

**Rensselaer**

**ESCE 4960: Open Source Software Practice  
Lecture 8: VTK Pipeline  
September 24, 2007**

**Dr. Will Schroeder, Kitware**

# Interpreters

VTK provides automatic wrapping for the following interpreted languages:

- Tcl
- Java
- Python

*Interpreters provide faster turn-around (no compilation) but suffer from slower execution*

# Tcl Interpreter

To use VTK from Tcl, add the following line to the beginning of your script:

```
package require vtk
```

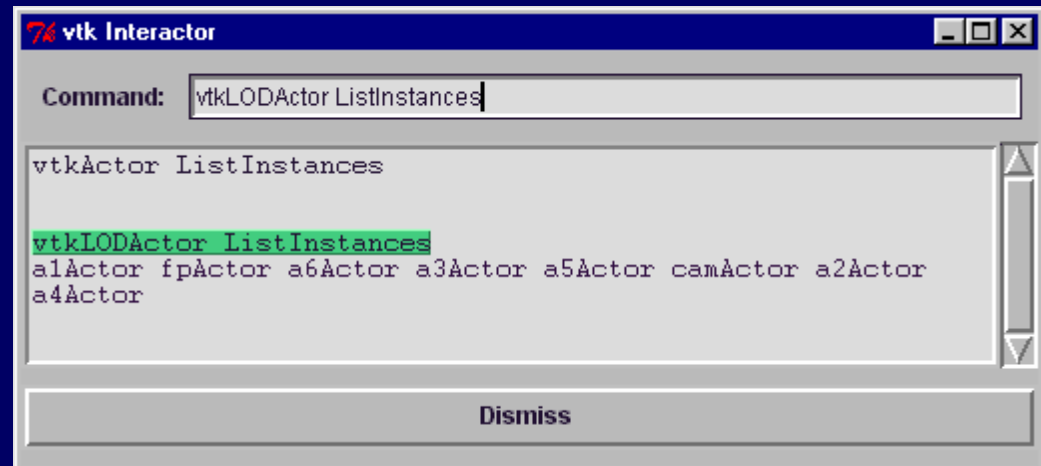
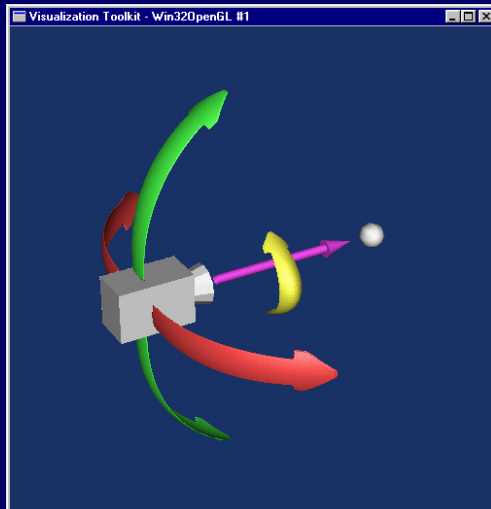
Create an actor in Tcl: `vtkActor actor`

Invoke a method: `actor SetPosition 10 20 30`

# Tcl Interpreter

A special package provides a Tcl interpreter when the 'u' key is pressed in the render window:

```
package require vtkinteraction  
iren AddObserver UserEvent {wm deiconify .vtkInteract}
```



# Tcl Interpreter

`vtkActor ListInstances`: list all `vtkActor` objects

`vtkActor ListMethods`: list all `vtkActor` methods

`anActor Print`: print internal state of `anActor`

`vtkCommand DeleteAllObjects`: delete all VTK objects

`vtkTkRenderWindow`: embed a render window in Tk

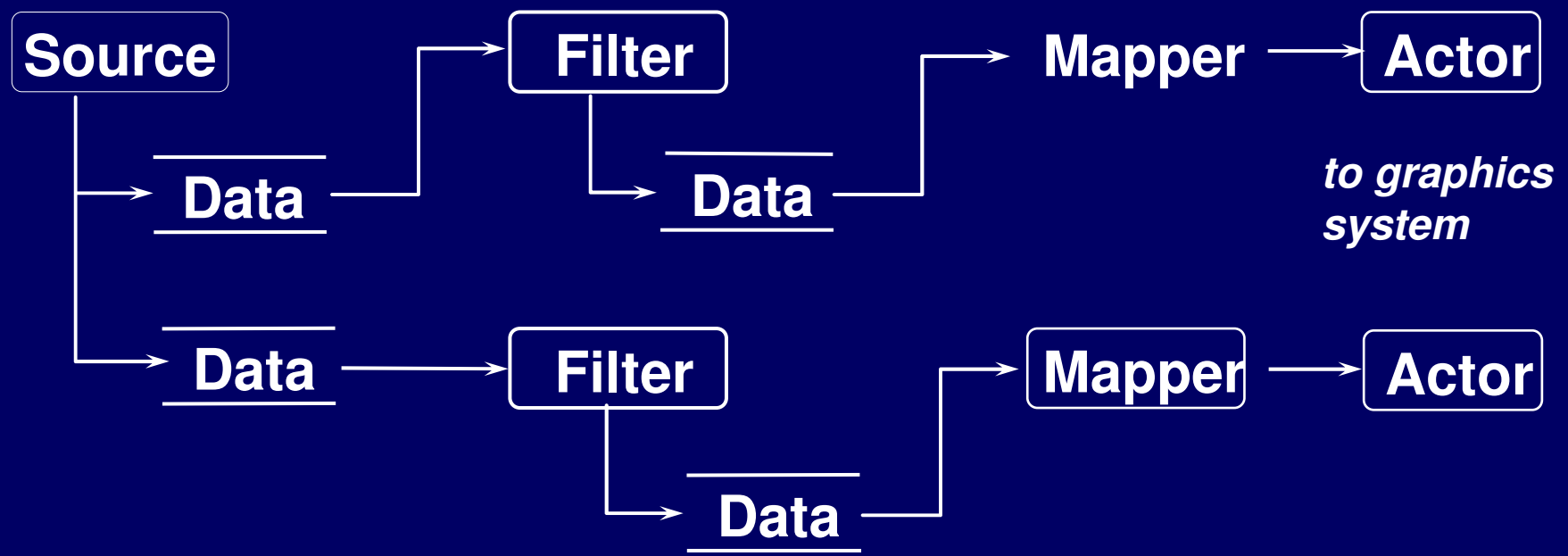
`vtkTkImageViewerWidget`: embed an image window in Tk

## Exercise 2

- Run an example `exercise2.tcl`
- Use 'u' (user-defined) key to bring up the interactor interface
- Try some commands
  - `vtkLODActor ListInstances`
  - `sphereActor Print`
  - `sphereActor ListMethods`

# The Visualization Pipeline

A sequence of algorithms that operate on data objects to generate geometry that can be rendered by the graphics engine or written to a file



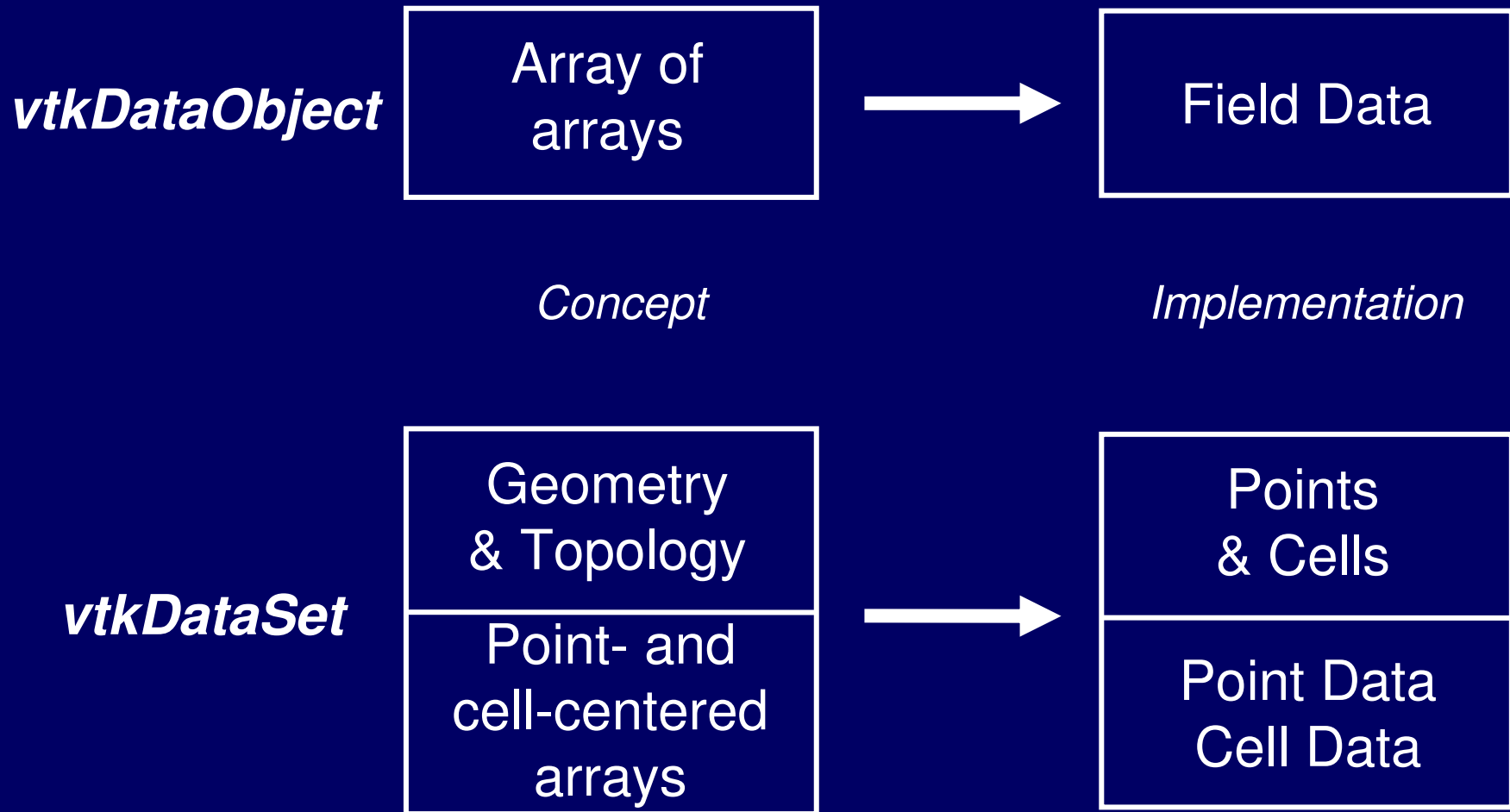
# Visualization Model

- Data Objects
  - represent data
  - provide access to data
  - compute information particular to data (e.g., bounding box, derivatives)
- Algorithms
  - Ingest, transform, and output data objects

