Kitware SOURCE

SOFTWARE DEVELOPER'S QUARTERLY

Issue 2 • Oct 2006

TABLE OF CONTENTS

EDITOR'S NOTE

This issue of the Kitware Software Developer's Quarterly newsletter contains three in-depth articles related to Ktiware's open source projects. Julien Jomier introduces the Spatial Object Viewers (SOV) toolkit, tying both VTK and ITK together to provide visualization methods for ITK's Spatial Objects. Utkarsh Ayachit delivers a tutorial on the use of shaders in VTK, allowing users access to advanced GPU capabilities from within the VTK rendering paradigm. Berk Geveci contributes an overview of multi-block and adaptive mesh refinement (AMR) data sets in VTK, including details on writing algorithms to process composite data sets.

Kitware would like to encourage contributions to this newletter from our active developer community by offering a free five-volume set of Kitware books for any accepted article. Perhaps you have contributed to one of the open source projects and would like to write a technical article describing your enhancement. Or perhaps you are developing a product that is built upon one or more of Kitware's open source projects, and would like to document your success or lessons learned. Please send your ideas to kitware@kitware.com.

This newsletter is just one of a suite of products and services that Kitware offers to assist developers in getting the most out of our open source products. Each project web site contains links to free resources including mailing lists, documentation, FAQs and Wikis. In addition, Kitware offers technical books and user's guides, consulting services, support contracts, and training courses. For more information on Kitware's products and services, please visit our web site at www.kitware.com.

RECENT RELEASES

VTK RELEASE

VTK 5.0.2 was released September 11, 2006. This is the second patch release to VTK 5.0. To download this release, visit http://www.vtk.org/get-software.php. Some of the changes in this release include:

- Use correct libs and include dirs for X on Mac OSX if VTK_USE_X is ON.
- Bug fix in vtkAmoebaMinimizer.
- Memory leak fix in MPEG writer.
- API fix in vtkPNGReader.
- Bug fix in GetArrayContainingComponent method of vtkFieldData.
- Bug fix in AllocateArrays method of vtkFieldData.
- Make sure CTest 2.4 loads the testing customization.
- Update API of DICOM reader for access to Image Orientation (Patient).
- Bug fix in Floor function: use union to avoid violation of aliasing rules.
- Fix for missing include of qvtkwidget.h.

PARAVIEW 3 ALPHA RELEASE

In October 2006, a new snapshot (2.9.4) of the alpha release of ParaView 3 was created. This is the fifth monthly snapshot; the first was created in June 2006. It includes binaries for Windows, Linux (32 and 64 bit) and Mac OS X. To download the snapshot, visit http://paraview.org/Wiki/ParaView_III_snapshots. The following new features are included.

- Anonymous CVS access to the source code.
- Addition of Display and Information tabs similar to ParaView 2. We are still trying to decide what to do with these pages so the way they work may change in the future. It is still possible to access the statistics page from View->Statistics View and the display page from the contextual menu of individual pipeline elements (right click -> Display Settings). We would love feedback on this.
- Layout and geometry is not stored at exit and restored at startup.
- Improved property pages for readers and filters.
- Preliminary support for plots. The first plot view we added is a bar plot view. This is used only for histograms. To try it out, follow these steps:
  - Create a sphere from the sources menu
  - Apply the histogram filter (Filters->Test->ExtractHistogram)
  - Create a histogram view (Edit->Add Plot->Histogram)
  - Select the new view by clicking on it
  - Turn the visibility of the histogram filter on by clicking on the eye icon

CMAKE RELEASE

CMake 2.4.3 was released in July 2006. It is available for download at http://www.cmake.org/HTML/Download.html. Highlights of this release include:

- Improved color output support.
- Improved Xcode support.
- Improved Fortran support.
- Improved hp itanium support.
- Added support for CXX only projects.
- Better FindWxWidgets.
- Added FindBoost.cmake.
- CPack supports multiple packages at the same time.
- Fixed to FindKDE4 to look for kde4-config first.
- Support CMAKE_CONFIG_TYPE in ctest.
- Added creation of XXX_FIND_COMPONENTS list of all components requested with REQUIRED option.
- Progress is now reported with makefiles.
- Location of CMakeTmp changed to a variable.
- Improved FindQt4 on Mac.
- Better search paths for finding VTK.
- Corrected relative path problems in ADD_SUBDIRECTORY.
- Fixed long link commands on UNIX shells.
- CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS allows for if/endif without variable.
- Added target/fast rules in the sub directories.
- Fixed in Visual Studio C and C++ targets to not add /TP and /TC.
- Bug fix in CMakeSetup when status line is long.

ITK RELEASE

ITK 2.8 was released on June 1, 2006. To download this release, go to http://www.itk.org/HTML/Download.php. Highlights of the release include:

- Adding capabilities for performing explicit instantiation.
- Adding capabilities for performing concept checking.

Both of these features are very important in the context of generic programming when C++ templates are used as an implementation mechanism. The motivation for explicit instantiation is that, by default, when a library based on templates is used for building an application, the code of the library is compiled directly into the object files of the application. If the application contains many object files, then large pieces of the templated library are compiled multiple times, imposing a burden in the compilation time, and are also stored in duplicate in the object files, resulting in overuse of disk space. This default behavior is known as implicit instantiation, to indicate that the code is compiled as it is used inside the code of the application. For example, the code of the itk::Image class will be instantiated at every declaration of: typedef itk::Image<char,3> ImageType;.

When using explicit instantiation, the developer can deliberately trigger the instantiation of the templated code, regardless of whether it is used in an application or not. The advantage is that the templates are compiled only once, and the memory required by the collective object files is smaller. The disadvantage is that the developer may end up instantiating many templates that are not actually used inside the application. Therefore, it is desirable to find a balance on the number of classes that are instantiated explicitly. In ITK this balance was found by counting the number of instantiations of every class when the 1108 tests and 400 examples of the toolkit are compiled. From those statistics, a group of the "most frequently" compiled classes was identified, along with their common template parameters. This selected group of classes was then instantiated explicitly. Brad King and Julien Jomier at Kitware found a clever way of getting the explicit instantiation to work on multiple platforms, which is not a minor feat.

Concept checking is also very important in generic programming given that C++ templates can be instantiated, in principle, over any type of argument. However, in practice, the types used in expressions may be valid only for some specific types. For example, a variable in a templated class may be used in an expression that requires addition; therefore, the type of that variable must provide an add operator. Unfortunately, in C++, when a type that does not support such an operator is used as argument for that templated class, the error message provided by the compiler is not very informative. It is common to find error messages that span five to ten lines of text. Concept checking is a mechanism for producing more informative error messages. This is done by creating small modular classes called "checkers" that exercise basic capabilities of types, and then explicitly subject the template parameters of a class to one or more of these "checkers." The class name of each checker is a very explicit error message indicating what functionality a type does not support if the checker fails to compile. In this way, when a type fails to provide a capability that the templated class needs, then a "preemptive" compilation error message is produced that starts with the class name of the "checker," giving the developer a better indication of the real source of the problem. Amy Squillacote and Brad King at Kitware, with the support of other ITK developers worked on introducing this useful functionality in the toolkit and making it work on most commonly used platforms.

