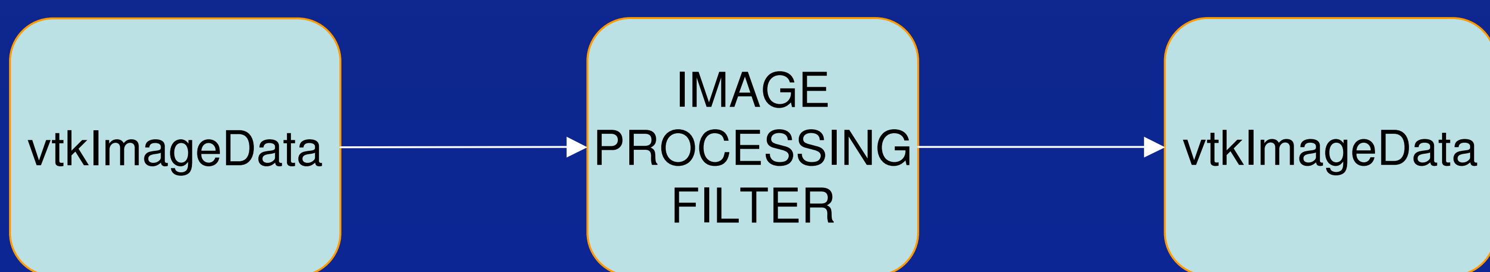


Image Processing

In visualization, image processing is used to manipulate the image content to improve the results of subsequent processing and interpretation. Example: remove noise from an image using a median filter before applying a contour filter.

Image processing filters take vtkImageData as input, and produce vtkImageData as output. The data may change dimensions, spacing, scalar type, number of components, etc.

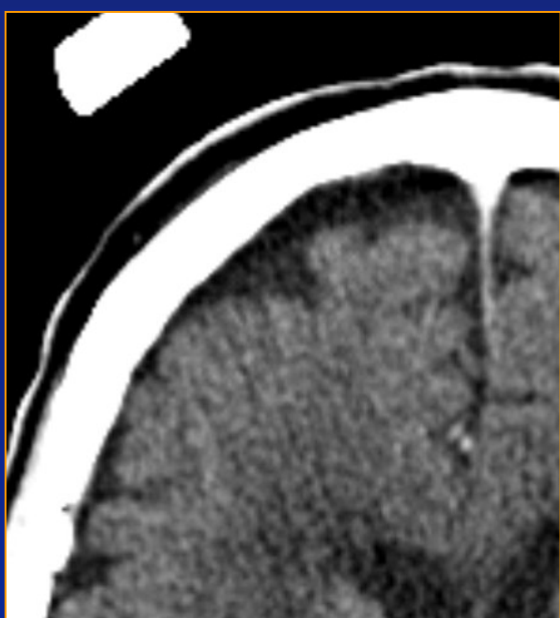


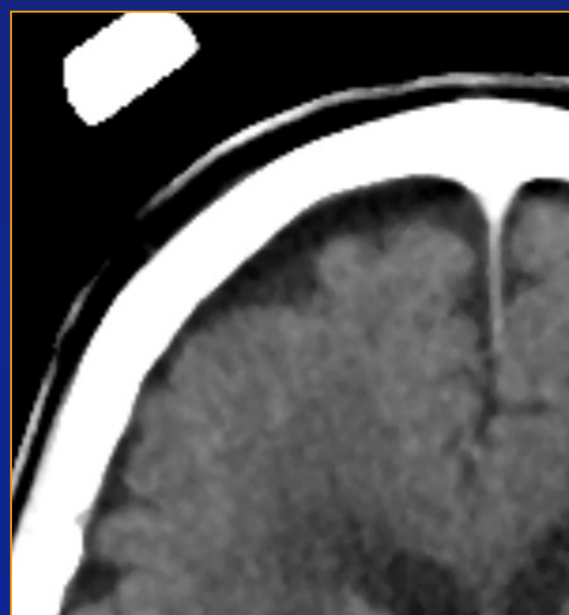
Partial List of Image Processing Filters

- AnisotropicDiffusion
- Blend
- Butterworth Low/HighPass
- CityBlockDistance
- Clip
- ConstantPad
- Correlation
- Difference
- DilateErode3D
- DotProduct
- Euclidean Distance
- Gradient Magnitude
- Ideal High/Low pass
- Laplacian
- Logic
- Luminance
- MapToColors
- Mask
- MaskBits
- Median3D
- NonMaximumSuppression
- OpenClose3D
- PadFilter
- Permute
- RectilinearWipe
- Resample
- Reslice
- ShiftScale
- Shrink 3D
- Threshold

Image Processing Example

```
vtkImageMedian3D *median =
   vtkImageMedian3D::New();
median->SetInput( reader->GetOutput() );
median->SetKernelSize( 3, 3, 3 );
median->Update();
```





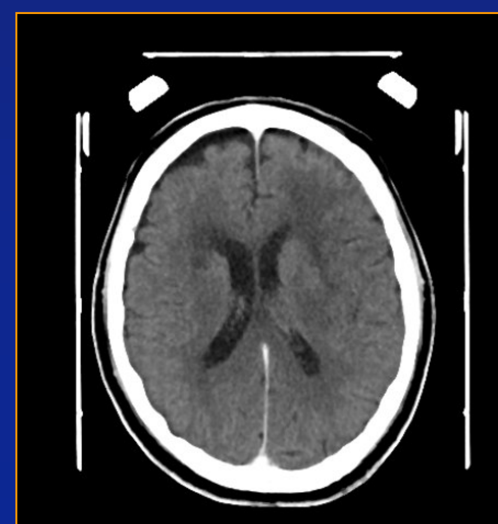


Image Processing

Speed Issues:

- The image processing filters can take some time to execute – for example when operating on large data using a large kernel size. Most filters provide progress information and can be aborted
- If multiple processors are available, VTK will make use of them in most imaging filters

Memory Issues:

- Most filters in VTK keep the input data intact, and create a new output data set. When working with large volumetric data sets this can be a problem, especially for filters that create internal buffers for intermediate results as well

Image Processing

- Internally, most image processing filters are templated to handle all data types from unsigned char through double:

```
switch (inData->GetScalarType())
  {
  vtkTemplateMacro7(vtkImageShrink3DExecute, this, inData,
                    (VTK_TT *)(inPtr), outData, (VTK_TT *)(outPtr),
                    outExt, id);
  default:
    vtkErrorMacro(<< "Execute: Unknown ScalarType");
    return;
  }
```

- Extensive segmentation and registration filters are available in The Insight Toolkit (ITK), which can be connected to VTK

Scalar Visualization – Part II

Lisa Avila
Kitware, Inc.