# vtkDataObject / vtkDataSet

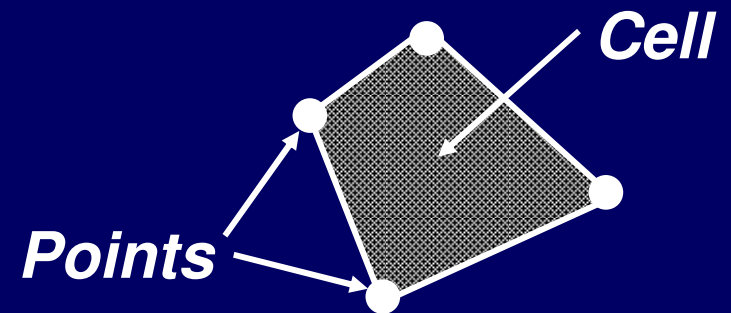
- vtkDataObject represents a “blob” of data
  - contain instance of vtkFieldData
  - an array of arrays
  - no geometric/topological structure
  - Superclass of all VTK data objects
- vtkDataSet has geometric/topological structure
  - Consists of geometry (points) and topology (cells)
  - Has associated point- and cell-centered data arrays
  - Convert data object to data set with vtkDataObjectToDataSetFilter

# vtkDataObject / vtkDataSet



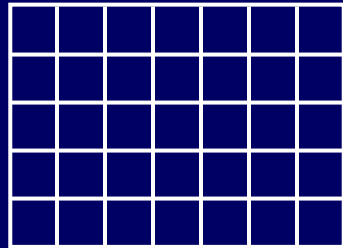
# Dataset Model

- A dataset is a data object with structure
- Dataset structure consists of
  - points (x-y-z coordinates)
  - cells (e.g., polygons, lines, voxels) which are defined by connectivity list referring to points ids
  - Access is via integer ID
  - implicit representations
  - explicit representations

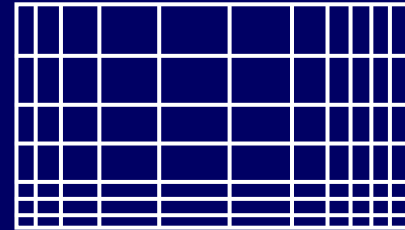


# vtkDataSet Subclasses

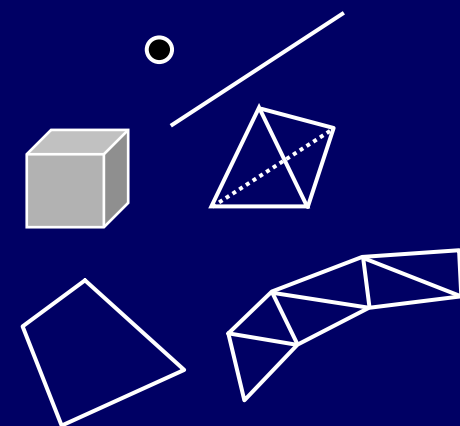
**vtkImageData**



**vtkRectilinearGrid**



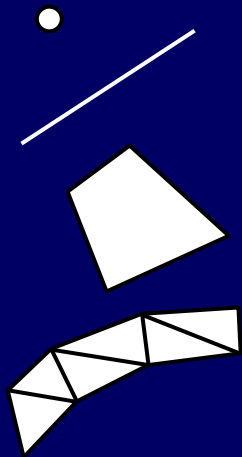
**vtkUnstructuredGrid**



**vtkStructuredGrid**



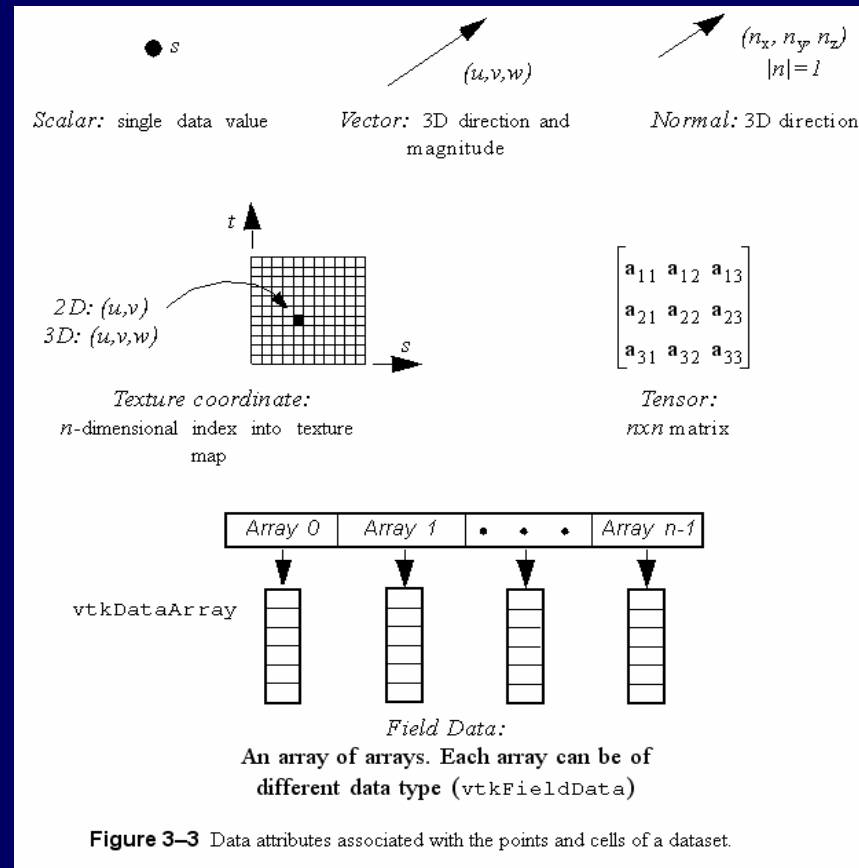
**vtkPolyData**



# Data Set Attributes

- **vtkDataSet** also has point and cell attribute data:
  - **Scalars**
  - **Vectors** - 3-vector
  - **Tensors** - 3x3 symmetric matrix
  - **Normals** - unit vector
  - **Texture Coordinates** 1-3D
  - **Array of arrays** (I.e. FieldData)

# Data Set Attributes (cont.)

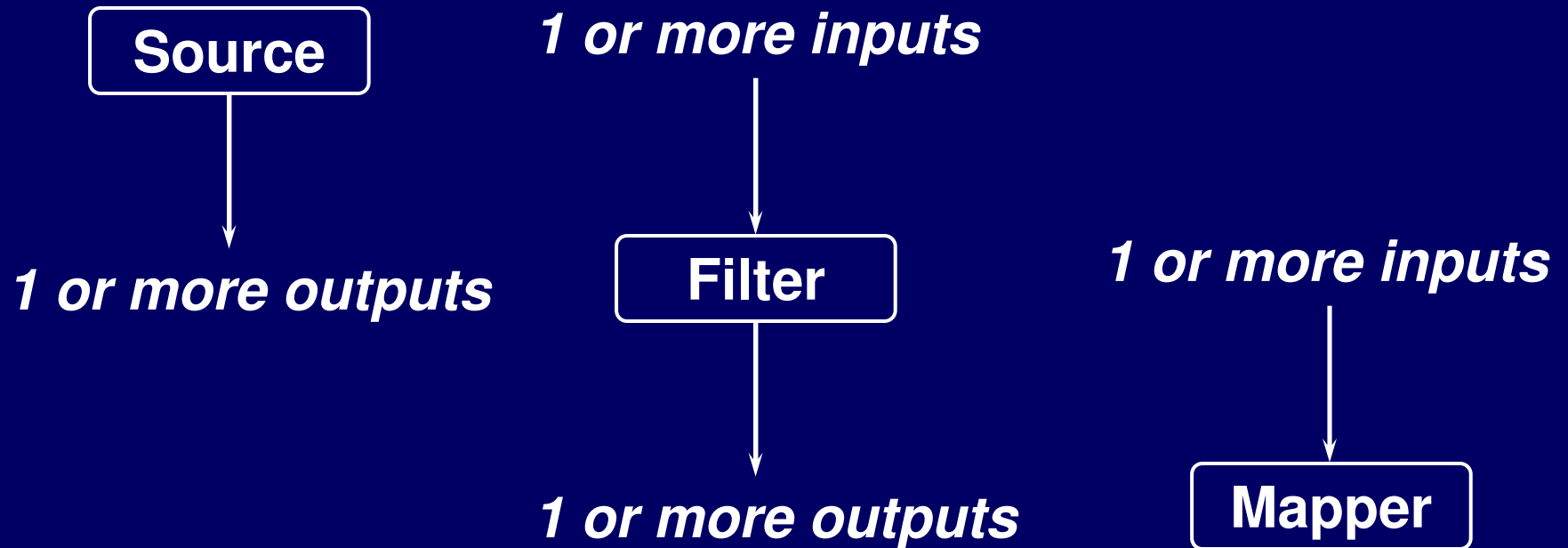


# Scalars (An Aside)

- Scalars are represented by a `vtkDataArray`
- Scalars are typically single valued
- Scalars can also represent color
  - I (intensity)
  - IA (intensity-alpha: *alpha is opacity*)
  - RGB (red-green-blue)
  - RGBA (RGB + alpha)
- Scalars can be used to generate colors
  - mapped through lookup table
  - if unsigned char → direct color specification