Another highlight of this release was the addition of the itkSignedMaurerDistanceMapImageFilter class, which illustrated a perfect success story of the use of the Insight Journal. In a time frame of 4 hours, this class was submitted to the Insight Journal, used by a different research group, and received practical reviews by that second group. This class has now been integrated with the Insight Toolkit. Its major advantage is to provide one order of magnitude speed up with respect to the Danielsson distance map filter.

A new group of statistical distribution classes was also contributed by Jim Miller from GE Global Research.

TECHNICAL BOOKS UPDATE

The *VTK User's Guide* (ISBN 1-03-034-18-1) has been recently updated for VTK 5. This new version is 382 pages long and is printed in full color. New in this version is discussion of the re-architected execution pipeline, unstructured grid volume rendering, and support for hierarchical datasets. It is available both from Kitware's on-line store (http://www.kitware.com/products/vtkguide.html) and from Amazon.com. *The Visualization Toolkit* has also been revised for VTK 5 and is currently being printed; it is expected to be available by the end of 2006.

Will you use Kitware books to teach a course next semester? Kitware is offering significantly reduced prices for university bookstore purchases for the spring semester. Please contact sales@kitware.com for more information.

SPATIAL OBJECT VIEWERS: A VISUALIZATION TOOLKIT FOR ITK'S SPATIAL OBJECTS

Spatial Object Viewers (SOV) is a free, cross-platform, open-license toolkit for the visualization of ITK's Spatial Objects. The SOV library has been under active development for four years. It was initiated by the Computer-Aided Diagnosis and Display Lab (CADDLab) at the University of North Carolina and is now supported by Kitware.

The SOV toolkit is an object oriented C++ set of classes which integrates the Insight Toolkit (ITK) and the Visualization Toolkit (VTK) to provide rendering and user interaction to ITK Spatial Objects. The base SOV classes derive from ITK and therefore respect the same coding framework and guidelines as ITK.

Spatial Objects and Spatial Object Viewers have been used by several research groups and, as an open-source development, the toolkit has been benefiting from the feedback of its users and developers worldwide.

SPATIAL OBJECTS

ITK's Spatial Objects class hierarchy provides a consistent API for querying, manipulating, and interconnecting objects in physical space. By abstracting the representation of objects to a common data structure, Spatial Objects support a broad range of medical image analysis research such as model-to-image and model-to-model registration, atlas formation, data storage, and IO.

Spatial Objects use the *scene graph* concept from computer graphics and, therefore, they can be combined hierarchically to form a tree. Each Spatial Object has two main rigid transformations associated with it: the Object-to-Parent Transform which defines the relationship between a child and its parent and the Object-to-World transform, which is automatically computed by composing the Object-to-Parent transforms down the tree. The SceneSpatialObject provides a top level container without any transforms.

For each Spatial Object, users can query if a given point in space is inside or outside the object. This can also be applied to a collection of objects. Users can also get the value and the derivatives at any given point in space. Some filters have also been developed using SpatialObjects as input, like the itkSpatialObjectToImageFilter which rasterizes any Spatial Object to an image or the itkSpatialObjectToImageStatisticsCalculator class which computes the mean and standard deviation of a set of pixels of an image inside a given object.

The Image-Guided Surgery Toolkit (IGSTK), developed by Kitware and Georgetown University, uses ITK's Spatial Objects for object representation.

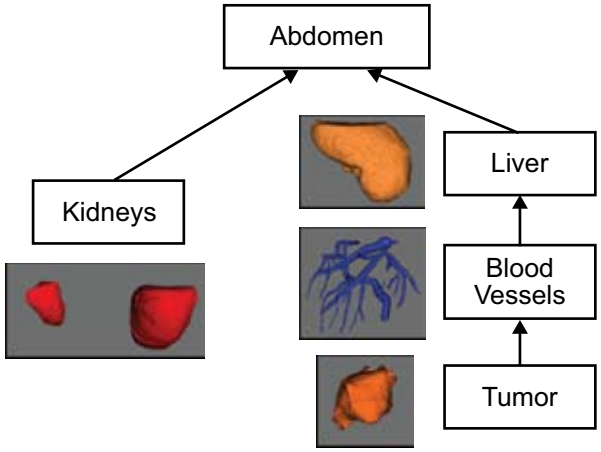


Figure 1: Scene Graph example of Spatial Objects

SPATIAL OBJECT VIEWERS

The Spatial Object Viewers toolkit allows for quick development and prototyping of applications. The toolkit is very generic and not tied to any particular graphical user interface (GUI); however, classes have been developed to support FLTK (www.fltk.org), Qt from Trolltech, and Kitware's KWWidgets (www.kwwidgets.org).

SOV provides an easy way to visualize Spatial Objects and to interact with them.

The sovRenderer is the core of the system and links the Scene, the Display and the RenderMethods Factory together. The RenderMethods Factory provides rendering capabilities at runtime for a specific object. The factory provides a simple way to plug render methods into the system. This software design allows the user to add visualization to an application quickly and with a minimal amount of code.

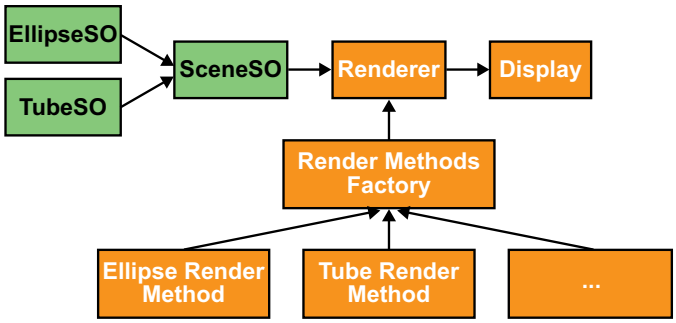


Figure 2: Spatial Object Viewers Workflow

The following code snippet shows how to implement a simple rendering application using SOV.

```
// Create a VTK Renderer
typedef sov::VTKRenderer3D RendererType;
RendererType::Pointer myRenderer3D =
   RendererType::New();

// Add the Scene to the Renderer
myRenderer3D->SetScene(m_Scene);

// Add the Renderer to the Display
myDisplay->AddRenderer(myRenderer3D);

// Update the Display
myDisplay->Update();
```

First, a SceneSpatialObject (m_Scene in the above code segment) containing spatial objects is plugged into an SOVRenderer, and that renderer is plugged into an SOVDisplay (a visualization box in the application GUI).

Second, when the display is updated, the renderer queries the RenderMethods Factory for any available render methods capable of rendering the object. This plug-in mechanism allows for flexibility and diversity of the render methods. The current 3D rendering methods are implemented using VTK and the 2D ones using OpenGL.

SOV also supports haptic devices, particularly the PHANTOM® from SensAble Technologies. In order to "feel" SpatialObjects users just have to modify the type of renderer to use. Easy enough?

```
// Declare a Phantom Renderer
typedef sov::PhantomRenderer RendererType;
RendererType::Pointer phantomRenderer =
   RendererType::New();
// Set the Scene to the Renderer
phantomRenderer->SetScene(m_Scene);
// Start the Haptic device
phantomRenderer->Start();
// Update the display
phantomRenderer->Update();
```

Spatial Object Viewers have been actively used at the University of North Carolina to represent fiber tracks from diffusion tensor images. The radio-frequency liver tumor ablation project has also greatly benefited from SOV. Among the users, Gavin Baker, from the University of Melbourne (Australia), has also chosen SOV to perform model-to-image registration of the human cochlea.