Overview

Volume Visualization in VTK:

- Definition and basic example
- Supported data types
- Available rendering methods
- Volume appearance
- Cropping and clipping
- Intermixing with geometric objects
- Achieving interactivity

Definitions

Volume rendering is the process of generating a 2D image from 3D data. The goal of volume visualization is to generate informative images from 3D data using a variety of techniques including volume rendering.

The line between volume rendering and geometric rendering is not always clear. Volume rendering may produce an image of an isosurface, or may employ geometric hardware for rendering.

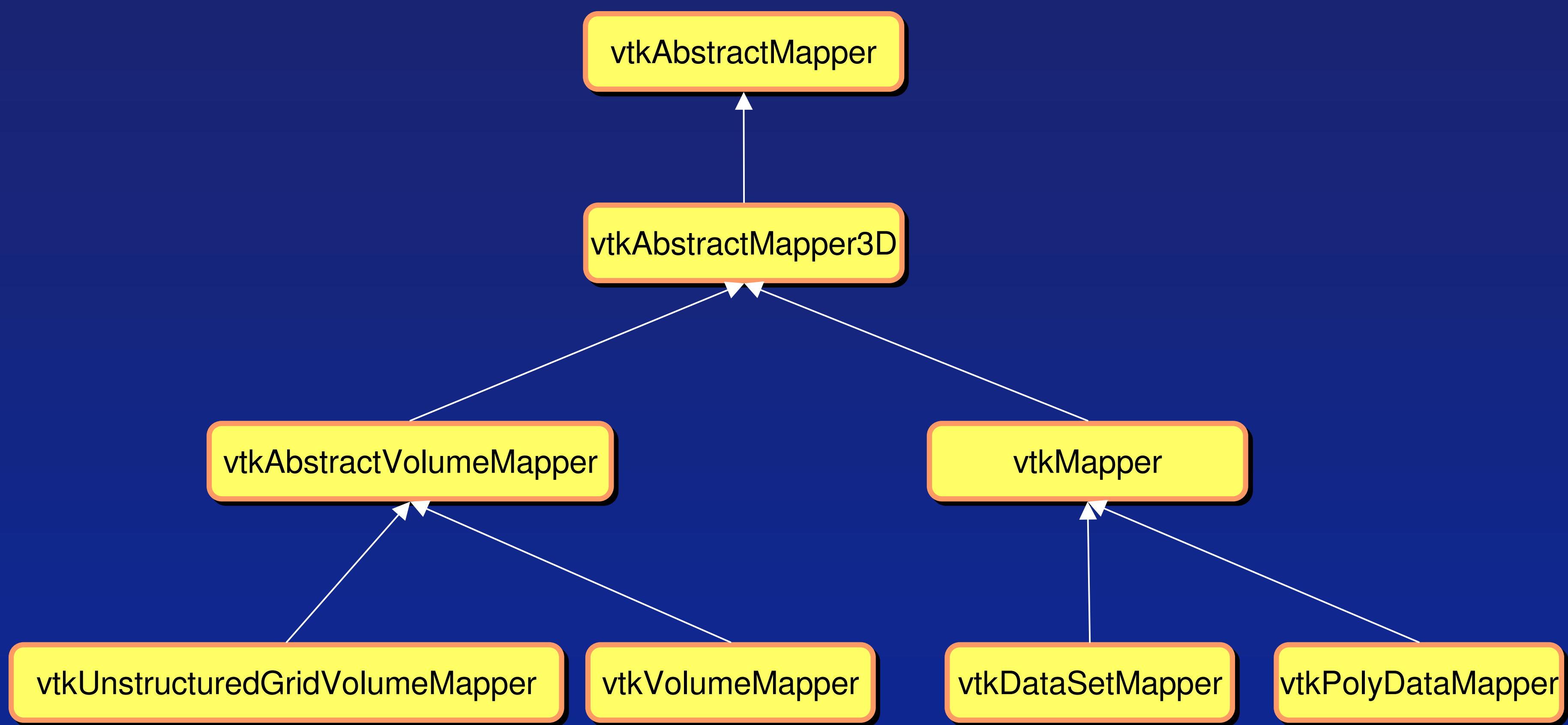
Basic Example

```
vtkVolumeTextureMapper2D *mapper =
   vtkVolumeTextureMapper2D::New();

mapper->SetInput( reader->GetOutput() );

vtkVolume *volume = vtkVolume::New();
volume->SetMapper( mapper );
volume->SetProperty( property);

renderer->AddProp( volume );
```