# Algorithms

- Algorithms operate on data objects



# Pipeline Execution Model

*(conceptual depiction)*

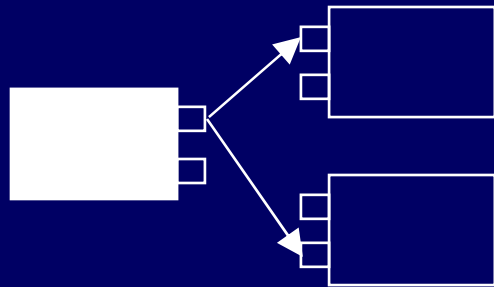
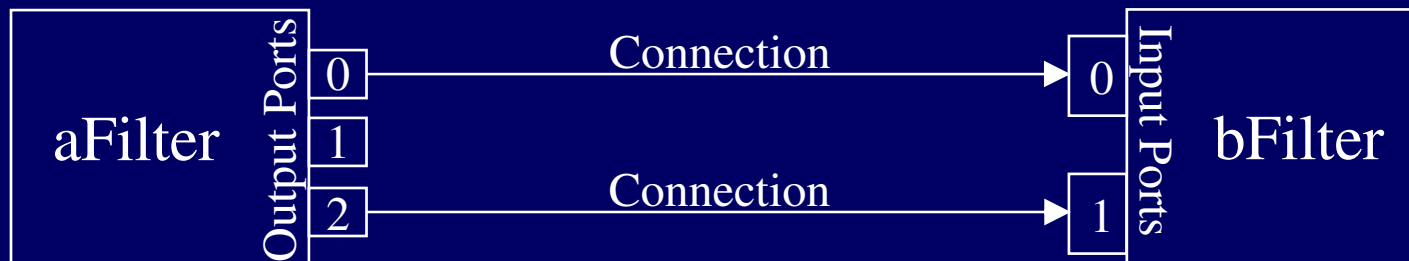
*direction of data flow (via RequestData())*



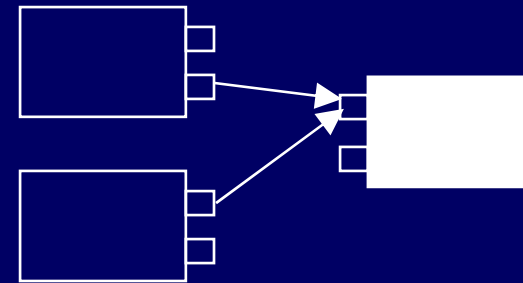
*direction of update (via Update())*

# Creating Pipeline Topology

- `bFilter->SetInputConnection(aFilter->GetOutputPort());`
- `bFilter->SetInputConnection(1,aFilter->GetOutputPort(2));`



Reuse an output port: OK



Several connections on an input port:  
`AddInputConnection()` if allowed by  
the filter (ex: `vtkAppendFilter`)

# Role of Type-Checking

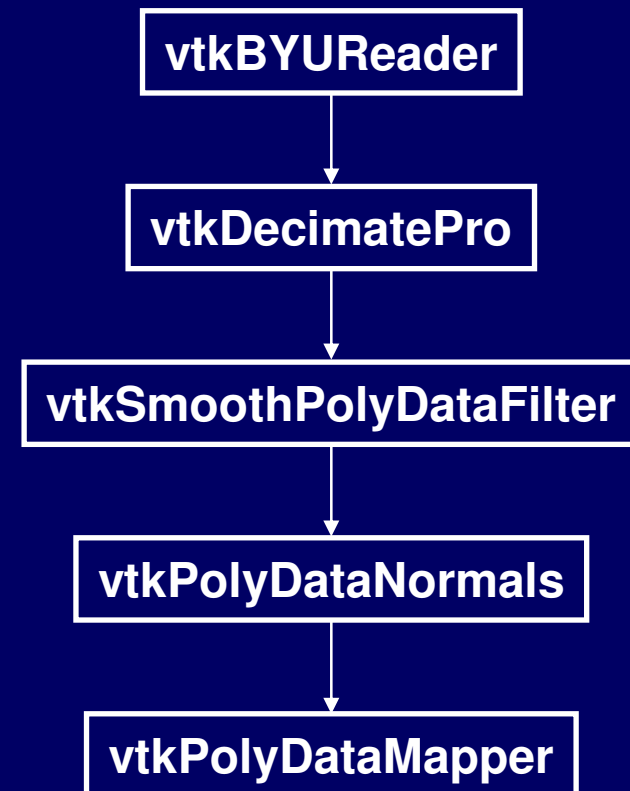
- FillInputPortInformation() specifies input dataset type
- FillOutputPortInformation() specifies output dataset type
- Type-checking performed at run-time

```
int vtkPolyDataAlgorithm::FillInputPortInformation(  
    int vtkNotUsed(port),  
    vtkInformation *info)  
{  
    info->Set(vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE(),  
            "vtkPolyData");  
    return 1;  
}
```

# Example Pipeline

- Decimation, smoothing, normals
- Implemented in C++

***Note: data objects are not shown → they are implied from the output type of the filter***



# Create Reader & Decimator

```
vtkBYUReader *byu = vtkBYUReader::New();
    byu->
SetGeometryFileName("../..//vtkdata/fran_cut.g");

vtkDecimatePro *deci = vtkDecimatePro::New();
    deci->SetInputConnection( byu->GetOutputPort() );
    deci->SetTargetReduction( 0.9 );
    deci->PreserveTopologyOn();
    deci->SetMaximumError( 0.0002 );
```

# Smoother & Graphics Objects

```
vtkSmoothPolyDataFilter *smooth = vtkSmoothPolyDataFilter::New();
smooth->SetInputConnection(dec1->GetOutputPort());
smooth->SetNumberOfIterations( 20 );
smooth->SetRelaxationFactor( 0.05 );

vtkPolyDataNormals *normals = vtkPolyDataNormals::New();
normals->SetInputConnection( smooth->GetOutputPort() );

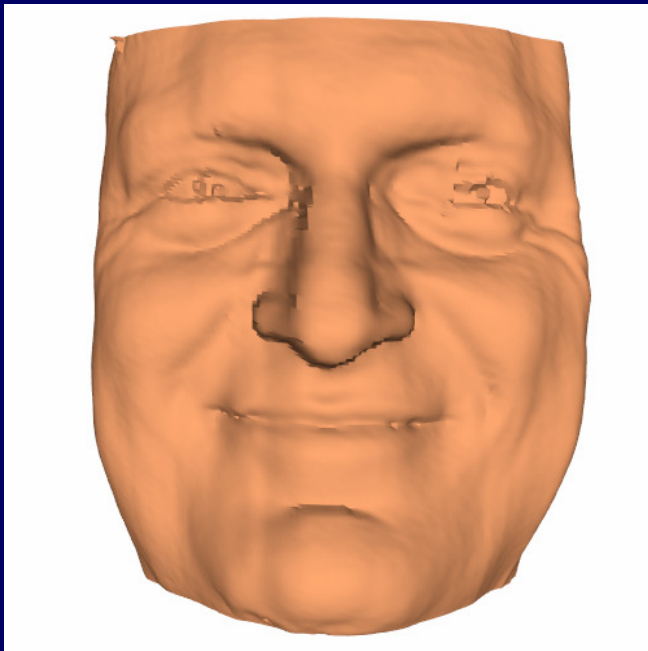
vtkPolyDataMapper *cyberMapper = vtkPolyDataMapper::New();
cyberMapper->SetInputConnection( normals->GetOutputPort() );

vtkActor *cyberActor = vtkActor::New();
cyberActor->SetMapper (cyberMapper);
cyberActor->GetProperty()->SetColor ( 1.0, 0.49, 0.25 );
cyberActor->GetProperty()->SetRepresentationToWireframe();
```

