VTK Classes for DICOM Files and Meta Data

Release 0.3.2

David G. Gobbi[1,2]

October 7, 2013

[1]Department of Radiology, University of Calgary, Calgary, Canada
[2]Calgary Image Processing and Analysis Centre, Foothills Medical Centre, Calgary, Canada
david.gobbi@gmail.com

Abstract

This document describes a library of C++ classes for use with VTK that provide functionality for reading and writing DICOM files, and for accessing both the images and the meta data for the images. The primary functional requirements of the library were efficient and intuitive access to the meta data, and correct geometrical representation of the images within VTK, even for images with more than three dimensions. The classes can be built with CMake and used with VTK 5.8 or later versions of VTK.

Contents

1 Introduction

DICOM is the acronym for Digital Imaging and Communications in Medicine, which is the international standard for transferring medical images across computer networks and storing them within digital archives. Its development began in the early 1980's, when digital archiving of medical images was new, and in the decades since then it has become universally supported by medical imaging device manufacturers.

While most image file formats include a file header that provides only basic information about the image, the DICOM file header is very rich, and provides standardized and extensible meta data related not only to the image itself, but also related to the medical imaging device, the hospital, and the patient.

The purpose of this document is to describe a set of C++ classes that can be used with VTK not only to read DICOM images into VTK, but also to access and manipulate all of the attributes stored in the DICOM meta data. Two classes in particular, the vtkDICOMSorter and vtkDICOMReader, utilize the meta data to ensure that VTK loads each series of DICOM images as a structed image volume with correct geometry and orientation.

2 Meta Data

The attributes of a DICOM data set are stored in the DICOM meta data. The meta data is an ordered series of data elements, where each data element consists of a 32-bit key called a *tag* and a data array called a *value*. Another way to think of the meta data is that it is a mapping of *tags* to *values* that defines the attributes of the image. As an example, the data element that gives the pixel spacing for the image will consist of the tag (0028,0030) followed by a string of bytes that contains two ASCII decimal numbers separated by a backslash. The fields that make up a data element are shown in Fig. 1, and are described in detail in the following sections.

2.1 Tag

Each attribute in the DICOM meta data is identified by a 32-bit tag that is written in hexadecimal as a four-digit *group number* followed by a four-digit *element number*. For example, (0008,103E) is the tag for the DICOM "Series Description" data element. Each tag that uses an even group number has a strict definition within the DICOM public standard, while tags that use an odd group number are defined within the internal standards of one of the many medical device manufacturers. The latter are often called *private tags* because the manufacturers are not required to reveal their definition to the public. All non-private tags, however, are described in parts 3 and 6 of the public DICOM standard [1].

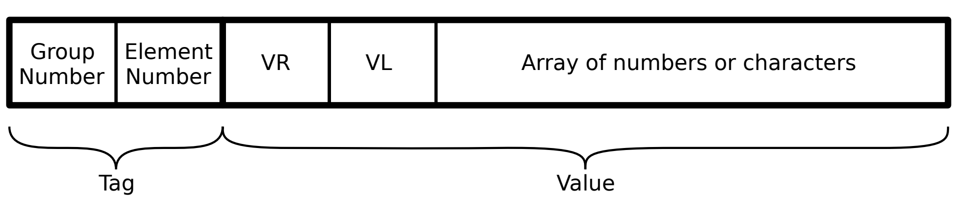

Figure 1: A DICOM DataElement.

```
// Construct a tag with default value (0x0000,0x0000).
vtkDICOMTag tag;

// Two ways to create a tag.
tag = vtkDICOMTag(0x0028, 0x0030);
tag = vtkDICOMTag(DC::PixelSpacing);

// Get the group or element from a tag.
unsigned short group = tag.GetGroup();
unsigned short element = tag.GetElement();
```

2.2 VR (Value Representation)

For each DICOM data element, the VR is a two-character code that specifies the type and formatting of the associated value. Table 1 provides a full list of the VRs that exist within the DICOM standard. VRs generally specify a text format, and in DICOM even numeric values are usually stored as text. Most text VR formats utilize backslash as a separator, e.g. ORIGINAL\PRIMARY indicates two separate strings within a single array. Notable exceptions to this backslash-as-separator rule are the ST, LT, and UT VRs; these are intended to be used to store one or more paragraphs of free-form text in which a backslash is interpreted as a backslash and not a separator. Within these VRs, line breaks between the paragraphs should be indicated with CR NL, never by NL on its own. In comparison, the SH and LO text VRs are meant for short single-line blocks of text and do not allow CR or NL at all. Furthermore, leading and trailing spaces in single-line text VRs are not significant and should be removed before the value is used in an application.

```
// Construct a VR with an invalid initial value.
vtkDICOMVR vr;

// Two ways to create a VR.
vr = vtkDICOMVR("CS");
vr = vtkDICOMVR(vtkDICOMVR::CS);

// Get information about the VR.
const char *name = vr.GetText();
int vtk_type_id = vr.GetType();
```

2.3 Value

The value within a DICOM data element is actually an array of items, where each item is (confusingly) also typically called a value. The VRs discussed above describe the format of each item in the array, where all items in the array must be of the same format. The allowed number of items in the array is called the *value multiplicity* or VM, and for most attribute values defined in the DICOM standard the VM is one.