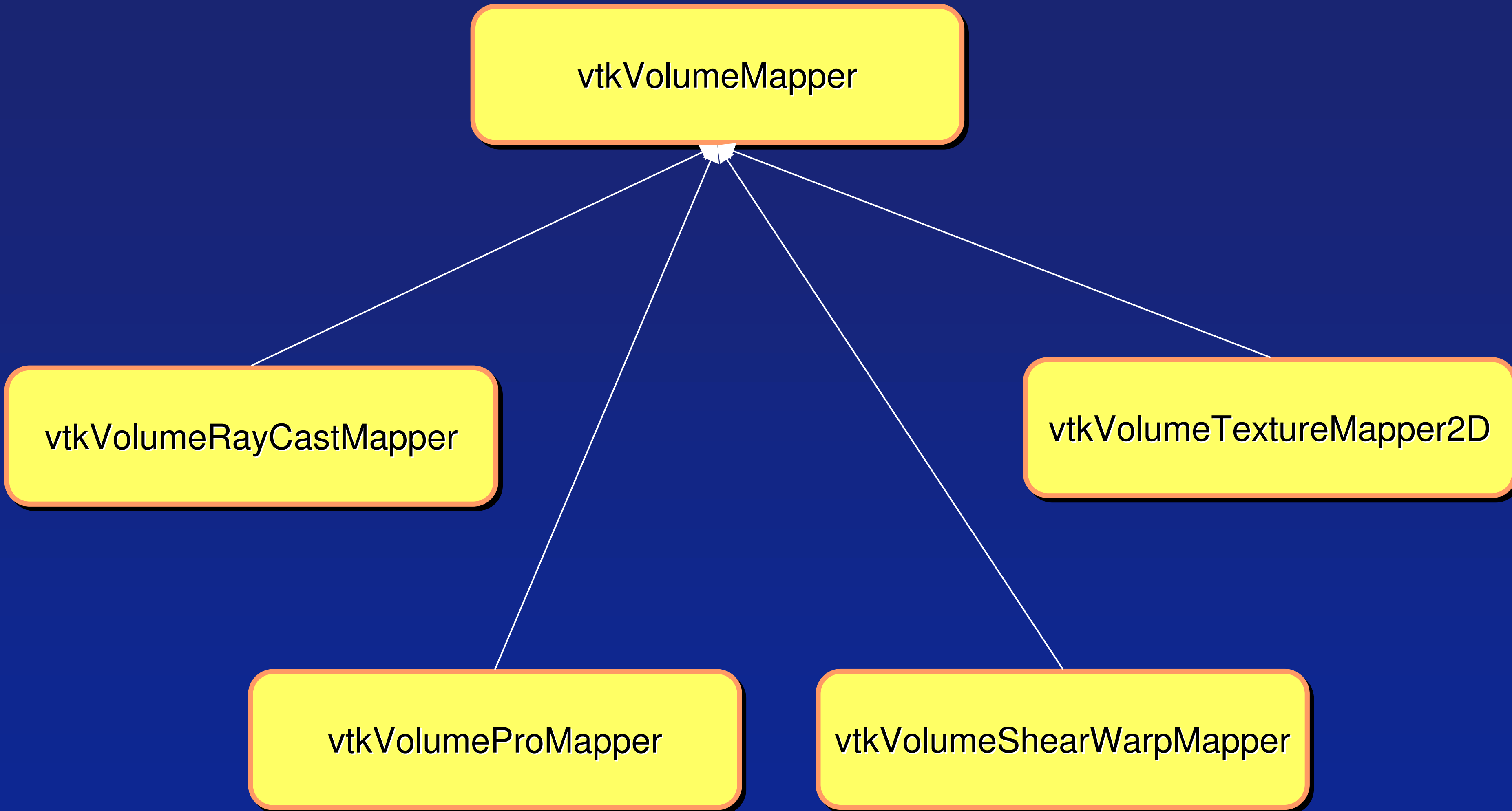
Basic Example

```
vtkPiecewiseFunction *opacity =
   vtkPiecewiseFunction::New();
opacity->AddPoint( 0, 0.0 );
opacity->AddPoint( 255, 1.0 );

vtkColorTransferFunction *color =
   vtkColorTransferFunction::New();
color->AddRGBPoint( 0, 0.0, 0.0, 1.0 );
color->AddRGBPoint( 255, 1.0, 0.0, 0.0 );

vtkVolumeProperty *property = vtkVolumeProperty::New();
property->SetOpacity( opacity );
property->SetColor( color );
```

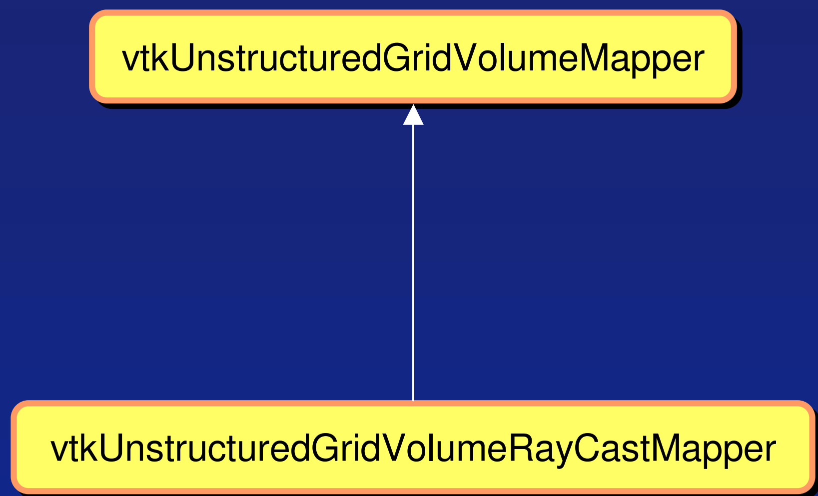
Inheritance Diagram



Inheritance Diagram

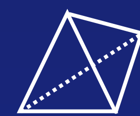


Inheritance Diagram



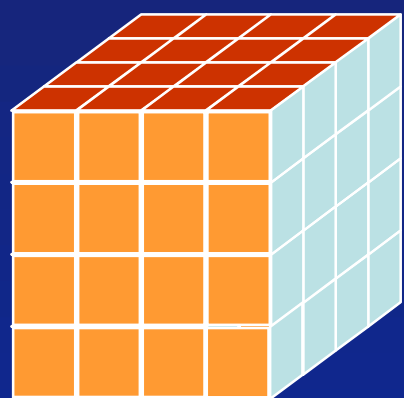
Supported Data Types

vtkUnstructuredGrid



- vtkTetra cells
- point data scalars will be rendered

vtkImageData

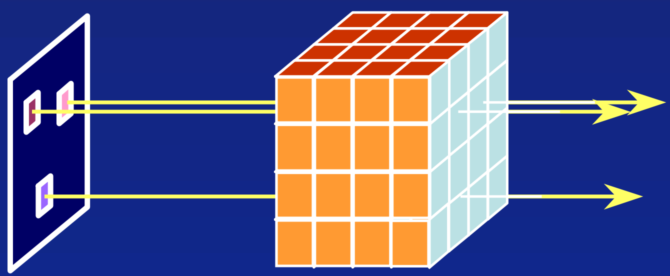


- unsigned char or unsigned short
- one component
- point scalars data will be rendered