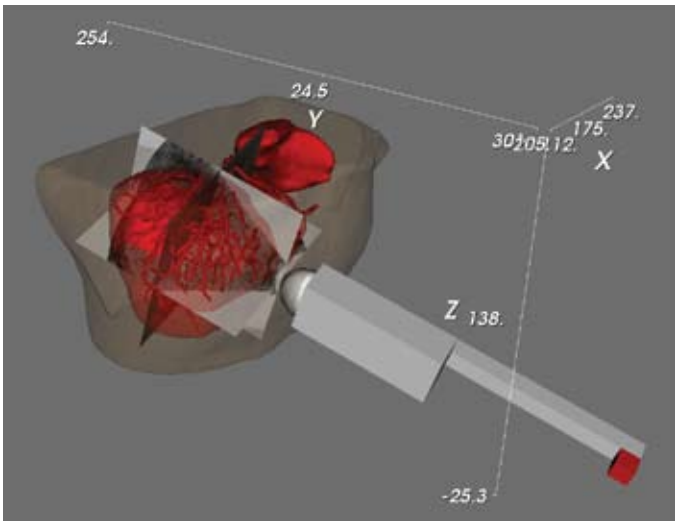


Figure 3: Image-Guided Liver Radio-Frequency Ablation under Ultrasound (IGSTK)

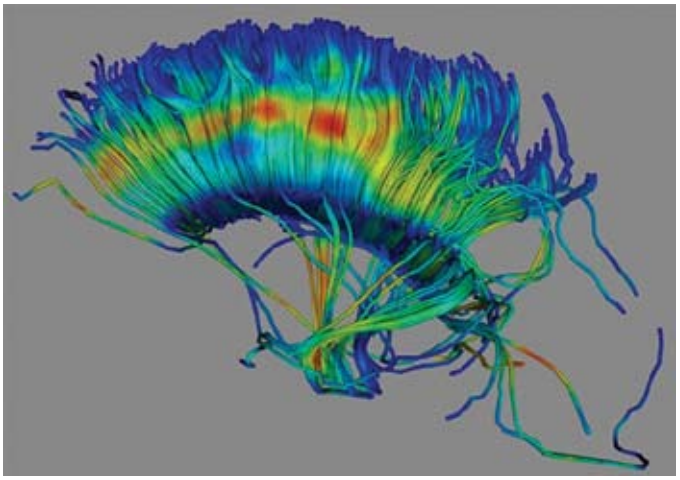


Figure 4: Diffusion Tensor Fibers represented using Tube Spatial Objects courtesy of Dr. Guido Gerig (UNC)

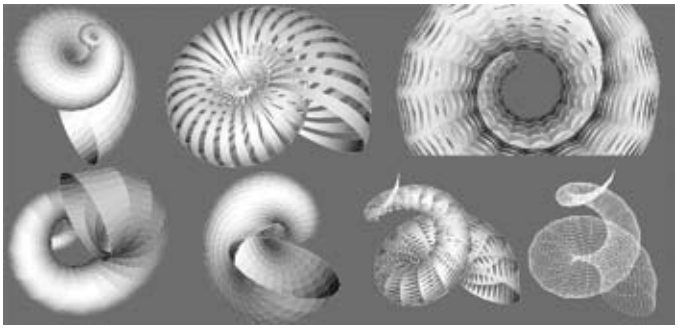


Figure 5: Model-to-Image Registration of the Human Cochlea courtesy of Gavin Baker (University of Melbourne)

SOViewers can be compiled on a large number of platforms, and the toolkit is currently tested on a nightly basis using CTest (part of CMake).

Give SpatialObjectViewers a try at http://public.kitware.com/SOViewer. More information regarding ITK's SpatialObjects can be found at www.itk.org.

ACKNOWLEDGMENTS

This work has been supported by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) at the National Institute of Health (NIH) under grant R42EB000374 and the National Library of Medicine (NLM) under contract N01-LM-0-3501.

Julien Jomier *is a technical developer in Kitware's Chapel Hill, NC office. He is one of the developers of the Insight Toolkit and the Insight Journal. He is the main architect of the Spatial Objects and Spatial Object Viewers.*

VTK SHADERS

In recent years there has been a growing interest in GPU programming. Many researchers are trying to exploit the ever increasing parallelism provided by graphics processors for generating visualizations. Several high level languages such as nVidia's Cg or GLSL, are becoming increasingly popular to harness the GPU capabilities. In collaboration with Sandia National Laboratories, we have added support for Cg as well

as GLSL GPU programs in VTK. The VTK rendering path was modified in order to allow VTK programmers to use vertex and/or fragment shaders. This article is an introduction to using this new capability. Several books and tutorials are available online for learning Cg and GLSL. This article assumes basic knowledge of either. Most of what is discussed here applies to both Cg as well as GLSL, unless otherwise noted.

ENABLING SHADERS

Shader support is available on the cvs version of VTK and is scheduled to be released with VTK 5.2. To enable shaders, we must enable the CMake variables VTK_USE_CG_SHADERS and/or VTK_USE_GLSL_SHADERS.

MATERIALS

Shaders can be applied to any actor using *"materials."* A material can be thought of as a property defining the lighting and other surface properties of an object. Materials are defined in an xml file, called a *material description file*. A material description file can be loaded into a vtkProperty using *LoadMaterial(const char* filename)*, but multiple materials cannot be loaded simultaneously.

```
vtkActor actor
[actor GetProperty] LoadMaterial "/tmp/material.xml"
```

The root element in the material description file is *<Material />*. Next we describe some supported child elements and their uses.

PROPERTY

Since a vtkProperty itself defines the surface properties of an object, the material file can be used to change the vtkProperty itself in which the material is loaded.

```
<?xml version="1.0"?>
<Material name="HelloWorld">
  <Property>
    <Member name="AmbientColor"
       value="0.5 0.5 0.5" />
  </Property>
</Material>
```

All ivars of vtkProperty (except the material itself) can be set in the material description file, following the pattern of setting the AmbientColor ivar shown in the above example. The "name" attribute of the *Member* XML element is the name of the vtkProperty ivar.

SHADER

The *Shader* element is used to include a Cg or GLSL shader in the material. More than one shader can be used in a material file; however, all of them must be in the same language. To enable the shaders defined in a material, we must also enable *Shading* on the vtkProperty.

```
[actor GetProperty] ShadingOn
```

Following are the attributes supported by the Shader element.

```
<Shader name="ShaderName"
  location="[path to shader] | Inline | Library"
  language="Cg | GLSL"
  scope="Vertex | Fragment"
  entry="[code entry point]"
  args="[arguments for compiling the shader]" >
</Shader>
```

The Cg or GLSL code can be inline, added as text under the shader element; in that case the *location=Inline.* Alternatively, a separate file can be provided; then the *location* attribute must point to the file on the disk. VTK also provides a few built in shaders as well as the capability to build shader libraries. When using a shader defined in such libraries, *location=Library* is used. The *language* attribute denotes the shader language, while *scope* indicates if the shader is a vertex program or a shader program. The entry attribute defines the *entry* routine (default is *main*).