As stated in the previous section, text items are separated by backslashes, e.g. 1.000\2.000 while items stored in binary such as FL (float) and FD (double) are simply stored in a binary array of their native type. When you index into a value, you get one of the backslash-delimited items if the value is text, or one of the native types if the value is binary.

Within our C++ interface to the DICOM values, we provide automatic conversion between numbers stored as text and native numeric types. So if a number is stored in text as 512 you can retrieve it as an int, or

Table 1: List of DICOM Value Representations (VRs).

VR	VR Name	Format	Example	Type	Length
AE	Application Entity	(see notes 1,2,3)	MyLabNode01	char	16 bytes max
AS	Age String	nnnY (or D,W,M)	038Y	char	4 bytes fixed
AT	Attribute Tag	*binary*		*tag*	4 bytes fixed
CS	Code String	(see note 5)	ORIGINAL	char	16 bytes max
DA	Date	YYYYMMDD	20130604	char	8 bytes fixed
DS	Decimal String	+/-, 0-9, E/e and .	10.2	char	16 bytes max
DT	Date Time	(see note 6)	20130604160259	char	26 bytes max
FL	Float	*binary*		float	4 bytes fixed
FD	Float (Double)	*binary*		double	8 bytes fixed
IS	Integer String	+/-, 0-9	15	char	12 bytes max
LO	Long String	(see notes 1,2,4)		char	64 bytes max
LT	Long Text	(see note 1)		char	10240 bytes max
OB	Other Bytes	*binary*		unsigned char	unlimited
OF	Other Floats	*binary*		float	unlimited
OW	Other Words	*binary*		short	unlimited
PN	Person Name	(see notes 1,2,4)	Doe^John	char	64 bytes max
SH	Short String	(see notes 1,2,4)		char	16 bytes max
SL	Signed Long	*binary*		int	4 bytes fixed
SQ	Sequence	*binary*		*item*	unlimited
SS	Signed Short	*binary*		short	2 bytes fixed
ST	Short Text	(see note 1)		char	1024 bytes max
TM	Time	HHMMSS[.FF...]	4160259.569834	char	16 bytes max
UI	Unique Identifier	0-9, . (see note 7)	1.234.2345	char	64 bytes max
UL	Unsigned Long	*binary*		unsigned int	4 bytes fixed
UN	Unknown	*binary*			unlimited
US	Unsigned Short	*binary*		unsigned short	2 bytes fixed
UT	Unlimited Text	(see note 1)		char	unlimited

1. The default character repertoire for DICOM includes all printing characters in ISO 8859 but no control characters except for CR, NL, FF, and ESC. A new line should always be indicated by CR NL, rather than NL alone. Other character repertoires may only be used if the Specific Character Set (0008,0005) attribute is set for the DICOM data set, and even then the use of the new repertoire is limited to the SH, LO, PN, ST, LT, and UT VRs.
2. Backslash (\ ) is a value separator, and is not permitted in text VRs that allow multiple data values.
3. This VR does not permit any of the control characters CR, NL, FF, or ESC.
4. This VR does not permit CR, NL, or FF, but it does permit ESC.
5. This VR only permits A-Z, 0-9, underscore, and space. Leading/trailing spaces should be ignored if present.
6. The full DT format is YYYYMMDDHHMMSS.FFFFFF+ZZXX where .FFFFFF optionally gives a fraction of a second (up to 6 digits), and +ZZXX (or -ZZXX) optionally gives an offset in hours and minutes from UTC.
7. If the number of characters in a Unique Identifier is odd, it will be padded to even with a null byte.

Table 2: The default DICOM character repertoire.

00–0F	10–1F	20-2F	30–3F	40–4F	50-5F	60–6F	70–7F
		SP	0	@	P	`	p
		!	1	A	Q	a	q
		"	2	B	R	b	r
		#	3	C	S	c	s
		$	4	D	T	d	t
		%	5	E	U	e	u
		&	6	F	V	f	v
		'	7	G	W	g	w
		(	8	H	X	h	x
		)	9	I	Y	i	y
NL		*	:	J	Z	j	z
	ESC	+	;	K	[	k	{
FF		,	<	L	\	l	|
CR		−	=	M	]	m	}
		.	>	N	^	n	~
		/	?	O	_	o	