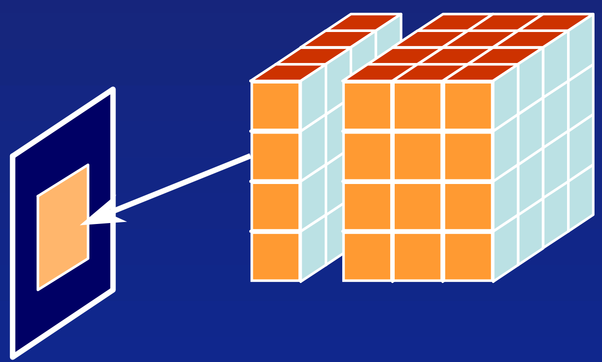
Volume Rendering Strategies

Image-Order Approach:
Traverse the image pixel-by-pixel and sample the volume via ray-casting.

Object-Order Approach:
Traverse the volume, and project onto the image plane.



Ray Casting



2D Texture Mapping

Hybrid Approach: Combine the two techniques.

Available Methods

For vtkImageData:

- vtkVolumeRayCastMapper
- vtkVolumeTextureMapper2D
- vtkVolumeProMapper

For vtkUnstructuredGrid

- vtkUnstructuredGridVolumeRayCastMapper

Why Multiple Methods?

Volume rendering is not a "solved" problem – there is no single "right way" to render a volume. Multiple methods allow the user to choose the method that is right for the application. The VTK volume rendering framework allows researchers to develop new algorithms.

Ray Casting: image-order method that is implemented in software with excellent quality but slow speed

2D Texture Mapping: an object-order method with lower quality due to limited frame buffer depth, but fast due to hardware acceleration

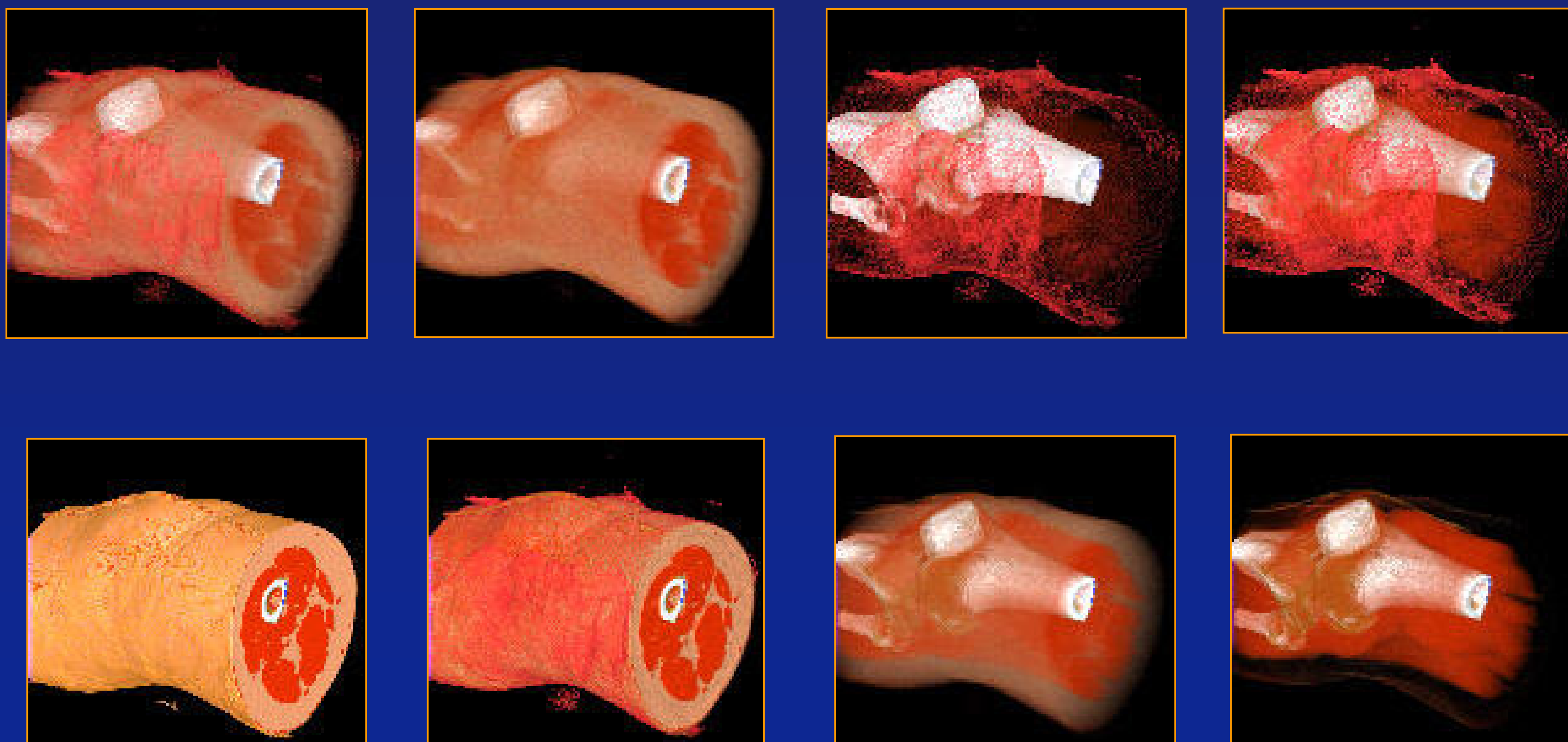
VolumePro: dedicated hardware for volume rendering