Here's an example of a material defining a simple vertex shader written in Cg.

```
<?xml version="1.0" ?>
<!-- Simple Vertex Shader which changes the color of
the vertex. -->
<Material name="Example">
  <Shader name="Example"
      location="Inline" language="Cg"
      scope="Vertex" entry="main">
    struct Vertout {
      float4 oPosition  :POSITION;
      float4 oColor     :COLOR;
      };

    struct Vertin {
      float3 iPosition  :POSITION;
      };

    Vertout main(Vertin In)
      {
      Vertout Out;
      Out.oPosition =
         float4(In.iPosition, 1);
      Out.oColor =
         float4(0.5, 1.0, 0.5, 0.5);
      return Out;
      }
  </Shader>
</Material>
```

UNIFORM VARIABLES

Any realistic shader code uses *uniform variables*. Generally speaking, these are variables that are set by the user/application and passed on to the shader code while executing. In both Cg and GLSL, values for the uniform variables are bound when the shader is loaded. Uniform variables for a shader can be defined in the material description file as follows.

```
<Shader … >
…
  <Uniform name="VariableName"
      type="float|double|int"
      number_of_elements="[size]"
      value="[values separated by white space]">
  </Uniform>
…
</Shader>
```

Values for uniform variables can also be defined at runtime. These must be defined using the *ApplicationUniform* element and can be set using the *AddShaderVariable()* method of vtkProperty. Here's an example:

```
<Shader … >
  …
  <ApplicationUniform
    name="uAmplitude"
    value="svAmplitude">
  </ApplicationUniform>
  …
</Shader>
```

To set the variable at runtime, we do the following.

```
property LoadMaterial "Example.xml"
property AddShaderVariable {svAmplitude} 3.25
```

MATRIX UNIFORM VARIABLES

Matrix uniform variables can be used to pass matrices to the shaders. They can also be used to pass state matrices to Cg shaders. GLSL programs directly have access to the GL state hence things like the model-view-projection matix need not be passed as uniforms. However in Cg shaders, these must be explicitly passed, in which case we use the *MatrixUniform* element.

```
<MatrixUniform name="ModelViewProj"
   type="State|int|float"
   number_of_elements="[number of elements]"
   value="[pair of Cg state matrices or values]">
</MatrixUniform>
```

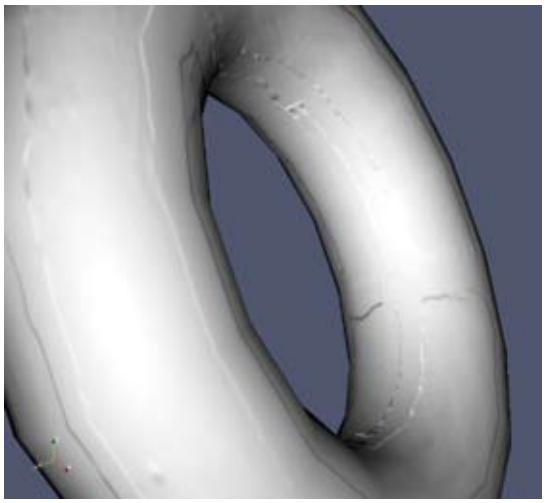


Figure 1: Bump mapping in VTK using brick texture.

TEXTURES

Texture memory is accessible in fragment shaders. It is possible to pass textures to the shaders using *SamplerUniform* elements. A texture can be loaded using the *Texture* element. It can also be loaded at runtime using *vtkProperty::SetTexture()*.

```
<Property>
  <Texture name="name1" type="2D"
     format="jpg|png|tiff|bmp|ppm"
     location="[path to the image]" />
</Property>
<Shader … >
  <SamplerUniform name="var name"
    value="name1" />
</Shader>
```

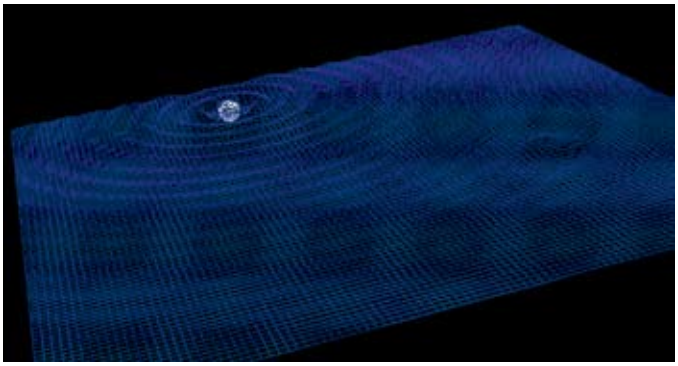


Figure 2: Animation with water surface animated using Cg shaders.

Many other special uniform elements are also available, such as *CameraUniform, LightUniform,* etc., which can be used to pass camera/lighting values to shader programs. The reader is referred to the collection of examples in *VTK/Utilities/MaterialLibrary.*

ACKNOWLEDGEMENTS

Shader support in VTK was added in collaboration with Gary Templet at Sandia National Laboratories. This work was funded by Sandia National Laboratories.

Utkarsh Ayachit is a technical developer in Kitware's Clifton Park, NY office. Mr. Ayachit is a contributor to both VTK and ParaView, and is a principal developer on Kitware's ParaView Enterprise Edition product.

COMPOSITE DATASETS IN VTK

VTK 5.0 introduced support for composite datasets. Here, a composite dataset is defined as a data object that contains one or more other datasets. These are referred to as sub-datasets. Some common examples of composite datasets include multi-block datasets and AMR (adaptive mesh refinement) datasets. Multi-block datasets have been used in scientific computation for a long time. Their most common uses are in supporting complex geometries with structured grids as well as supporting multiple parts or materials.

Figure 1: A multi-block dataset consisting of two parts: a press and a can being crushed.

Adaptive mesh refinement is a more recent development in scientific computing. It is a technique for automatically refining certain regions of the physical domain during a

numerical simulation. It is usually used in finite difference calculations with structured grids but unstructured grid applications also exist. For more information on AMR, see http://seesar.lbl.gov/AMR/Overview/amrReview.html.

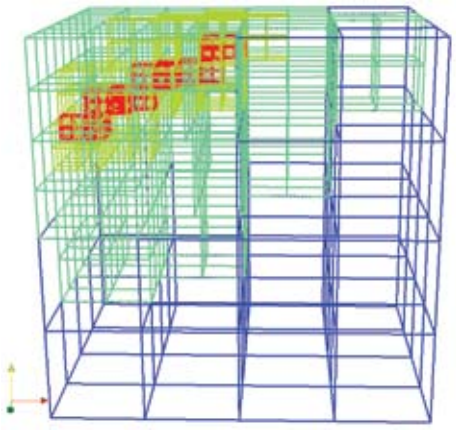


Figure 2: An AMR grid showing 3 levels of refinement.

This article describes composite data objects and related pipeline support as they exist in the current development version of VTK (5.1). Please note that the API has changed significantly since VTK 5.0.

COMPOSITE DATASETS

The class hierarchy of composite datasets is shown below.