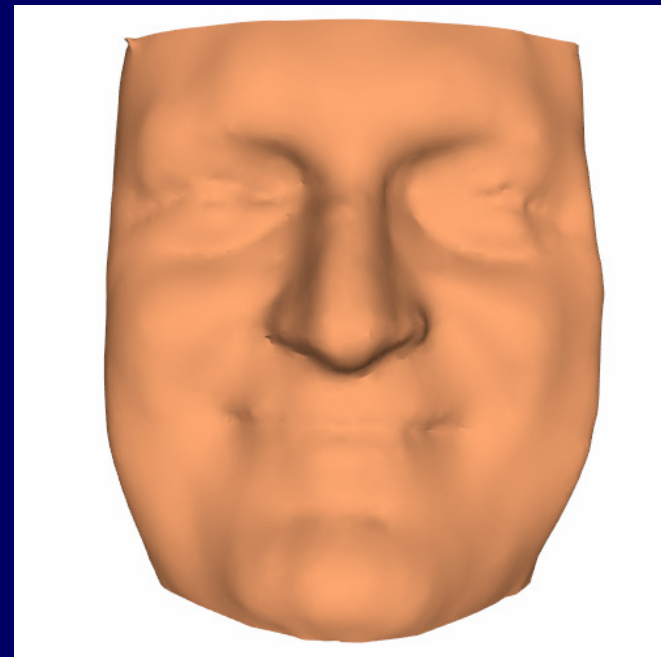
# More Graphics Objects

```
vtkRenderer *ren1 = vtkRenderer::New();  
  
vtkRenderWindow *renWin = vtkRenderWindow::New();  
renWin->AddRenderer( ren1 );  
  
vtkRenderWindowInteractor *iren =  
    vtkRenderWindowInteractor ::New();  
iren->SetRenderWindow( renWin );  
  
ren1->AddViewProp( cyberActor );  
ren1->SetBackground( 1, 1, 1 );  
renWin->SetSize( 500, 500 );  
  
iren->Start();
```

# Results



**Before**  
(52,260 triangles)



**After Decimation  
and Smoothing**  
(7,477 triangles)

## Exercise 2b

- Compile and run `exercise2b.cxx`
- Understand what the example does...
- How many source objects are there?
- How many filter objects?
- How many mapper objects?
- How many data objects are there?

## Exercise 2b (solution)

- # of Sources: 1
- # of Filters: 5
- # of Mappers: 2
  
- # of Data Objects: 6

# Exercise 2b - Notes

- `vtkImplicitFunction`
  - Any function of the form  $F(x,y,z) = 0$   
sphere of radius  $R$ :  $F(x,y,z) = R^2 - (x^2 + y^2 + z^2) = 0$
  - Examples: spheres, quadrics, cones, cylinders, planes, ellipsoids, etc.
  - Can be combined in boolean trees
    - Union, difference, intersection
  - Powerful tools for
    - Modeling
    - Clipping
    - Cutting
    - Extracting data

# Volume Rendering

---

Volume rendering is the process of generating a 2D image from 3D data.

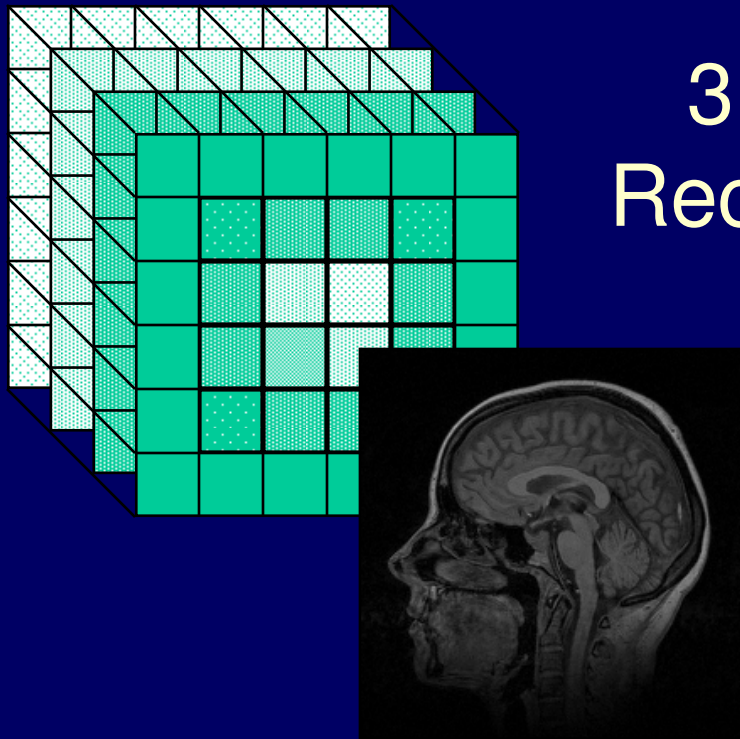
The line between volume rendering and geometric rendering is not always clear. Volume rendering may produce an image of an isosurface, or may employ geometric hardware for rendering.

# Volume Data Structures

---

- ImageData: 3D regular rectilinear grid
- UnstructuredGrid: explicit list of 3D cells

# 3D Image Data Structure



## 3D Regular Rectilinear Grid

`vtkImageData:`

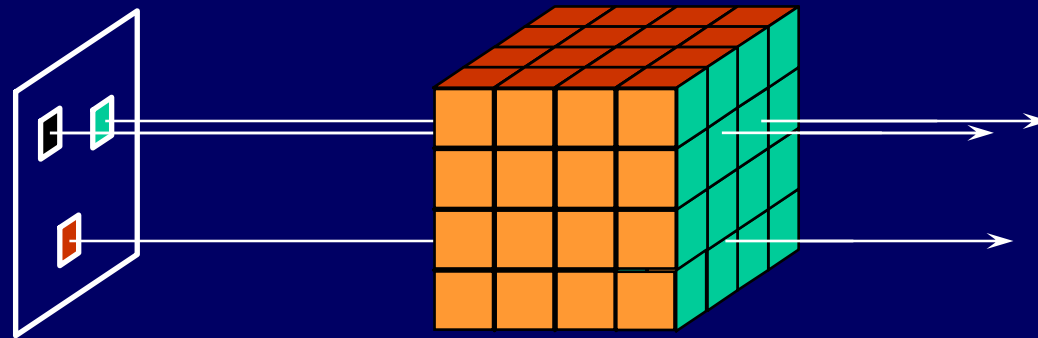
Dimensions =  $(D_x, D_y, D_z)$

Spacing =  $(S_x, S_y, S_z)$

Origin =  $(O_x, O_y, O_z)$

# Volume Rendering Strategies

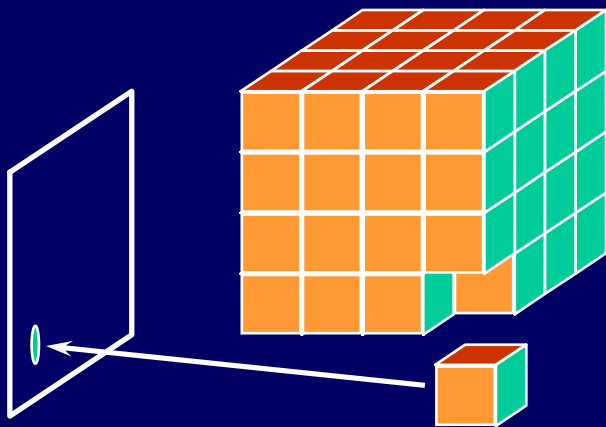
Image-Order Approach: Traverse the image pixel-by-pixel and sample the volume via ray-casting.



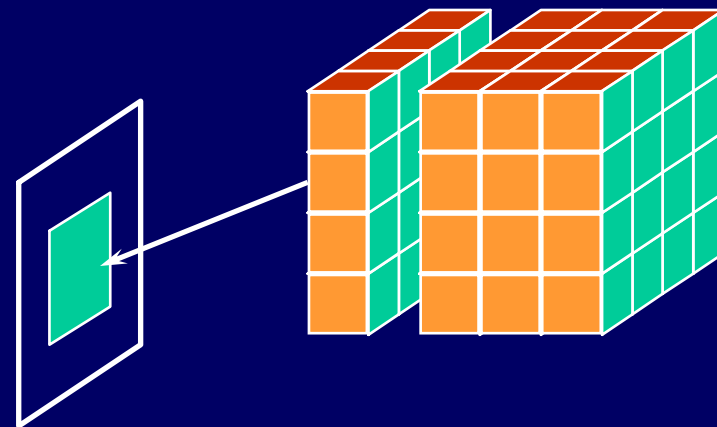
Ray Casting

# Volume Rendering Strategies

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

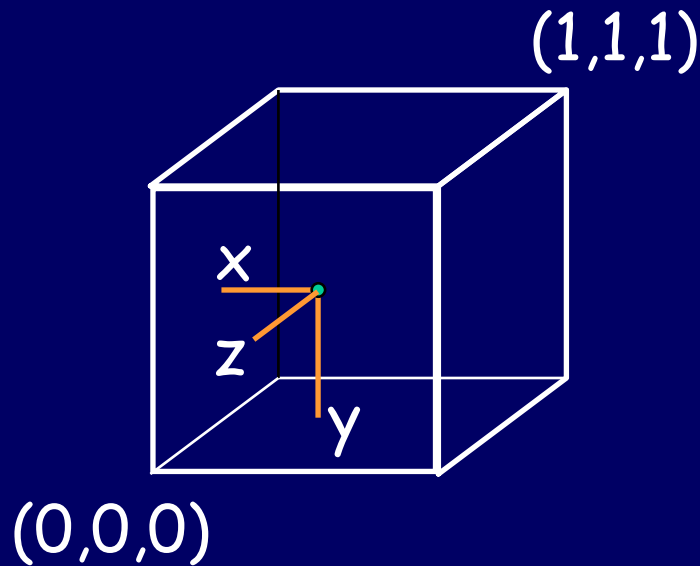


Splatting  
cell-by-cell



Texture Mapping  
plane-by-plane

# Scalar Value Interpolation



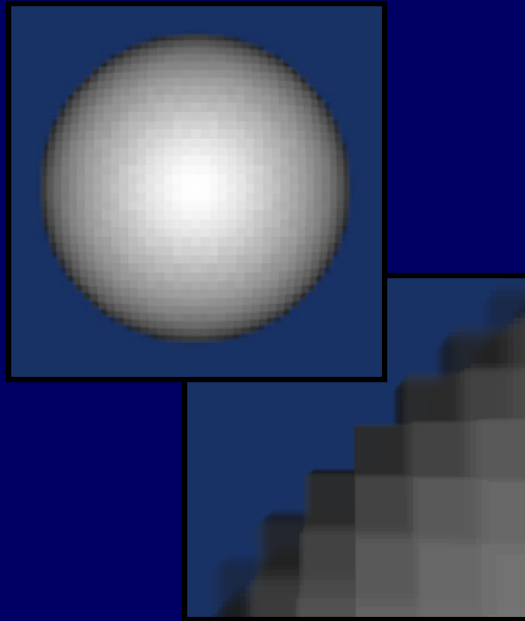
$$v = S(\text{rnd}(x), \text{rnd}(y), \text{rnd}(z))$$