Volume Appearance

- Mapping from scalar to color
- Mapping from scalar to opacity
- Mapping from gradient magnitude to opacity modulator
- Ray function
- Shading
- Interpolation
- Sample distance

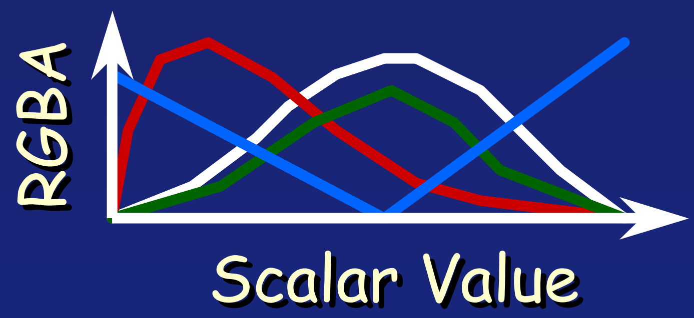
Material Classification

Transfer functions are the key to effective volume rendering



Material Classification

Scalar value can be classified into color and opacity (RGBA)

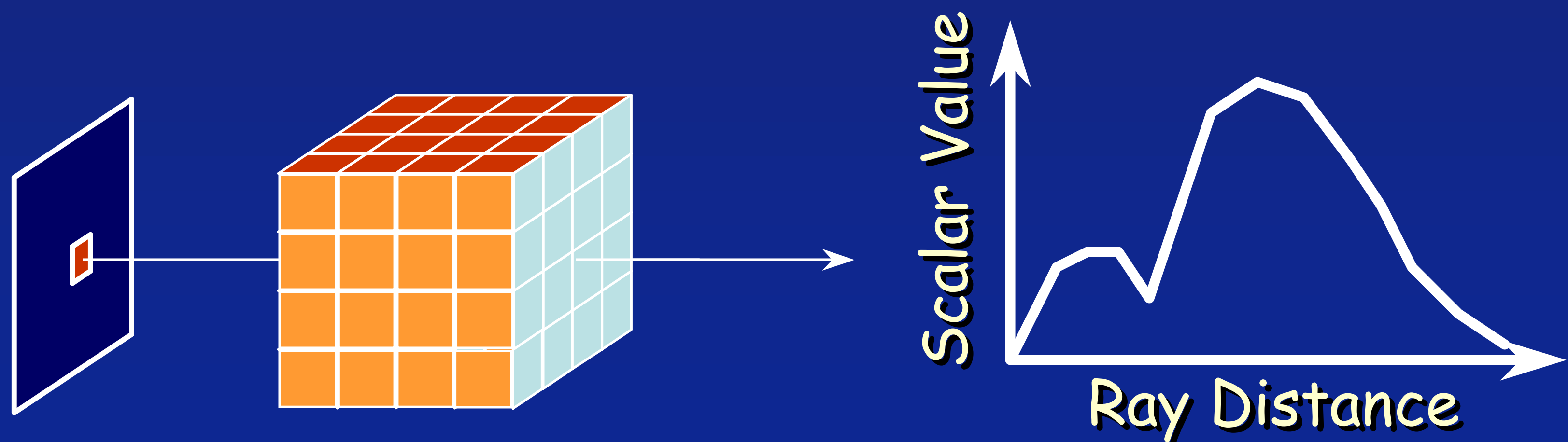


Gradient magnitude can be classified into opacity

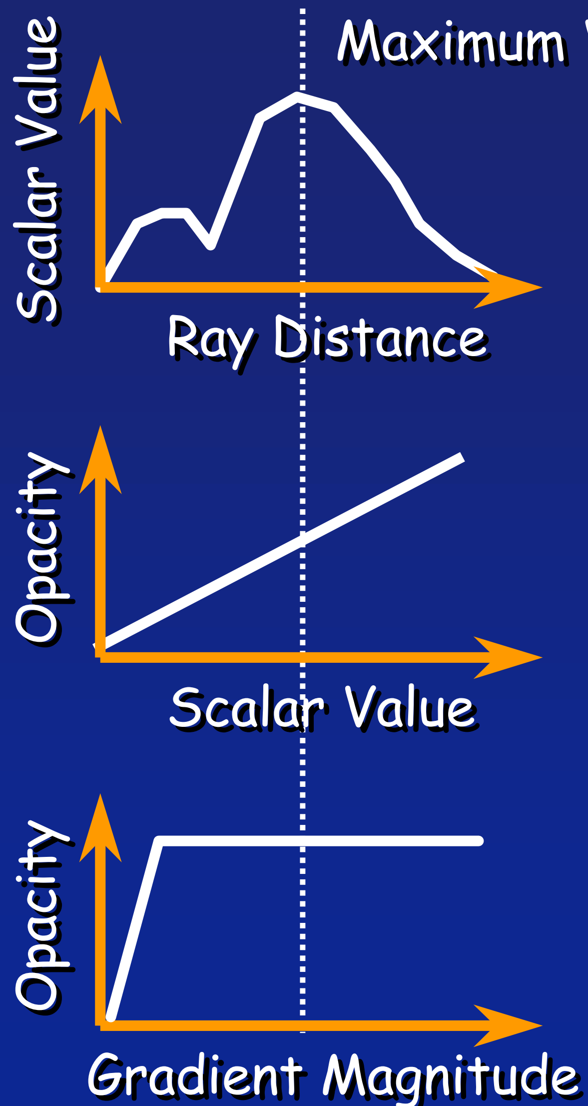
Final opacity is obtained by multiplying scalar value opacity by gradient magnitude opacity

Ray Functions

A Ray Function examines the scalar values encountered along a ray and produces a final pixel value according to the volume properties and the specific function.