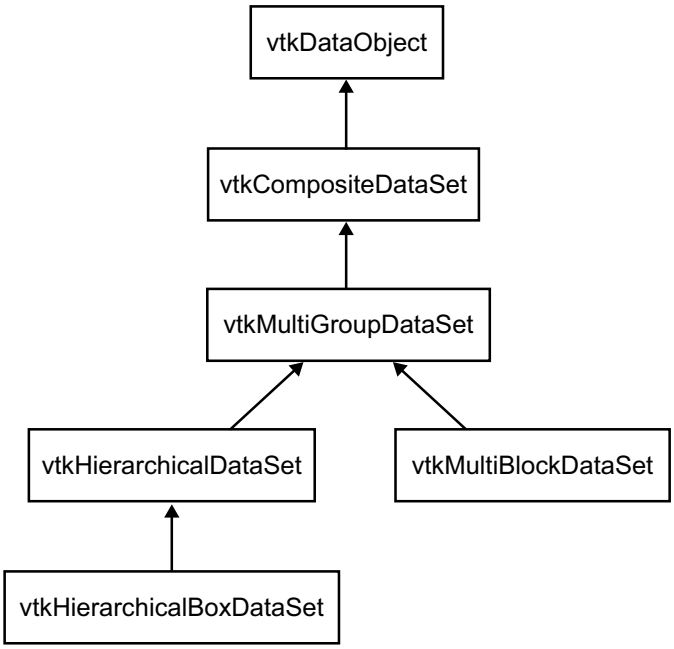


Figure 3: Composite data hierarchy.

vtkCompositeDataSet is an abstract sub-class of *vtkDataObject*. It's API does not specify how sub-datasets are stored. Therefore, the only way to access the sub-datasets is through iterators. *NewIterator()* is an abstract method that returns a *vtkCompositeDataIterator* pointer. Each sub-class of *vtkCompositeDataSet* is required to implement this method and return a concrete iterator type. Below is example code demonstrating the use of the composite data iterator.

```
vtkCompositeDataIterator *iter =
   compositeData->NewIterator();
iter->InitTraversal();
while (!iter->IsDoneWithTraversal())
   {
   vtkDataSet* ds = vtkDataSet::SafeDownCast(
                    iter->GetCurrentDataObject());
   if (ds)
      {
      cout << ds->GetNumberOfCells() << endl;
      }
   iter->GoToNextItem();
   }
```

The composite dataset API does not prevent the developer from assigning composite datasets as sub-datasets. Therefore, GetCurrentDataObject() returns a *vtkDataObject* instead of a *vtkDataSet*. However, by default, *vtkCompositeDataIterator* is in the *VisitOnlyLeaves* mode. In this mode, the iterator will only return leaf nodes of a composite data tree. If the iterator encounters a composite sub-dataset, it will start traversing it and will continue until it visits all of the sub-dataset's leaf nodes. Here, a leaf node is a sub-dataset that is not a *vtkCompositeDataSet*. In Figure 4, datasets (3), (4) and (5) are leaf nodes.

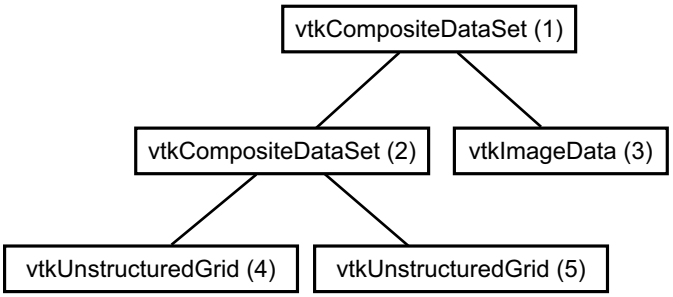


Figure 4: A composite data tree.

Note that *vtkCompositeData* is a sub-class of *vtkDataObject* not *vtkDataSet*. Therefore, it does not provide any direct access to geometry (points and cells) or attributes (point and cell data). It is the developer's responsibility to iterate over sub-datasets to obtain this information. For example, the total number of points can be computed as follows.

```
vtkCompositeDataIterator* iter =
   compositeData->NewIterator();
iter->InitTraversal();
int numPts=0;
while (!iter->IsDoneWithTraversal())
  {
  vtkDataSet* ds = vtkDataSet::SafeDownCast(
                   iter->GetCurrentDataObject());
  if (ds)
    {
    numPts += ds->GetNumberOfPoints();
    }
  iter->GoToNextItem();
  }
```

Furthermore, because *vtkCompositeDataSet* does not know how sub-datasets are stored, it does not provide any accessors that would depend on such structure. Therefore, the only way to set and get sub-datasets besides using iterators is to use the following methods.

```
virtual void AddDataSet(vtkInformation* index
   vtkDataObject* dobj);
virtual vtkDataObject* GetDataSet(
   vtkInformation* index);
```

These two abstract methods require information about which dataset is being set or gotten; this information is stored in the index argument. It is up to the subclass to interpret this information and return the appropriate value. For example, *vtkMultiGroupDataSet* searches for *vtkMultiGroupDataSet::GROUP()* and *vtkCompositeDataSet::INDEX()* keys in the information object and uses them if available.

```
void vtkMultiGroupDataSet::AddDataSet(
   vtkInformation* index,
   vtkDataObject* dobj);
{
   if (index->Has(INDEX()) && index->Has(GROUP())
      {
      this->SetDataSet(index->Get(GROUP()),
      index->Get(INDEX()),
      dobj);
      }
}
```

As can be guessed from the previous example, *vtkMultiGroupDataSet* and its sub-classes store sub-datasets in groups. The *vtkMultiGroupDataSet* API is shown below.

```
unsigned int GetNumberOfGroups();
void SetNumberOfDataSets(unsigned int group,
   unsigned int numDataSets);
unsigned int GetNumberOfDataSets(
   unsigned int group);
void SetDataSet(unsigned int group, unsigned int id,
   vtkDataObject* dataSet);
vtkDataObject* GetDataSet(unsigned int group,
   unsigned int id);
```

In essence, each *vtkMultiGroupDataSet* contains a vector of vectors of datasets.

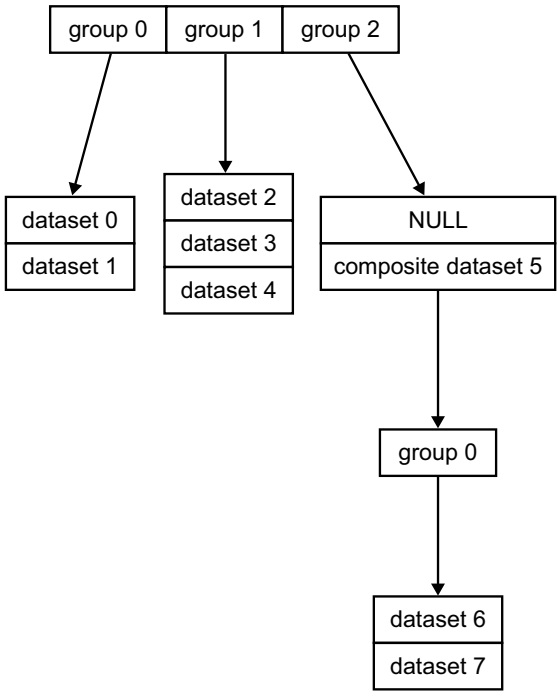


Figure 5: A multi-block data tree.

The multi-block dataset shown above consists of three groups; the first group contains two datasets and the second one contains three. The third group has one NULL dataset pointer and one composite dataset.

MULTI-BLOCK DATASETS

At the superclass level, a group is an abstract entity used to collect datasets together. The two sub-classes of *vtkMultiGroupDataSet* associate different meanings to the concept of group. *vtkMultiBlockDataSet* treats each group as blocks. The blocks might represent regions of different material or might be structured blocks created by a simulation. There is no restriction on the dataset type of blocks.

By convention, one sub-dataset is added as dataset 0 to each block (i.e. group) when running on a single processor. However, when a multi-block dataset is distributed in parallel, one dataset is added as dataset *N* (where *N* is the local process number) to each block.