unsigned int, double, etcetera. And likewise, if a value is stored as short or float, you can retrieve it as a text string. If the conversion is impossible, then the result will be an emptry string (if attempting to convert to a string) or zero (if attempting to convert to a numerical value).

```
// Create an empty, typeless, invalid value.
vtkDICOMValue v;

// Get a value from the meta data.
v = metaData->GetAttributeValue(DC::PixeSpacing);

// Verify that the attribute was present in the meta data.
if (!v.IsValid())
  {
  // attribute not found!
  }

// Verify that the correct number of items are present.
if (v.GetNumberOfValues() == 2)
  {
  double xs = v.GetDouble(0);
  double ys = v.GetDouble(1);
  }

// Get an array of values from the meta data.
v = metaData->GetAttributeValue(DC::ImagePositionPatient);
double position[3] = { 0.0, 0.0, 0.0 };
if (v.GetNumberOfValues() == 3)
  {
  v.GetValues(position, position+3);
  }
```

```
// Get values that have a VM of 1 (preferred method).
std::string name = metaData->GetAttributeValue(DC::Modality).AsString();
int rows = metaData->GetAttributeValue(DC::Rows).AsInt();

// Get pointer to the native array within the value object (very efficient).
// For text VRs, a null-terminated string is always returned.
v = metaData->GetAttributeValue(DC::StudyInstanceUID);
const char *uid = v.GetCharData();
if (uid != 0)
  {
  // Use the pointer only if it isn't NULL!
  // Remember that the pointer becomes invalid when the value destructs.
  }

// Get a value that holds a tag pointer to another value.
vtkDICOMTag tag = metaData->GetAttributeValue(DC::FrameIncrementPointer).GetTag(0);
if (tag != vtkDICOMTag(0,0)) // make sure tag isn't zero
  {
  v = metaData->GetAttributeValue(tag);
  // do something with the referenced value
  }
```

2.4 Accessing meta data

The meta data can be loaded from a DICOM file in two ways: either via the vtkDICOMReader class, or via the vtkDICOMParser class (which reads only the meta data, not the image). Both of these classes will be discussed in more detail in later sections of this document.

DICOM meta data is stored in every file, and because it takes a full series to create a complete 3D image volume in VTK, when querying for DICOM attributes it is often necessary to specify the file you want to query. This, in turn, will depend on which slice of the image volume you are currently viewing. To support this, the GetAttributeValue() methods of the vtkDICOMMetaData class accept a file index as a parameter, where the index is in the range $[0, n-1]$ for a series of n files.

```
// Get meta data from a vtkDICOMReader
reader->UpdateInformation();
vtkDICOMMetaData *meta = reader->GetMetaData();

// Check if an attribute exists, and get it for the first file.
if (meta->HasAttribute(DC::EchoTime))
  {
  int fileIndex = 0;
  double t = meta->GetAttributeValue(fileIndex, DC::EchoTime).AsDouble();
  }

// Get an attribute that will be the same for all files.
std::string str = meta->GetAttributeValue(DC::SeriesDescription).AsString();
```

The data elements are stored in a hash table within the vtkDICOMMetaData object, so accessing them with GetAttributeValue() is very efficient. Also, when the value of an attribute is the same for all files, only one copy of that value is saved in the hash table. This means that vtkDICOMMetaData is a very memory-efficient data structure, even for a very long series. Note that if you do not provide a file index when you call GetAttributeValue(), you will always get the same result as if you provided a file index of zero.

One interesting property of DICOM meta data is that it can be nested, similarly to the way that directories on a file system can have subdirectories. This is used, for example, to store meta data for each frame in an enhanced multi-frame DICOM file. In order to make it easy to access nested attributes, we provide a vtkDICOMTagPath class that can describe the full path to a nested attribute.

```
// Get an attribute for frame 3 of a multi-frame file.
int frameIdx = 3;
double echoTime = meta->GetAttributeValue(
  vtkDICOMTagPath(DC::PerFrameFunctionalGroupSequence, frameIdx,
                  DC::CardiacSynchronizationSequence, 0,
                  DC::NominalCardiacTriggerDelayTime)).AsDouble();
```

This is not the only way to access per-frame attributes. It is also possible to specify the frame index after the file index, in which case the GetAttributeValue() method will perform a search for the attribute without requiring a full path.

```
// Get an attribute for frame 3 of a multi-frame file.
int fileIdx = 0;
int frameIdx = 3;
vtkDICOMValue vw = meta->GetAttributeValue(fileIdx, frameIdx, DC::WindowWidth);
vtkDICOMValue vc = meta->GetAttributeValue(fileIdx, frameIdx, DC::WindowCenter);
if (vw.IsValid() && vc.IsValid())
  {
  // set the window for the image
  }
```