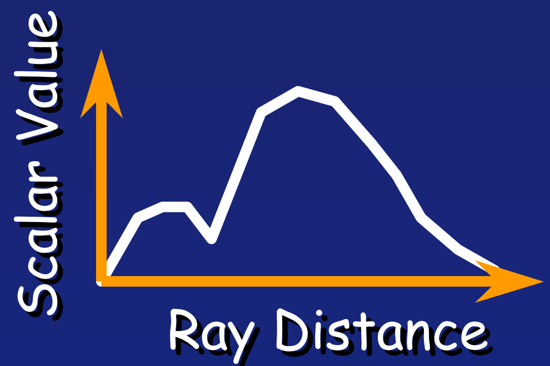


Maximum Intensity Function



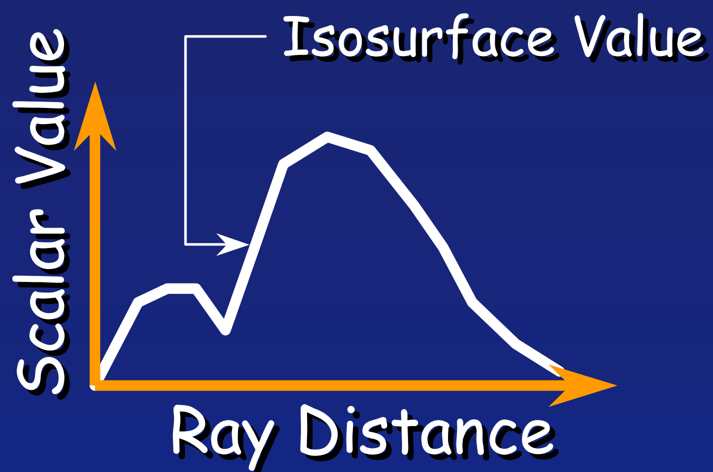
Maximize Scalar Value

Composite Function

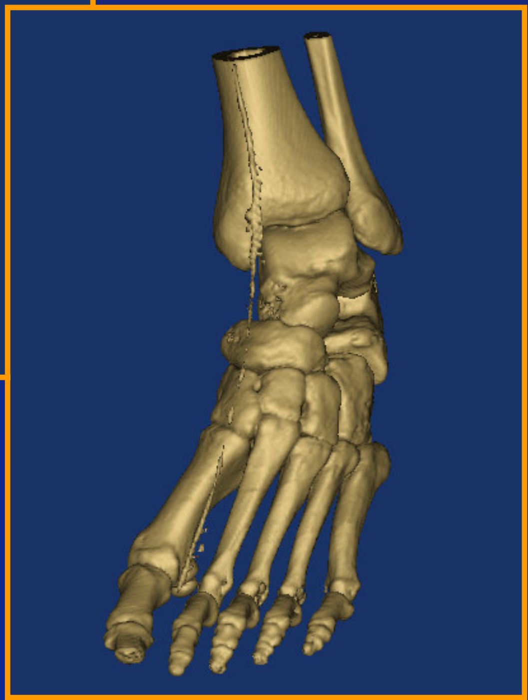


Use α-blending along the ray to produce final RGBA value for each pixel.

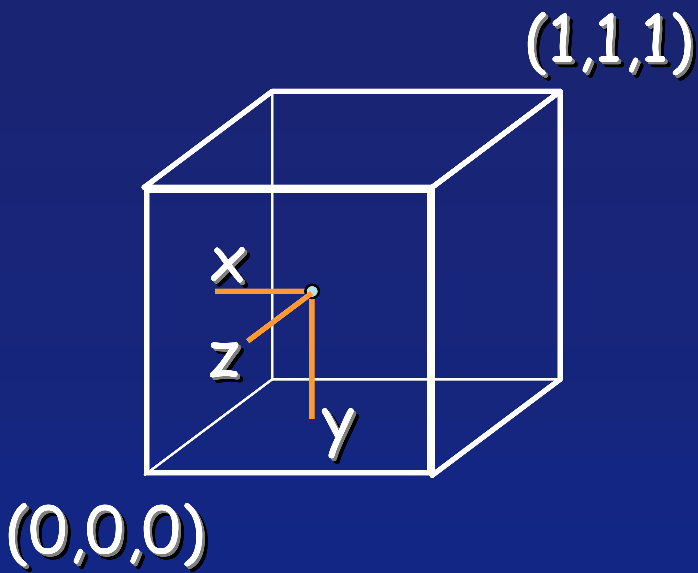
Isosurface Function



Stop ray traversal at isosurface value. Use cubic equation solver if interpolation is trilinear.



Scalar Value Interpolation



v = S(rnd(x), rnd(y), rnd(z))

Nearest Neighbor

v = (1-x)(1-y)(1-z)S(0,0,0) +
(x)(1-y)(1-z)S(1,0,0) +
(1-x)(y)(1-z)S(0,1,0) +
(x)(y)(1-z)S(1,1,0) +
(1-x)(1-y)(z)S(0,0,1) +
(x)(1-y)(z)S(1,0,1) +
(1-x)(y)(z)S(0,1,1) +
(x)(y)(z)S(1,1,1)

Trilinear

Using Ray Functions

Ray functions are used only by vtkVolumeRayCastMapper.

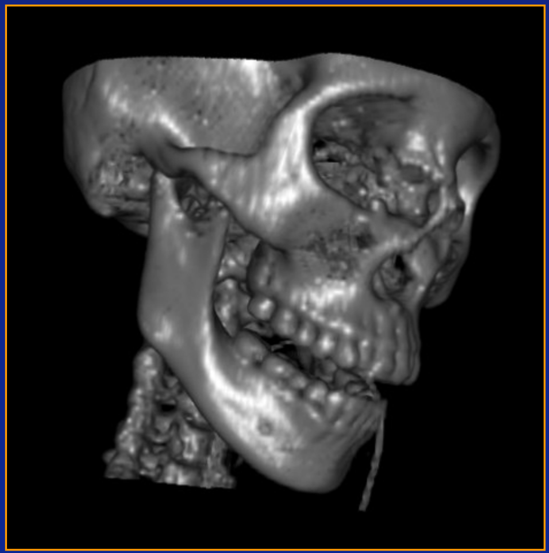
```
// Create the isosurface function and set the isovalue
vtkVolumeRayCastIsosurfaceFunction *func =
  vtkVolumeRayCastIsosurfaceFunction::New();
func->SetIsoValue( 27.0 );

// Create the mapper and set the ray cast function
vtkVolumeRayCastMapper *mapper =
    vtkVolumeRayCastMapper::New();
mapper->SetVolumeRayCastFunction( func );
```