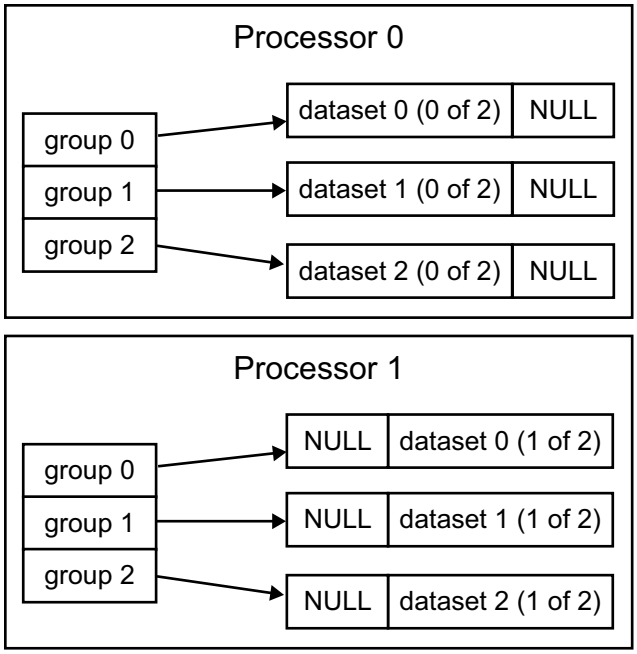


Figure 6: Multi-block dataset distributed among 2 processes.

A multi-block dataset can be populated serially as follows.

```
vtkMultiBlockDataSet* multiBlock =
   vtkMultiBlockDataSet::New();
// Add block 0
multiBlock->SetDataSet(0, 0, block0);
// Add block 1
multiBlock->SetDataSet(1, 0, block1);
```

Populating a multi-block dataset in parallel is shown below.

```
int localProcessId =
   controller->GetLocalProcessId();
vtkMultiBlockDataSet* multiBlock =
   vtkMultiBlockDataSet::New();
// Add block 0
multiBlock->SetDataSet(0, localProcessId, block0);
// Add block 1
multiBlock->SetDataSet(1, localProcessId, block1);
```

Note that this is merely a convention and is not required for the pipeline to function properly. The main reason behind assigning the dataset to *localProcessId* is to facilitate creat-

ing meta-data about the whole distributed dataset on all processes. Meta-data is associated with multi-group datasets using the *MultiGroupDataInformation* instance variable. *MultiGroupDataInformation* retains the same hierarchy as the multi-group dataset. However, instead of storing datasets, it stores vtkInformation objects. Using these objects, arbitrary information can be associated with each sub-dataset as follows.

```
vtkMultiBlockDataSet* multiBlock =
   vtkMultiBlockDataSet::New();
// Add block 0
multiBlock->SetDataSet(0, 0, block0);
// Add meta-data
vtkInformation* metaData = multiBlock->
   GetMultiGroupDataInformation()->
   GetInformation(0, 0);
metaData->Set(SOME_INFORMATION_KEY(), aValue);
// Add block 1
multiBlock->SetDataSet(1, 0, block1);
metaData = multiBlock->
   GetMultiGroupDataInformation()->
   GetInformation(1b, 0);
metaData->Set(SOME_INFORMATION_KEY(), aValue);
```

If the convention described above is followed, it is possible to associate meta-data with sub-datasets that actually reside on other processes. This is done by attaching meta-data to a sub-dataset while leaving the data pointer NULL.

HIERARCHICAL DATASETS

Another sub-class of *vtkMultiGroupDataSet* is *vtkHierarchicalDataSet.* This dataset is used to reprent AMR data. In AMR data, datasets are collected together by their level of refinement. Therefore, *vtkHierarchicalDataSet* uses groups to represents levels of refinement.

The AMR simulation usually starts with one level. After the first calculation pass, it finds all regions in which the computational error is above the acceptable value and refines them by creating new blocks that have smaller cells. Depending on the simulation, these new blocks might overlap with the lower level blocks. If this is the case, the lower level cells that have been replaced by refined ones are marked as invalid or invisible. This refinement process creates a hierarchy of datasets in which the higher levels result from more accurate calculations.

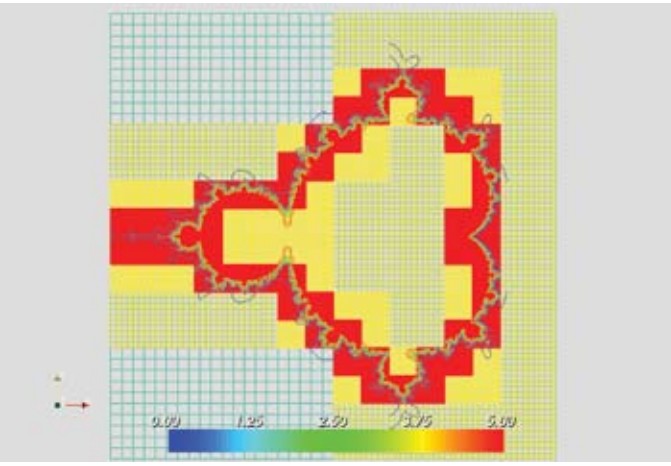


Figure 7: 2D AMR grid colored by level as well as the contour lines from the data

vtkHierarchicalDataSet does not impose any restriction on the type of dataset that it can store. Therefore, it can be used to represent both unstructured and structured AMR datasets. However, the most common AMR data type is uniform, rectilinear grid. *vtkHierarchicalBoxDataSet* is used to model such datasets. This class restricts its sub-dataset type to be *vtkUniformGrid*. In most cases, *vtkUniformGrid* is equivalent to *vtkImageData*. (It is actually a sub-class of *vtkImageData*.) The main difference between the two classes is that *vtkUniformGrid* supports cell and point blanking. A blanked cell or point is one that is replaced by a higher level cell or point and is not processed by the pipeline. Cell blanking information can be automatically generated by *vtkHierarchicalBoxDataSet* as long as it knows the extent of each block. This information is stored as meta-data described above. Since this information is always required, *vtkHierarchicalBoxDataSet* defines a new signature for setting a dataset (shown below).

```
void SetDataSet(unsigned int level, unsigned int id,
   vtkAMRBox& box,
   vtkUniformGrid* dataSet);
```

vtkAMRBox is a simple class that has two instance variables *int LoCorner[3]* and *int HiCorner[3].* This is similar to the box concept used by Chombo (http://seesar.lbl.gov/ANAG/chombo/). All boxes share the same origin but use indices based on the local refinement level. The position of a point in a box can be computed as *(x,y,z) = (originx,originy,originz)+(i*dx, j*dy, k*dz) where LoCorner <= (i,j,k) < HiCorner.* Another required piece of information is the refinement ratio.