$$\begin{aligned} v = & (1-x)(1-y)(1-z)S(0,0,0) + \\ & (x)(1-y)(1-z)S(1,0,0) + \\ & (1-x)(y)(1-z)S(0,1,0) + \\ & (x)(y)(1-z)S(1,1,0) + \\ & (1-x)(1-y)(z)S(0,0,1) + \\ & (x)(1-y)(z)S(1,0,1) + \\ & (1-x)(y)(z)S(0,1,1) + \\ & (x)(y)(z)S(1,1,1) \end{aligned}$$

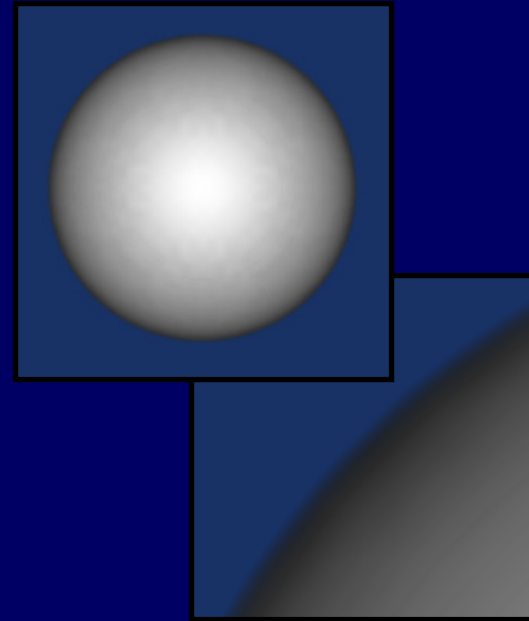
Nearest Neighbor

Trilinear

# Scalar Value Interpolation



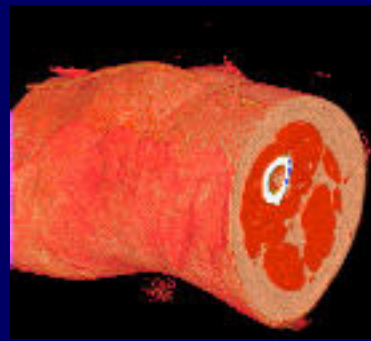
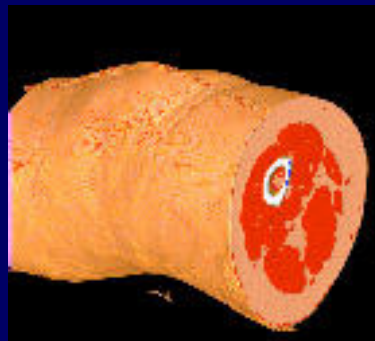
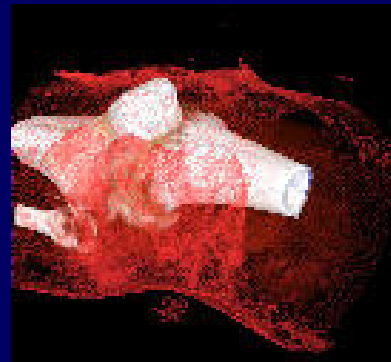
Nearest Neighbor  
Interpolation



Trilinear  
Interpolation

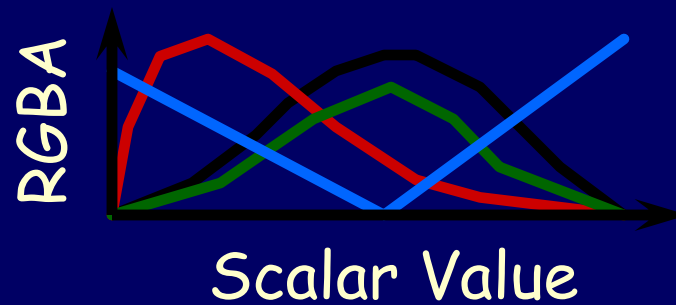
# Material Classification

Transfer functions are the key to volume renderings

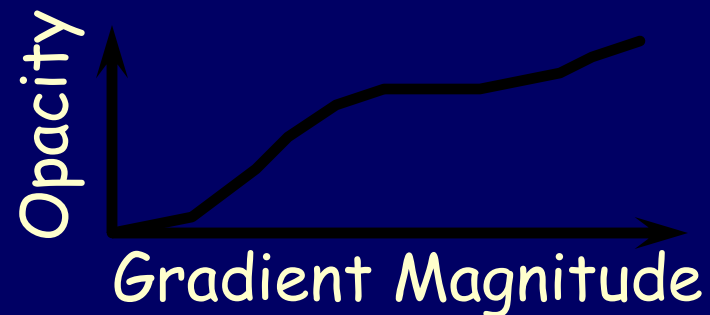


# Material Classification

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

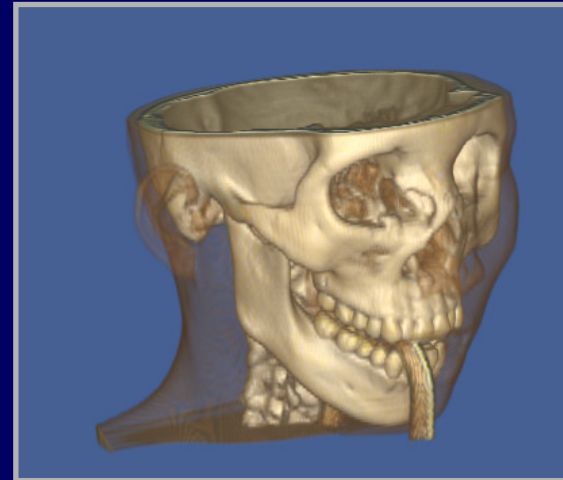
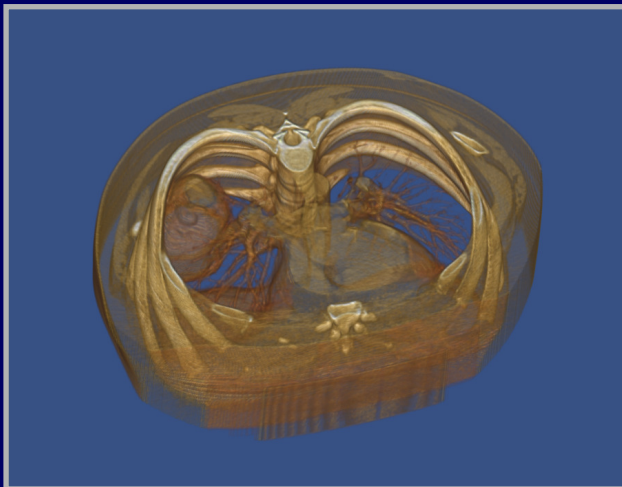
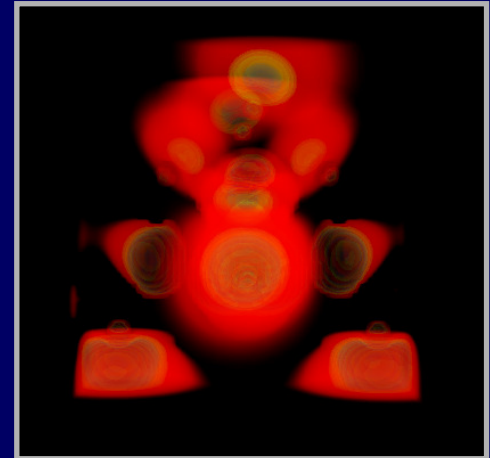
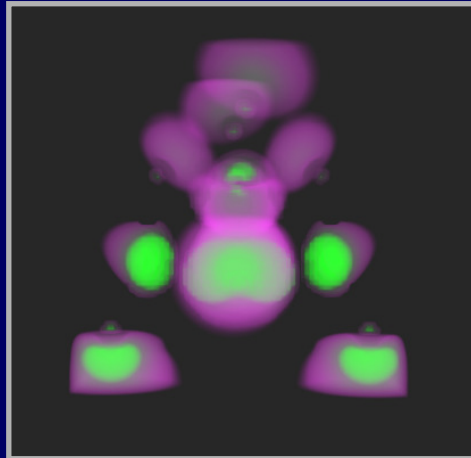
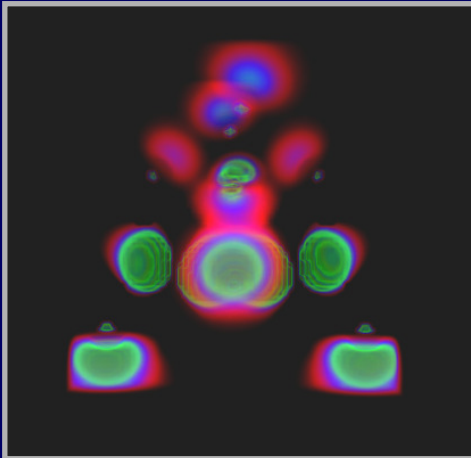