Shading

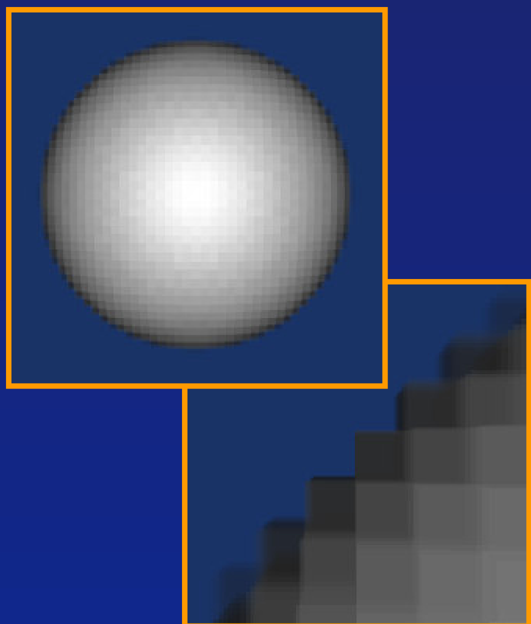
- Shading can aid in understanding the shape of the data
- Normals are derived using scalar value gradients
- Set Ambient, Diffuse, Specular and Specular Power
- Turn shading on

Shading

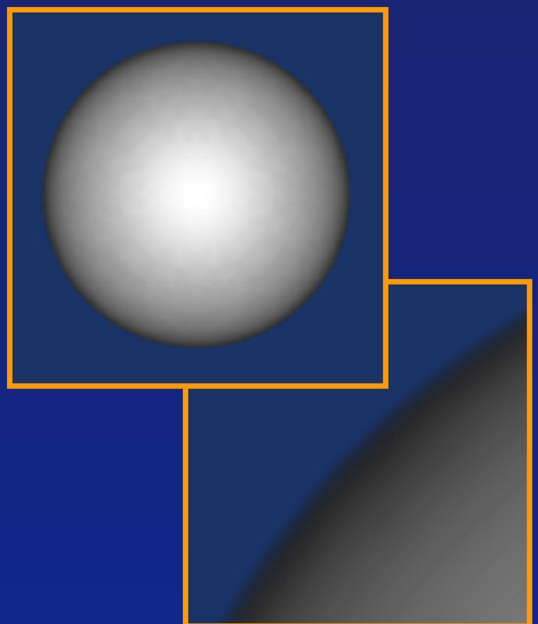
```
vtkVolumeProperty *property = vtkVolumeProperty::New();
property->SetAmbient( 0.0 );
property->SetDiffuse( 0.9 );
property->SetSpecular( 0.2 );
property->SetSpecularPower( 10.0 );
property->ShadeOn();
```



Scalar Value Interpolation

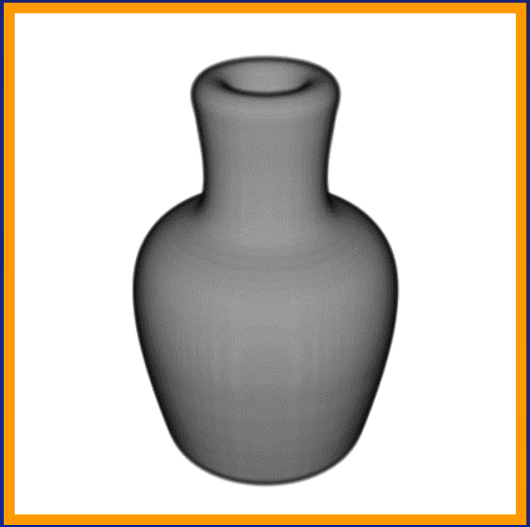


Nearest Neighbor
Interpolation



Trilinear
Interpolation

Sampling Distance



0.1 Unit Step Size

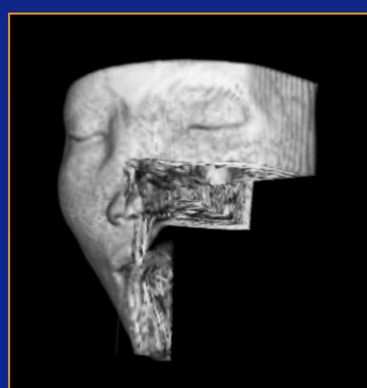
1.0 Unit Step Size

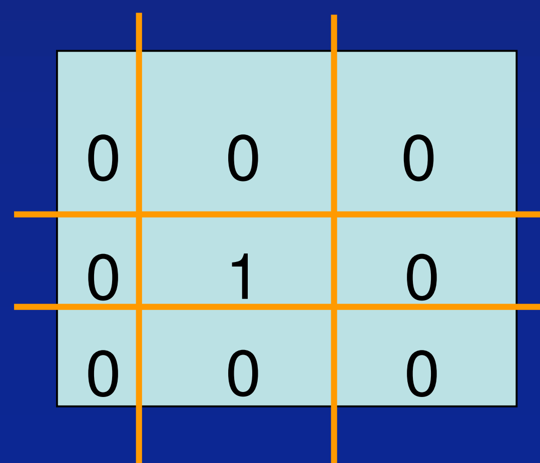
2.0 Unit Step Size

Cropping

- Two planes along each major axis divide the volume into 27 regions
- A flag indicates which regions are on / off
- Commonly used for subvolume and to cut out a corner

```
volumeMapper->CroppingOn();
volumeMapper
   ->SetCroppingRegionPlanes( 17, 21, 13, 27, 0, 26 );
volumeMapper->SetCroppingRegionFlagsToSubVolume();
```