This is defined for all levels except the last as *rl = dxl+1 / dxl,* where *dx* is the spacing, *l* is the level, and *r* is the refinement ratio. (The methods below are defined for *vtkHierarchicalBoxDataSet.*)

```
void SetRefinementRatio(unsigned int level,
   int refRatio);
int GetRefinementRatio(unsigned int level);
```

Once the boxes and refinement ratios are defined, the visibility arrays are computed using the following method.

```
void GenerateVisibilityArrays();
```

PIPELINE EXECUTION

The composite data infrastructures in VTK makes it unnecessary to rewrite or change existing filters to support composite datasets. Instead, it allows using "simple" filters (filters that can only operate on non-composite datasets) whenever possible. This requires looping over leaf nodes in a composite dataset, passing each simple dataset to the simple filter, executing the filter and collecting the results in an output composite dataset. This responsibility falls to *vtkCompositeDataPipeline*. All composite data algorithms automatically use a *vtkCompositeDataPipeline* as their executive. (For more information on executives in the new pipeline, see Kitware Source, Issue 1.) It falls to the developer to make sure that:

1. Each algorithm that will process composite data in a mixed pipeline is assigned *vtkCompositeDataPipeline* as its executive.

2. The pipeline either contains a composite data consumer (i.e., a sink -- a mapper or writer -- that accepts composite data as input) or an algorithm that accepts composite data and produces simple data.

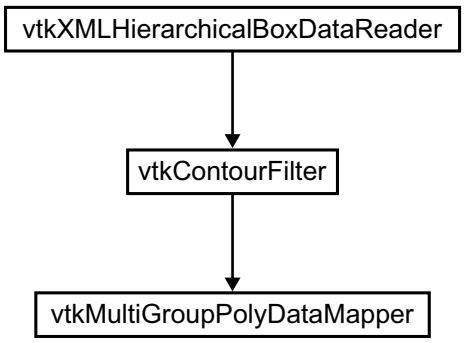


Figure 8: Mixed pipeline of composite data and simple data algorithms.

The pipeline shown in Figure 8 can be created with the following piece of code.

```
vtkCompositeDataPipeline* cdp =
   vtkCompositeDataPipeline::New();
vtkAlgorithm::SetDefaultExecutivePrototype(cdp);
cdp->Delete();

vtkXMLHierarchicalBoxDataReader* reader =
   vtkXMLHierarchicalBoxDataReader::New();

vtkContourFilter* contour = vtkContourFilter::New();
contour->SetInputConnection(
   reader->GetOutputPort());

vtkMultiGroupPolyDataMapper* mapper =
   vtkMultiGroupPolyDataMapper::New();
mapper->SetInputConnection(
contour->GetOutputPort());
```

Note the use of *SetDefaultExecutivePrototype()*. This method instructs all algorithms to instantiate a copy of the *Default ExecutivePrototype* in *CreateDefaultExecutive(),* making sure that all algorithms use a *vtkCompositeDataPipeline.* When the pipeline is updating and it is *vtkContourFilter*'s turn to execute, the composite data executive assigned to the algorithm visits each leaf in the composite data and executes the contour filter on it. The result is then collected to a *vtkMultiGroupDataSet* that mirrors the input's hierarchy. Further down the pipeline, the *vtkMultiGroupPolyDataMapper* receives the multi-group of *vtkPolyData* produced by the contour filter and renders it. There are some limitations to this behavior:

- If the simple filter has more than one input, only the first composite dataset is iterated over. All other composite inputs are passed as composite data to the algorithm.
- Regardless of the composite input type to a simple filter, its output is a *vtkMultiGroupDataSet.*
- The composite data pipeline uses piece based data distribution only. Structured extent requests are ignored.
- Support for ghost points/cells is minimal.

WRITING COMPOSITE DATA ALGORITHMS

It is not always possible to use algorithms designed for simple datasets on composite datasets. Certain algorithms require information about the whole dataset instead of one block at a time (e.g., streamlines, probing and group extraction). Furthermore, composite data readers must be able to produce all of their output at once. The easiest way to write a composite data algorithm is to subclass from one of the abstract superclasses listed below.

- *vtkMultiGroupDataSetAlgorithm*
- *vtkMultiBlockDataSetAlgorithm*
- *vtkHierarchicalDataSetAlgorithm*
- *vtkHierarchicalBoxDataSetAlgorithm*

In the concrete subclass, *RequestInformation(), RequestUpdateExtent()* and *RequestData()* are implemented as appropriate in the usual manner. The first thing to do is to make sure the number of input and output ports are set to the appropriate values in the constructor. For example:

```
vtkMultiBlockPLOT3DReader::
   vtkMultiBlockPLOT3DReader()
{
   // ...
   this->SetNumberOfInputPorts(0);
}
```

In *RequestInformation()*, the maximum number of pieces the algorithm can produce is set and the meta-data to be propagated downstream is created.

```
int vtkXMLMultiGroupDataReader::RequestInformation(
   vtkInformation *request,
   vtkInformationVector **inputVector,
   vtkInformationVector *outputVector)
{
   this->Superclass::RequestInformation(request,
   inputVector, outputVector);
   vtkInformation* info =
      outputVector->GetInformationObject(0);
   info->Set(vtkStreamingDemandDrivenPipeline::
      MAXIMUM_NUMBER_OF_PIECES(), -1);

   return 1;
}
```

Note that if the maximum number of pieces is set to -1, algorithms downstream will assume that the source can produce as many pieces as requested. In most situations, the maximum number of non-empty pieces a source can produce is limited by the number of cells or points available (in a file for example). If the requested number of pieces exceeds this value, some of the pieces may be empty datasets. The pipeline will handle empty datasets without errors. Therefore, although it is possible to set the maximum number of pieces to the number of cells, points or blocks, it is usually easier to simply set it to -1. The other extreme is when a source cannot produce more than 1 piece. In this situation, the maximum number of pieces should be set to 1.

It is possible to provide meta-data in *RequestInformation().* This is done by creating and populating a *vtkMultiGroupDataInformation* and setting it on the output information using the *vtkCompositeDataPipeline::COMPOSITE_DATA_INFORMATION()* key. This is not required and is not used by any algorithms yet.

The *RequestData()* method of composite data algorithms is usually very similar to that of simple data algorithms.

```
int vtkMultiGroupDataExtractDataSets::
   RequestDataObject(
      vtkInformation*,
      vtkInformationVector** inputVector ,
      vtkInformationVector* outputVector)
{
   vtkCompositeDataSet* input =
      vtkCompositeDataSet::GetData(inputVector[0], 0);
   vtkCompositeDataSet *output =
      vtkCompositeDataSet::GetData(outputVector, 0);
   // Do something with input and output

   return 1;
}
```

If, for some reason, it is not possible to sub-class from one of the composite data algorithm superclasses, writing an algorithm requires implementing a few more virtual functions. The first methods to override are *FillInputPortInformation()* and *FillOutputPortInformation()*.

```
int vtkMultiBlockDataSetAlgorithm::
   FillInputPortInformation(int,
      vtkInformation* info)
{
   // now add our info
   info->Set(
      vtkAlgorithm::INPUT_REQUIRED_DATA_TYPE(),
      "vtkMultiBlockDataSet");
   return 1;
}

int vtkMultiBlockDataSetAlgorithm::
   FillOutputPortInformation(int,
      vtkInformation* info)
{
   info->Set(
      vtkDataObject::DATA_TYPE_NAME(),
      "vtkMultiBlockDataSet");
   return 1;
}
```

If the algorithm can handle both composite and simple datasets, the *INPUT_REQUIRED_DATA_TYPE()* should be *vtkDataObject.* It is also important to make sure that the executive created by composite data algorithms is *vtkCompositeDataPipeline.*

```
vtkExecutive*
vtkMultiBlockDataSetAlgorithm::
   CreateDefaultExecutive()
{
   return vtkCompositeDataPipeline::New();
}
```

Depending on the superclass, it may also be necessary to implement *ProcessRequest()*. For more details on implementing this method, see Kitware Source, Issue 1.