2.5 Low-level data element access

The vtkDICOMMetaData object also provides container-style access to the data elements. This is useful, for instance, when you want to iterate through all of the elements in the meta data. It is also useful if you want an efficient way to check whether an attribute varies between files in the series. However, these are the only cases in which you should use this low-level style of access. In general, you should use the more convenient GetAttributeValue() method, which is just as efficient.

```
// Iterate through all data elements in the meta data.
for (vtkDICOMDataElementIterator iter = meta->Begin(); iter != meta->End(); ++iter)
  {
  vtkDICOMTag tag = iter->GetTag();
  std::cout << "tag: " << tag << std::endl;
  // crucial step: check for values that are multiplexed across the series
  if (iter->IsPerInstance())
    {
    int n = iter->GetNumberOfInstances();
    for (int i = 0; i < n; i++)
      {
      std::cout << "instance " << i << ": " << iter->GetValue(i) << std::endl;
      }
    }
  else
    {
    // not multiplexed: same value across all files in series
    std::cout << "all instances: " << iter->GetValue() << std::endl;
```

```
    }
}

// Get the iterator to a specific element (hash table lookup).
vtkDICOMDataElementIterator iter = meta->Find(DC::ImageOrientationPatient);
if (iter != meta->End())
  {
  // do something
  }
```

2.6 Dictionaries

When iterating through the data elements in the meta data, as described in the previous section, it can be useful to get information about the meaning of the data elements that are encountered. In-depth information can only be provided by the DICOM standards documents themselves, but the vtkDICOMDictionary can at least provide a summary of what kind of data to expect for a given attribute.

```
// do a dictionary lookup
vtkDICOMDictEntry entry;
entry = vtkDICOMDictionary::FindDictEntry(vtkDICOMTag(0x0008,0x0020));
// check if entry was found in dictionary
if (entry.IsValid())
  {
  std::cout << entry.GetName() << std::endl; // prints "Study Date"
  std::cout << entry.GetVR() << std::endl; // prints "DA"
  std::cout << entry.GetVM() << std::endl; // prints "1"
  }
```

The vtkDICOMDictionary class provides information for attributes that are described in the DICOM standard, but it cannot directly provide information for private attributes that are defined by medical device manufacturers. Every DICOM file is likely to have a mix of standard attributes and private attributes. Fortunately, it is easy to tell the difference between the two: private attributes always use a tag with with an odd group number, while the DICOM standard only uses even group numbers.

Due to the particulars of how private data is stored in a DICOM file, lookup of private information must be done with an existing vtkDICOMMetaData object, because the company/organization that owns the information will be listed in the meta data. So if you need to identify a private attribute, you must use the vtkDICOMMetaData's FindDictEntry() method instead of using the vtkDICOMDictionary method.

```
// do a dictionary lookup of a tag that might be private
vtkDICOMMetaData *meta = reader->GetMetaData();
vtkDICOMDictEntry entry = meta->FindDictEntry(vtkDICOMTag(0019,107E));
// check if entry was found in public dictionary or in any private dictionary
if (entry.IsValid())
  {
  // do something
  }
```

To complicate matters, if the private tag is present within nested meta data, it is necessary to perform the lookup with the full path to the attribute (see the discussion of vtkDICOMTagPath in Section 2.4). Support of private tags in the vtkDICOM library is still evolving.

3 Reading DICOM Files

3.1 Grouping files together

The first difficulty that one often runs into when trying to load DICOM files into VTK is that the files themselves do not usuall have descriptive names. With a directory full of DICOM images it can be difficult to know which ones to load. A typical DICOM folder listing looks something like this:

```
IM-0001-0001.dcm    IM-0001-0005.dcm    IM-0001-0009.dcm    IM-0001-0013.dcm
IM-0001-0002.dcm    IM-0001-0006.dcm    IM-0001-0010.dcm    IM-0001-0014.dcm
IM-0001-0003.dcm    IM-0001-0007.dcm    IM-0001-0011.dcm    IM-0001-0015.dcm
IM-0001-0004.dcm    IM-0001-0008.dcm    IM-0001-0012.dcm    IM-0001-0016.dcm
```

These files might be 16 slices of a 3D image, or the first three files might be locator images while the remaining 13 files are slices of a 3D volume, or they might be something else entirely. So the first thing to do with a batch of DICOM files is to find out how they fit together. The vtkDICOMSorter performs this task. Given a set of DICOM files, it will discover which files belong to the same DICOM series. Each series will generally be one of the following:

1. A stack of slices.
2. A series of sequential frames.
3. A combination of the above (multi-dimensional).

When we read DICOM images into VTK, we want to load just one series of images (or sometimes just a portion of a series) as a VTK data set. The vtkDICOMSorter makes this easy.

```
// Instantiate a DICOM sorter.
vtkSmartPointer<vtkDICOMSorter> sorter =
  vtkSmartPointer<vtkDICOMSorter>::New();

// Provide an array containing a list of filenames.
sorter->SetInputFileNames(filenames);

// Update the sorter (i.e. perform the sort).
sorter->Update();

// Get the first series.
int i = sorter->GetNumberOfSeries();
if (i > 0)
  {
  vtkStringArray *sortedFiles = sorter->GetFileNamesForSeries(0);
  // do something with the files
  }
```