Gradient magnitude can be classified into opacity

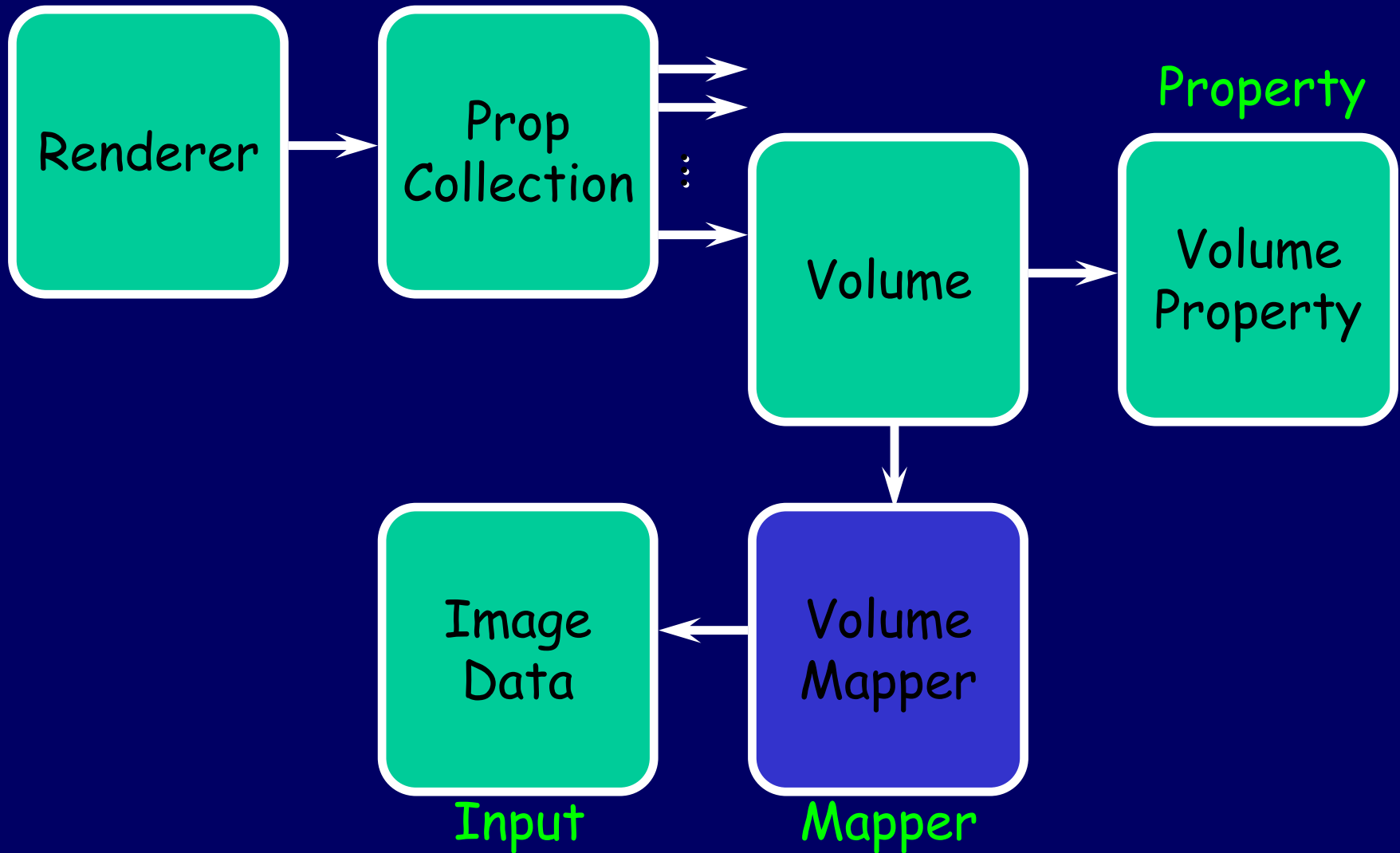


Final opacity is obtained by multiplying scalar value opacity by gradient magnitude opacity

# Material Classification



# Implementation



# Implementation

- vtkVolume
- vtkVolumeProperty
- vtkVolumeMapper
  - vtkVolumeRayCastMapper
  - vtkFixedPointVolumeRayCastMapper
  - vtkVolumeTextureMapper2D
  - vtkVolumeTextureMapper3D
  - vtkVolumeProMapper (removed in 5.0, back in CVS)

# Implementation (cont'd)

- `vtkUnstructuredGridVolumeMapper`
  - `vtkUnstructuredGridVolumeZsweepMapper`
  - `vtkUnstructuredGridVolumeRayCastMapper`
  - `vtkProjectedTetrahedraMapper`

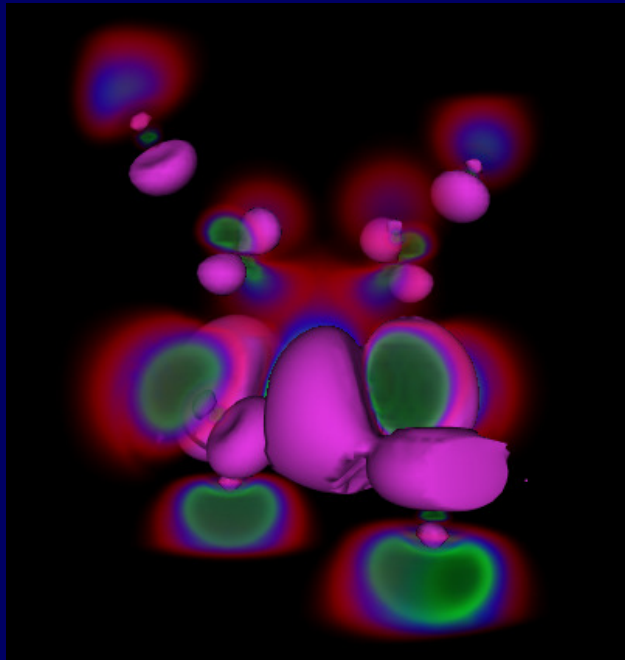
# Volume Rendering Issues

- **Quality** – is it accurate?
- **Speed** – is it fast?
- **Intermixing** – what can't I do?
- **Features / Flexibility** – what features does it have, and can I extend it?

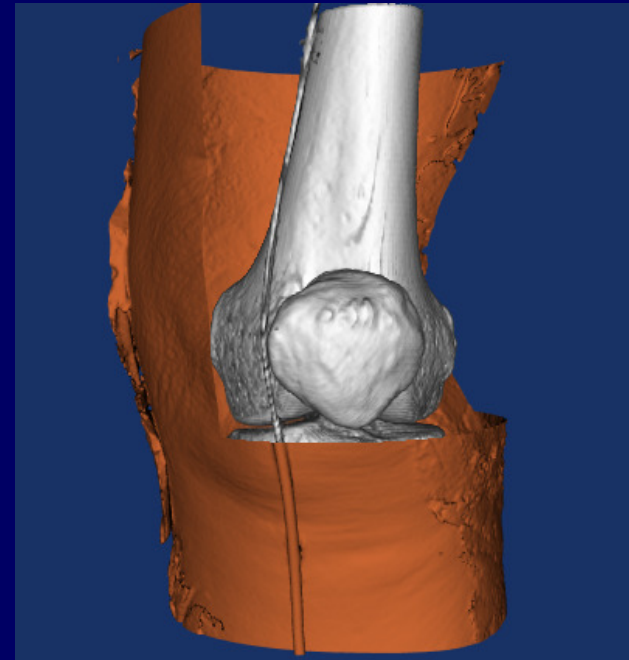
# Standard Features

- **Transfer Functions** – define color and opacity per scalar value. Modulate opacity based on magnitude of the scalar gradient for edge detection.
- **Shading** – specular / diffuse shading from multiple light sources.
- **Cropping** – six axis-aligned cropping planes form 27 regions. Independent control of regions (limited with VolumePro)
- **Cut Planes** – arbitrary cut planes. Limited by hardware (6 with OpenGL, 2 parallel with VolumePro)