Some existing composite data algorithms are listed below.

- Readers/Writers (in VTK/IO):
  - *vtkMultiBlockPLOT3DReader*
  - *vtkXMLMultiGroupDataReader*
  - *vtkXMLMultiBlockDataReader*
  - *vtkXMLHierarchicalDataReader*
  - *vtkXMLHierarchicalBoxDataReader*
  - *vtkXMLMultiGroupDataWriter*
  - *vtkGenericEnSightReader, and subclasses*
- Filters (in VTK/Graphics):
  - *vtkMultiGroupDataExtractDataSets*
  - *vtkMultiGroupDataExtractGroup*
  - *vtkMultiGroupDataGroupFilter*
  - *vtkMultiGroupDataGroupIdScalar*
- Mappers (in VTK/Rendering):
  - *vtkMultiGroupPolyDataMapper*

CONCLUSION

This article provides a brief summary of the multi-block and AMR data implementation in VTK 5.1. Additional information can be found in the online VTK documentation at: http://www.vtk.org/doc/nightly/html/classes.html. Click on the following links in the alphabetical class listing:

- vtkCompositeDataPipeline
- vtkMultiGroupDataSet
- vtkMultiBlockDataSet
- vtkHierarchicalDataSet
- vtkHierarchicalBoxDataSet

Please note that this is an ongoing project, and changes to the API are likely to happen as the code matures. Some of the work in progress includes volume rendering of AMR datasets and multi-time datasets.

ACKNOWLEDGEMENTS

This work was funded by the US Department of Energy ASCI Views program as part of a multi-year contract awarded to Kitware Inc. by a consortium of three National Labs: Los Alamos, Sandia and Lawrence Livermore. The author would like to thank Brian Wylie at Sandia National Laboratories, Jim Ahrens at Los Alamos National Laboratory, and John Biddiscombe and Jean Favre at the Swiss National Supercomputing Centre for their contributions to this work.

Berk Geveci *is a project lead in Kitware's Clifton Park, NY office. Dr. Geveci is the project manager for the open source ParaView application. He contributes to the technical development of VTK and ParaView.*

IN PROGRESS

VTK TIME SUPPORT

In collaboration with Sandia National Laboratories, Kitware is improving time support in VTK. Previously, VTK assumed that the contents of the pipeline represent a single timestep, which limited VTK to supporting only a single, discrete timestep per pipeline. The revised time support allows for both discrete and continuous time sources, and provides for interpolation of time-varying data through the use of specialized filters, therefore minimizing the coding impact on sources.

As part of this effort several new classes are being added to VTK including vtkTemporalDataSet, a subclass of vtkMultiGroupDataSet that is used to hold temporal data, and vtkTemporalDataSetAlgorithm, a parent class defining the API for temporal algorithms. The vtkTemporalInterpolator can be used to linearly interpolate between timesteps to convert a discrete time series into a continuous time series.

The vtkTemporalDataSetCache is a filter that caches time steps so that if a recently requested timestep is requested again it may be found in the cache, saving the effort of reloading or recomputing it.

Time support is currently under active development, with a preliminary version available in the cvs version of VTK. A finalized version of improved time supported is expected to be released with VTK 5.2.

KITWARE NEWS

HIGH-PERFORMANCE GPU COMPUTING

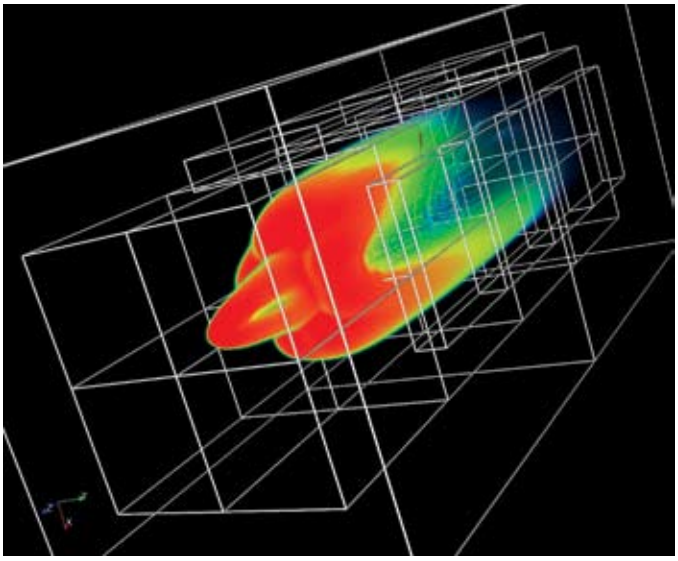
The DOD Army Research Lab has Awarded Kitware a Phase II STTR for accelerated data processing and rendering using GPU technologies. The focus of the work is to utilize the graphical processing unit to improve the data processing performance of the ParaView visualization system. Through the use of faster processing and rendering algorithms, and data saliency techniques that allow the user to focus on the important aspect of the problem, ParaView will be extended to support visualization of coupled simulations during the simulation process. The PI for this award is Dr. Amitabh Varshney http://www.cs.umd.edu/~varshney/ at the University of Maryland College Park. At Kitware, this effort will be lead by Andy Cedilnik.

AUTOMATIC SEGMENTATION

The DOD Air Force Research Lab has awarded Kitware a Phase I SBIR for automating the segmentation of 3D image datasets, for the purpose of converting these segmented datasets into simulation models. In addition, Kitware will develop semi-automatic methods including editing tools (i.e., 3D widgets) for manually guiding or modifying the results of segmentation tasks.

AMR VOLUME RENDERING

The National Science Foundation has awarded Kitware a Phase II SBIR for volume rendering of adaptive mesh refinement (AMR) data. Currently, scientists who utilize the AMR data structure in order to improve the performance and accuracy of their simulation process have very few options on how to visualize the resulting data. As part of this project, Kitware will extend the volume rendering functionality in VTK to support AMR and multi-block data. This new technology will appear in both the open source ParaView visualization tool, and Kitware's proprietary VolView application.



An example of AMR volume rendering intermixed with geometric streamlines. The bounding boxes show the hierarchical structure of the data set. Data courtesy of the Swiss National Supercomputing Centre.

EMPLOYMENT OPPORTUNITIES

Kitware is seeking talented software professionals with experience in volume rendering, GPU programming, medical image processing, 3D graphics, visualization, and/or computer vision. Applicants must have demonstrated software development skills, and must show the initiative, flexibility, and the focus necessary to deliver quality software (both open-source and proprietary code).

Qualified candidates will work with leaders in the field on cutting edge problems. Kitware offers significant growth opportunities, an annual bonus, six weeks total paid time off, health and dental benefits, and a 401(k) plan with company contributions. Kitware is an equal opportunity employer. Send your resume to kitware@kitware.com with "Kitware Employment" as the subject line. Please include a plain text cover letter in the body of the email.

Kitware's Software Developer's Quarterly is published by Kitware, Inc., Clifton Park, New York.

Contributors: Utkarsh Ayachit, David Cole, Berk Geveci, Bill Hoffman, Luis Ibáñez, Julien Jomier, Ken Martin, Amy Squillacote.

Design: Melissa Kingman, www.elevationda.com

Editor: Lisa Avila

To contribute to future editions of this publication, please contact the editor at kitware@kitware.com.