In addition, the sorter can discover which series belong to the same study. That is, it can tell us which series were collected during the same imaging session. One thing the sorter does *not* do is sort the images in the series according to slice location. It only sorts the images according to the Instance Number embedded in each image, where the Instance Number gives the logical viewing order prescribed by the medical device that generated the images. It is up to the vtkDICOMReader to check the slice positions for the files and sort them by location before generating an image volume or time series.

```
// Sort the input filenames by series and study.
sorter->SetInputFileNames(filenames);
sorter->Update();

// Iterate through all of the studies that are present.
int n = sorter->GetNumberOfStudies();
for (int i = 0; i < n; i++)
  {
  // Iterate through all of the series in this study.
  int j = sorter->GetFirstSeriesInStudy(i);
  int m = sorter->GetNumberOfSeriesInStudy(i);
  for (int k = 0; k < m; k++)
    {
    vtkStringArray *sortedFiles = sorter->GetFileNamesForSeries(j+k);
    // do something with the files
    }
  }
```

3.2 Reading a series of files as a VTK image

The design goal for the vtkDICOMReader is to convert a series of DICOM files into a geometrically accurate vtkImageData object. That is, the pixel spacing and the center-to-center distance between adjacent slices in the image data is as specified in the DICOM meta data. Furthermore, a $4{\times}4$ matrix is provided that can be used to position and orient the image. In order to achieve this, the reader checks the Image Position and Image Orientation that are recorded in the meta data for each slice. Then, if and only if the image positions line up along the normals of the slice planes, the reader sorts the slices according to location. The vtkImageData z spacing is then set to the average center-to-center distance between adjacent slices.

In the absence of Image Position information in the meta data, or if the slices do not form a rectilinear volume, then the slices are sorted only according to the Instance Number in the meta data. There is also a reader method called SortingOff() than can be called to disable sorting entirely, so that the order of the slices in the vtkImageData will reflect the order of the list of files provided to the reader.

In addition to sorting slices by location, the reader attempts to detect multi-dimensional data sets. It recognizes up to 5 dimensions: x, y, z, t, and a vector dimension. This is best illustrated by example. If a the DICOM series provides real and imaginary data slices at each location, then the vtkImageData produced by the reader will have two components. We interpret this as the image having a vector dimension of 2.

When a time dimension is present, things become interesting. The default behavior of the reader is to store adjacent time points in adjacent vtkImageData slices. This works well when the images are to be displayed slice-by-slice. It is, however, inappropriate if the vtkImageData is to be displayed as a multi-planar reformat or as a volume. For this reason, the vtkDICOMReader has a method called TimeAsVectorOn() that will cause the reader to treat each voxel as a time vector. In other words, if the DICOM data has 10 individual time slots, then the vtkImageData will have 10 components per voxel (or 30 components in the case of RGB data). By selecting a specific component or range of components when displaying the data, one can display a specific point in time.

Five dimensions come into play when the DICOM series has slices that are at the same location and within the same time slot. Going back to the (real,imaginary) example, if such a series of images is read after TimeAsVectorOn() is called, then the vtkImageData will have 20 components per voxel if there are 10 time slots. The 20 components can be thought of as 10 component blocks with 2 components per block. A filter like vtkImageExtractComponents can be used to extract a block of components that corresponds to a particular time slot.

If the behavior described in the preceding paragraphs is not desirable, then one can use the SetDesiredTimeIndex(int) method to read just one time slot, and use a set of *N* readers to read the *N* time slots as *N* separate VTK data sets.

```
// Instantiate the reader
vtkSmartPointer<vtkDICOMReader> reader =
  vtkSmartPointer<vtkDICOMReader>::New();

// Provide a vtkStringArray containing a list of filenames.
reader->SetFileNames(filenames);

// Read the meta data via UpdateInformation()
reader->UpdateInformation();

int numberOfTimeSlots = reader->GetTimeDimension();
if (numberOfTimeSlots > 1)
  {
  // for example, read only the final time slot
  reader->SetDesiredTimeIndex(numberOfTimeSlots-1);
  }

// Update the reader
reader->Update();
```

3.3 Multi-frame and multi-stack files

The DICOM standard allows for multiple slices (frames) per file, and even for multiple stacks of slices per file. In the case of multi-frame files, each frame is assigned a position and a time slot and the frames are sorted according to the slice sorting method described in the previous section.