# Intermixed Geometry

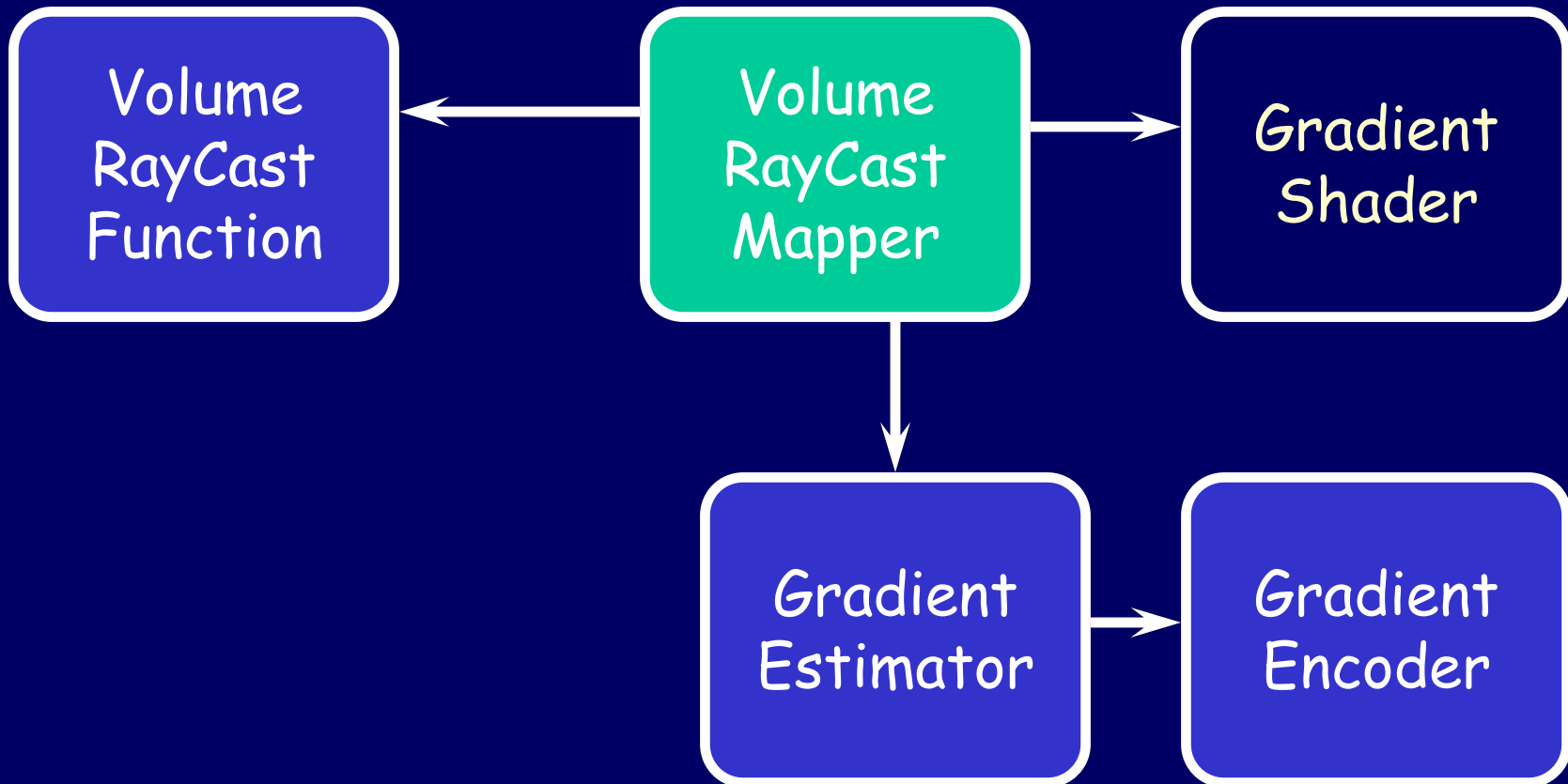


High potential  
iron protein



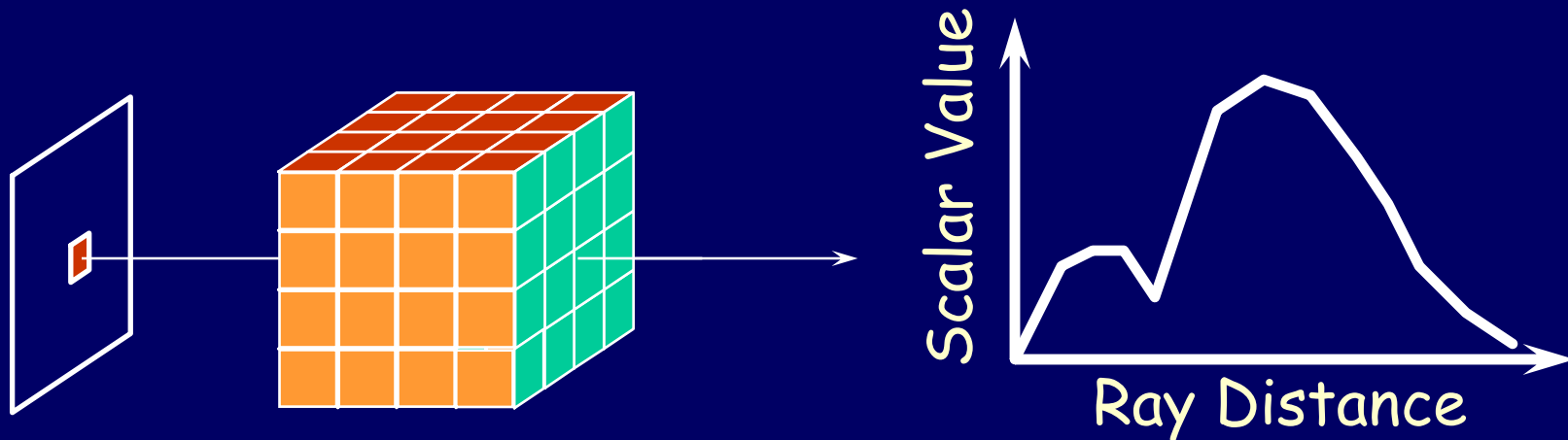
CT scan of the  
visible woman's  
knee

# Volume Ray Casting

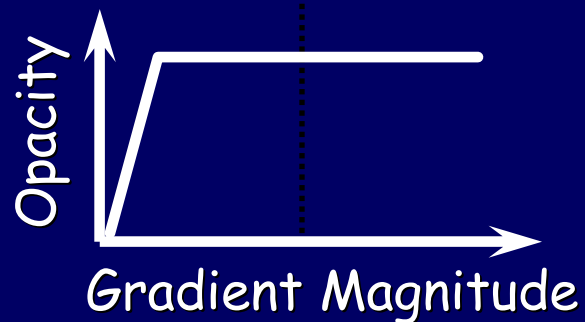
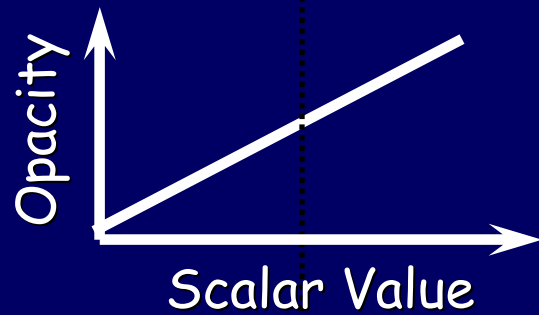
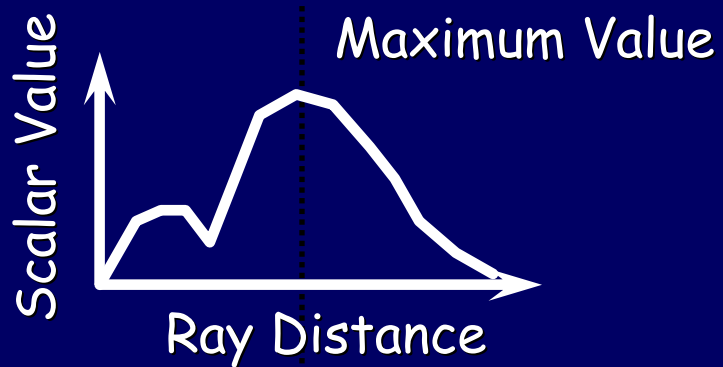


# Ray Cast Functions

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.