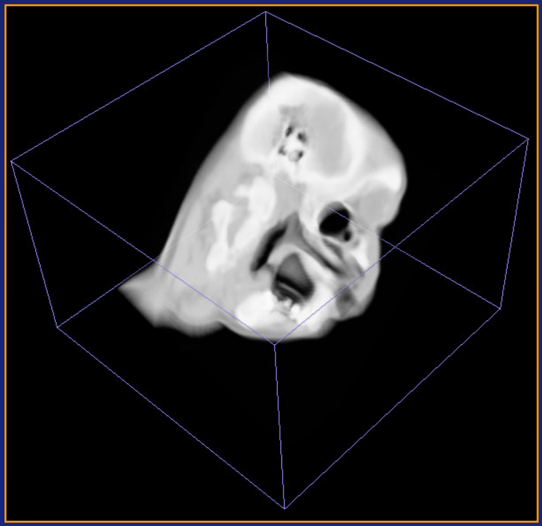


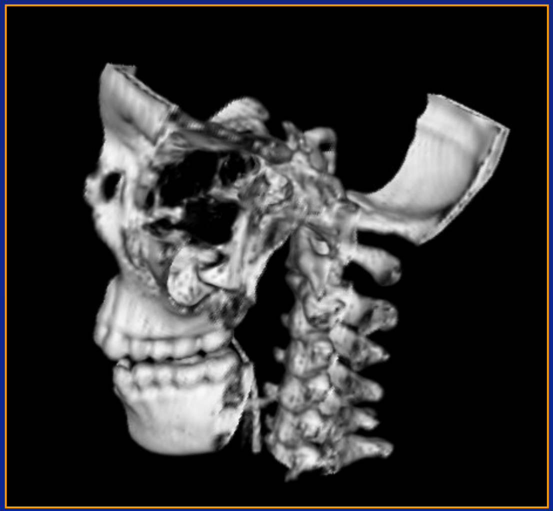


Clipping

- Up to six (more for ray casting) infinite planes can be used for clipping the data
- Two parallel planes can be used to create a "thick slab" rendering.

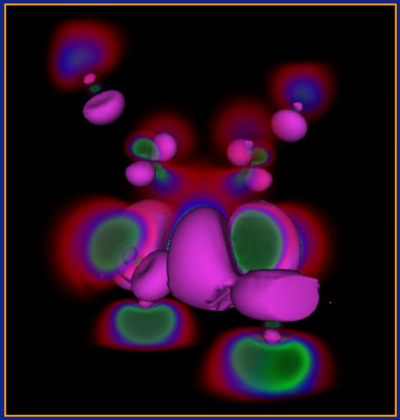
```
vtkPlane *plane1 = vtkPlane::New();
plane1->SetOrigin( 25, 25, 20 );
plane1->SetNormal( 0, 0, 1 );

vtkPlane *plane2 = vtkPlane::New();
plane2->SetOrigin( 25 25 30 );
plane2->SetNormal( 0 0 -1 );

volumeMapper->AddClippingPlane( plane1 );
volumeMapper->AddClippingPlane( plane2 );
```



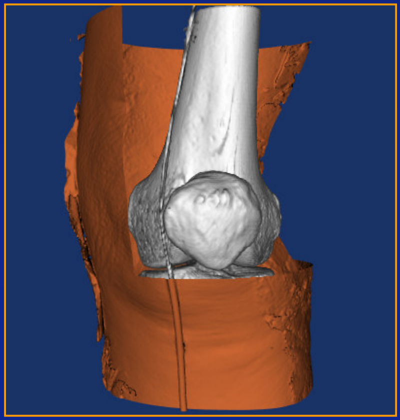


Intermixing

- Volumes can be intermixed with opaque geometry
- Multiple volumes and translucent geometry can be rendered in one scene, but each object must not overlap any other non-opaque object, and the vtkFrustumCoverageCuller must be used to sort the objects back-to-front before rendering.



High potential
iron protein

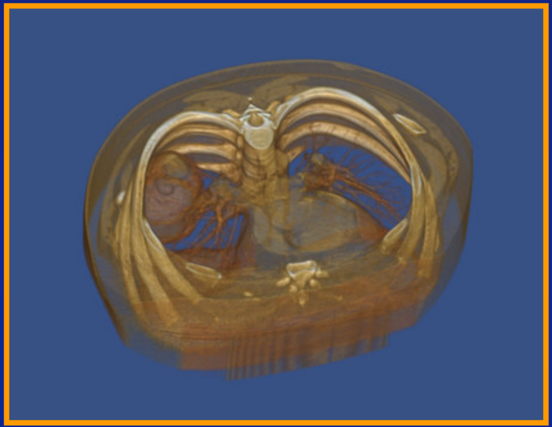


CT scan of the
visible woman's knee

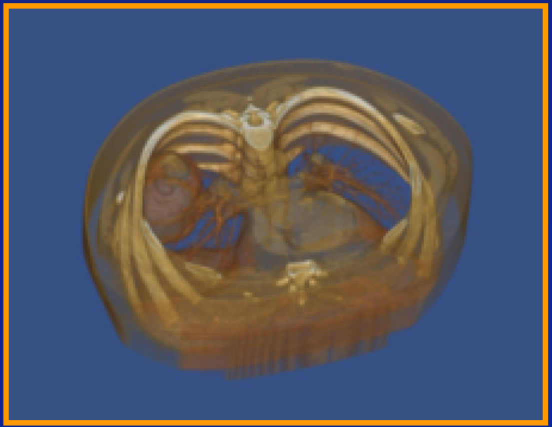
Interactivity

- vtkVolumeRayCastMapper will adjust the number of rays cast based on the AllocatedRenderTime of the vtkVolume.
- AllocatedRenderTime is set based on DesiredUpdateRate of the vtkRenderWindow.

Multi-resolution ray casting:



1x1 Sampling



2x2 Sampling

4x4 Sampling

Interactivity

- Create Levels-Of-Detail (LODs) using reduced resolution data or a faster rendering technique
- Use a vtkLODProp3D to hold the LODs
- Both geometric rendering and volume rendering can be intermixed within one vtkLODProp3D