In multi-stack files there are, as one might expect, more than one rectilinear (or perhaps non-rectilinear) volume. If sorting has been turned off with the SortingOff() method, then all the frames in the file are read sequentially into vtkImageData slices. If sorting is on, however, then the reader is only able to read one stack at a time. The method SetDesiredStackID() allows one of the stacks to be chosen by name.

```
// Instantiate the reader
vtkSmartPointer<vtkDICOMReader> reader =
  vtkSmartPointer<vtkDICOMReader>::New();

// Provide a multi-frame, multi-stack file
reader->SetFileName(filename);

// Read the meta data, get a list of stacks
reader->UpdateInformation();
vtkStringArray *stackNames = reader->GetStackIDs();

// Specify a stack, here we assume we know the name:
reader->SetDesiredStackID("1");
```

Sometimes one will find a DICOM series that contains slices that are implicitly arranged into separate stacks, even though there is no information in the meta data that explicitly assigns each slice to a stack.

In this situation, the vtkDICOMReader will synthesize the stack information by grouping blocks of slices together if that form rectilinear volumes, and it will name the blocks with decimal strings "0", "1", etc. This is particularly useful for the localizer series that is present in most MRI studies, since the localizer contains separate blocks of axial, coronal, and sagittal images.

3.4 DICOM image orientation

When people use the term "image orientation" with respect to medical images, they usually mean one or more of the items listed below:

1. The order in which the pixels and slices are stored in the computer's memory.
2. The orientation of the image slices in a real-world coordinate system, for example the patient coordinate system as defined by the medical imaging equipment that generated the images.
3. The orientation of the patient with respect to the viewer when the images are viewed on a workstation.

The first of these, the way the data is stored in memory, is an implementation detail. Or at least it should be. Unfortunately one image display class in VTK, the vtkImageViewer, insists that the pixels must be arranged in memory such that the image pixel at the bottom-left corner of the image is the pixel at the lowest memory address. This is in conflict with the way that DICOM files stores images with the top-left pixel as the first pixel in the file. Because the vtkImageViewer insists on a particular ordering of the pixels in memory, it should be avoided in applications where image orientation is important.

The second and third items in the list can be referred to as the real-world orientation, and the display orientation, respectively. Neither of these can be considered an implementation detail, as both of them are crucial to the user experience. Also, it is important not to confuse one with the other. An application can handle the real-world orientation incorrectly but still display the images to the user in the correct orientation, but such an application would certainly have a serious flaw.

The real-world orientation is provided by the GetPatientMatrix() method of the vtkDICOMReader. This method returns a vtkMatrix4x4 object that describes the coordinate transformation from the data coordinates of the vtkImageData that stores the image, to the real-world Patient Coordinate System defined in the DICOM standard [1]. The matrix is used to correctly place the image in the VTK world coordinate system.

```
reader->Update(); // update the reader
vtkMatrix4x4 *matrix = reader->GetPatientMatrix();

// Create an image actor and specify the orientation.
vtkSmartPointer<vtkImageActor> actor =
  vtkSmartPointer<vtkImageActor>::New();
actor->GetMapper()->SetInputConnection(reader->GetOutputPort());
actor->SetUserMatrix(matrix);
```

Setting the actor's UserMatrix will ensure that the real-world orientation of the image is correctly handled, as far as the VTK display pipeline is concerned. It does not, however, set the display orientation, which is the responsibility of the application. The display orientation can be set via manipulation of the VTK camera, or via certain methods in the vtkInteractorStyleImage class. The details of how this is done are outside of the scope of this document.

3.5 Obtaining per-slice meta data

A direct consequence of the sorting that is performed by the vtkDICOMReader is that the slices in the vtkImageData are not guaranteed to be in the same order as the files that were given to the reader's SetFileNames() method. Fortunately, the reader provides a look-up table that can be used to find out which file provided which slice. Hence, given a VTK slice number, the look-up table can provide the corresponding file, and then the meta data can be inspected for that particular file. Similarly, for multi-frame DICOM files, the reader provides a look-up table that gives the DICOM frame that provided each each slice. The vtkMetaData object provides a GetAttributeValue() method that takes both a file index and a frame index, along with the tag of the attribute to be inspected.

```
// Read the files and get the meta data.
reader->SetFileNames(fileNameArray);
reader->Update();
vtkDICOMMetaData *meta = reader->GetMetaData();

// Get the arrays that map slice to file and frame.
vtkIntArray *fileMap = reader->GetFileIndexArray();
vtkIntArray *frameMap = reader->GetFrameIndexArray();

// Get the file and frame for a particular slice.
int sliceIndex = 6;
int fileIndex = fileMap->GetComponent(sliceIndex, 0);
int frameIndex = frameMap->GetComponent(sliceIndex, 0);

// Get the position for that slice.
vtkDICOMValue pv = meta->GetAttributeValue(fileIdx, frameIdx,
  DC::ImagePositionPatient);
double position[3] = { 0.0, 0.0, 0.0 };
if (pv.IsValid() && pv.GetNumberOfValues() == 3)
  {
  pv.GetValue(position, position+3);
  }
```

As a caveat, for multi-frame files, the example given above assumes that the meta data contains a per-frame ImagePositionPatient attribute. This is the case for multi-frame CT and MRI files, but not for multi-frame nuclear medicine files. Whenever retrieving meta data from a DICOM image, it is wise to consult the DICOM standard to see how the attributes are defined for the various modality-specific IODs (information object descriptions).