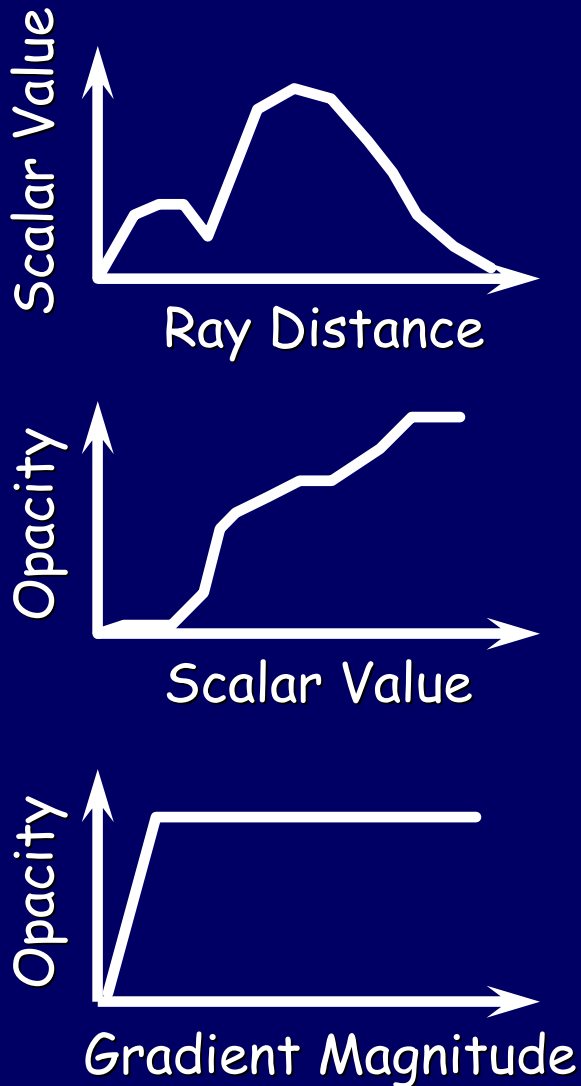


# Maximum Intensity Function



Maximize Scalar Value

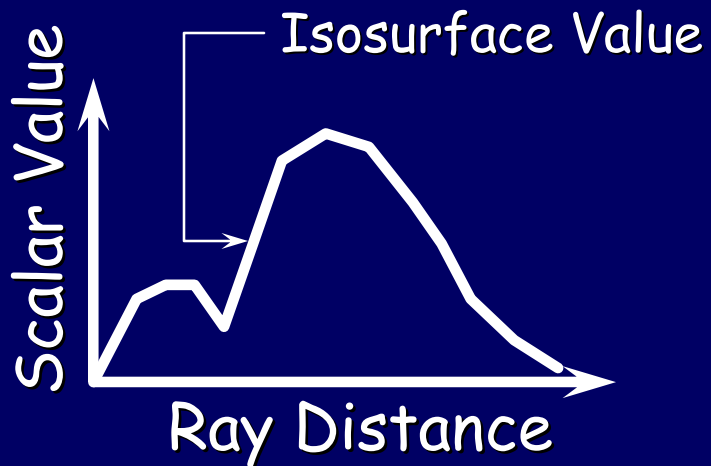
# Composite Function



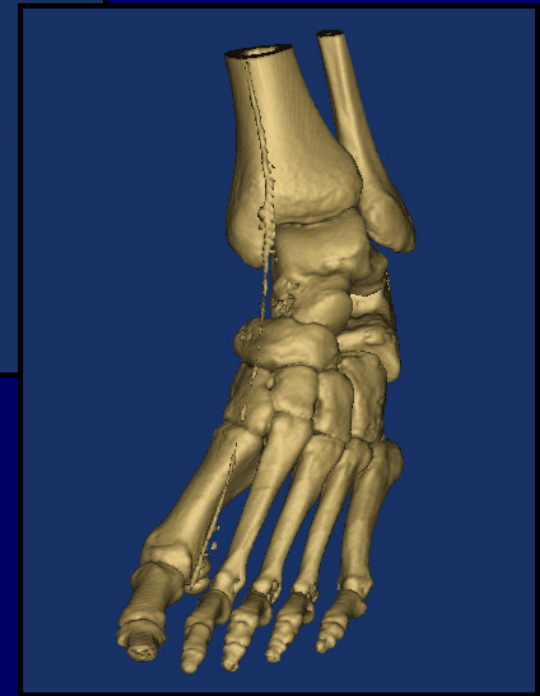
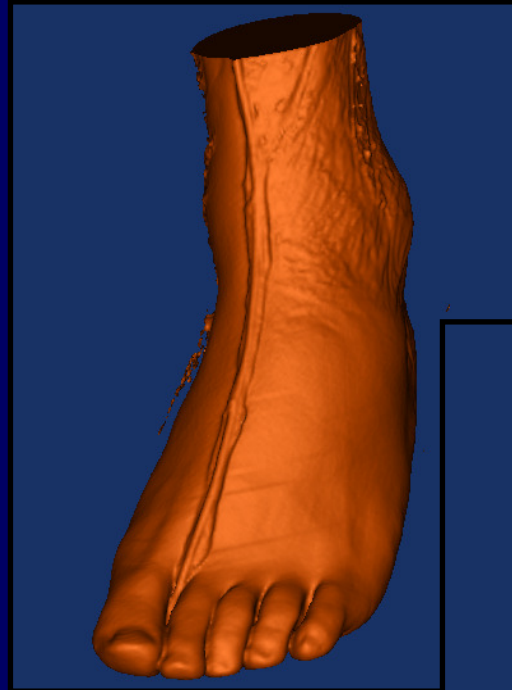
Use  $\alpha$ -blending along the ray to produce final RGBA value for each pixel.

$$I_i = c_i a_i + I_{i+1} (1-a_i)$$

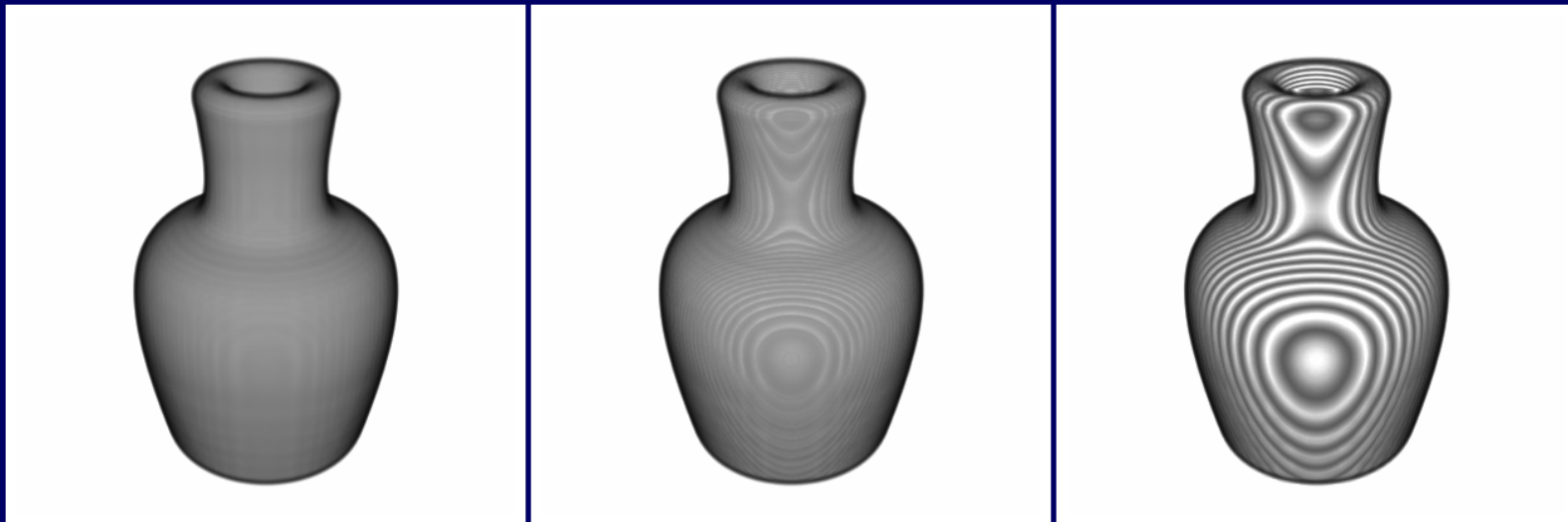
# Isosurface Function



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



# Sampling Distance



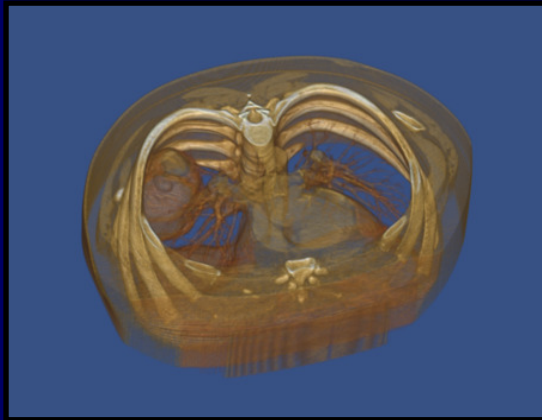
0.1 Unit  
Step Size

1.0 Unit  
Step Size

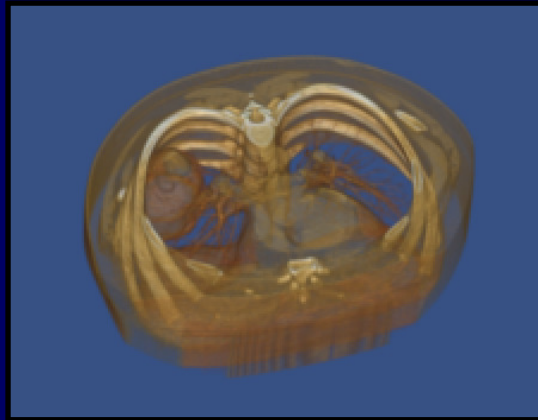
2.0 Unit  
Step Size

# Speed / Accuracy Trade-Off

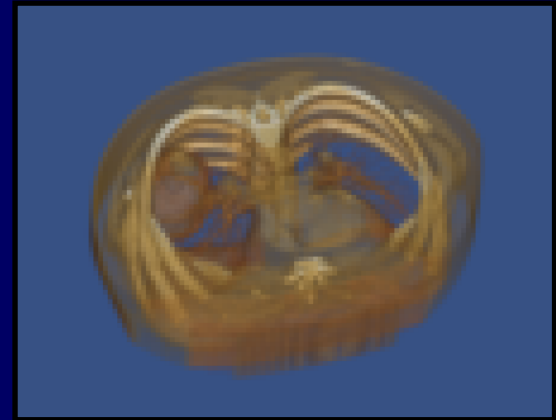
Multi-resolution ray casting:



1x1 Sampling



2x2 Sampling



4x4 Sampling

Combined approach: `vtkLODProp3D` can be used to hold mappers of various types. A volume can be represented at multiple levels-of-detail using a geometric isosurface, texture mapping, and ray casting. A decision between levels-of-detail is made based on allocated render time.