3.6 Obtaining per-component meta data

In the example in the previous section, the GetComponent() method was called with two arguments, but the second argument was set to zero. If the vtkDICOMReader assigned a vector dimension to the data, then the the vtkImageData will have multiple scalar values in each voxel. For instance, the first component in each voxel may have come from a file that provided the real component of a complex-valued image, while the second component came from a file that provided the imaginary component. In this case, one would do the following to retrieve the meta data from the “imaginary” file:

```
// Read the files and get the meta data.
reader->SetFileNames(fileNameArray);
reader->Update();
vtkDICOMMetaData *metaData = reader->GetMetaData();

// Get the arrays that map slice to file and frame.
vtkIntArray *fileMap = reader->GetFileIndexArray();
vtkIntArray *frameMap = reader->GetFrameIndexArray();

// Get the file and frame for a particular slice and component.
int sliceIndex = 6;
int vectorIndex = 1; // 2nd component is the imaginary component
int fileIndex = fileMap->GetComponent(sliceIndex, vectorIndex);
int frameIndex = frameMap->GetComponent(sliceIndex, vectorIndex);

// Get an attribute from the meta data.
vtkDICOMValue v = metaData->GetAttributeValue(fileIdx, frameIdx, tag);
```

If the data has a time dimension and the reader's TimeAsVectorOn() method was called, then the components of each voxel can correspond both to a specific time slot, and to a specific vector component. To make the situation even more complicated, each pixel in the DICOM files might be an RGB pixel and therefore have three components as given by the SamplesPerPixel attribute in the meta data.

The number of components in the FileIndexArray and FrameIndexArray is equal to the vector dimension, and if TimeAsVectorOn() was called, then the vector dimension will include the time dimension. The FileIndexArray and FrameIndexArray do not have components that correspond to the individual R,G,B components in RGB images, since the R, G, and B components will always have the same meta data because they always come from the same file and frame.

The following strategy is recommended for accessing per-component meta data in multi-dimensional images:

```
// Get the arrays that map slice to file and frame.
vtkIntArray *fileMap = reader->GetFileIndexArray();
vtkIntArray *frameMap = reader->GetFrameIndexArray();

// Get the image data and meta data.
vtkImageData *image = reader->GetOutput();
vtkDICOMMetaData *meta = reader->GetMetaData();

// Get the number of components in the data.
int numComponents = image->GetNumberOfScalarComponents();

// Get the full vector dimension for the DICOM data.
int vectorDimension = fileMap->GetNumberOfComponents();

// Compute the samples per pixel in original files.
int samplesPerPixel = numComponents/vectorDimension;

// Check for time dimension
int timeDimension = reader->GetTimeDimension();
if (timeDimension == 0)
  {
  timeDimension = 1;
  }
```

```
// Get all attributes for a specific time.
int vectorIndex = timeIndex*vectorDimension/timeDimension;
int vectorEndIndex = (timeIndex + 1)*vectorDimension/timeDimension;

for (int i = vectorIndex; i < vectorEndIndex; i++)
  {
  int fileIndex = fileMap->GetComponent(sliceIndex, i);
  int frameIndex = frameMap->GetComponent(sliceIndex, i);
  vtkDICOMValue v = meta->GetAttributeValue(fileIdx, frameIdx, tag);
  // print or display the value
  }

// Extract an image at the desired time slot (e.g. for display).
int componentIndex = timeIndex*vectorDimension/timeDimension*samplesPerPixel;
vtkSmartPointer<vtkImageExtractComponents> extractor =
  vtkSmartPointer<vtkImageExtractComponents>::New();
extractor->SetInputConnection(reader->GetOutputPort());
if (samplesPerPixel == 1)
  {
  extractor->SetComponents(componentIndex);
  }
else if (samplesPerPixel == 2) // rare/nonexistent in DICOM images
  {
  extractor->SetComponents(componentIndex, componentIndex + 1);
  }
else
  {
  extractor->SetComponents(componentIndex, componentIndex + 1, componentIndex + 2);
  }
extractor->Update();
```

4 Writing DICOM Files

4.1 Conforming to the standard

The DICOM standard is expansive, with approximately 1500 pages dedicated to how to put together DICOM data sets and write them as DICOM files. Some parts of conformance are simple, for example making sure that all values stored in the file match with the value representations described in Table 1. Other parts are are more complex, such as ensuring that a DICOM file that contains an MR image also contains all of the meta data that is required for an MR image.

A crucial facet of conformance is that every DICOM image must have a unique identifier or UID. No two images anywhere in the world are permitted by the standard to have the same UID, unless they are in fact the same image from the same source. The default method that the vtkDICOMWriter uses to generate unique IDs is to append a special 128-bit random number called a UUID to the special UID prefix “2.25.” Through the use of such a large random number, the probability that exactly the same number will be generated twice is vanishingly small. An example of such a UID is as follows:

2.25.1267999611695479980069765583325507254

If you have registered a UID prefix for your organization, then you can use it in the vtkDICOM library as shown in the following example. If your prefix is too long to allow for a 128-bit suffix, then a 96-bit suffix will be used instead.

```
// Set the UID prefix for all generated DICOM files (default 2.25.)
vtkDICOMUtilities::SetUIDPrefix("1.2.XXX.YYYYY.");

// Set a UID that identifies your software to the world.
vtkDICOMUtilities::SetImplementationClassUID("1.2.XXX.YYYYY.ZZZZZZZZZZZZZZZZZZZZ");

// Set the name and version number of your software.
vtkDICOMUtilities::SetImplementationVersionName("SOFTWARE VER X_Y_Z");
```

In addition to having a unique ID, every DICOM data set must conform to one of the classes identified in the DICOM standard. A DICOM "class" is described in the standard by an IOD (Information Object Description) that states which attributes are to be present for that class. A CT image, for example, must belong to one of the CT classes, and must contain information such as the energy of the X-rays that were used to make the image.

4.2 Writing a VTK image as DICOM

The vtkDICOMWriter takes a vtkImageData object as input, and writes a series of DICOM image files to disk. Since the required meta data for an image varies from one modality to another, the writer delegates the creation of the meta data to another class called a vtkDICOMGenerator. A short example of how this is done is as follows:

```
// Create a generator for MR images.
vtkSmartPointer<vtkDICOMMRGenerator> generator =
  vtkSmartPointer<vtkDICOMMRGenerator>::New();

// Create a meta data object with some desired attributes.
vtkSmartPointer<vtkDICOMMetaData> meta =
  vtkSmartPointer<vtkDICOMMetaData>::New();
meta->SetAttributeValue(DC::PatientName, "Doe^John");
meta->SetAttributeValue(DC::ScanningSequence, "GR"); // Gradient Recalled
meta->SetAttributeValue(DC::SequenceVatiant, "SP"); // Spoiled
meta->SetAttributeValue(DC::ScanOptions, "");
meta->SetAttributeValue(DC::MRAcquisitionType, "2D");

// Plug the generator and meta data into the writer.
vtkSmartPointer<vtkDICOMWriter> writer =
  vtkSmartPointer<vtkDICOMWriter>::New();
writer->SetInputConnection(lastFilter->GetOutputPort());
writer->SetMetaData(meta);
writer->SetGenerator(generator);

// Set the output filename format as a printf-style string.
writer->SetFilePattern("%s/IM-0001-%04.4d.dcm");
// Set the directory to write the files into.
writer->SetFilePrefix("/the/output/directory");

// Write the file.
writer->Write();
```

The vtkDICOMMRGenerator assists with conformance by generating all the data set attributes that are required by the MR IOD. It will also scan through the vtkMetaData object that is provided to the writer, and use any of its attributes as long as 1) they are defined in the MR IOD, and 2) they are deemed to be valid for the image that is being written. A partial list of attributes that are never taken from the input meta data is as follows:

1. SOPInstanceUID (this is always re-generated to ensure its uniqueness)
2. SeriesInstanceUID (ditto)
3. ImageType (this is set to DERIVED\SECONDARY\OTHER by default)
4. PixelSpacing (this is set from the VTK image information)
5. Rows and Columns (ditto)
6. ImagePositionPatient and ImageOrientationPatient (these are set from the PatientMatrix)

The generator always creates a new SOPInstanceUID for each file and a new SeriesInstanceUID for each series. There is no way to set these UIDs manually. The ImageType is set to DERIVED by default, because an image cannot be considered to be ORIGINAL if it was modifed in any way after its original acquisition. Finally, all information related to the pixel values or the slice geometry is generated from the vtkImageData information and from the PatientMatrix.

The vtkDICOMWriter allows several parameters, including ImageType, to be set when writing the file. These are demonstrated in the following example.

```
// Plug the generator and meta data into the writer.
vtkSmartPointer<vtkDICOMWriter> writer =
  vtkSmartPointer<vtkDICOMWriter>::New();
writer->SetInputConnection(lastFilter->GetOutputPort());
writer->SetMetaData(meta);
writer->SetGenerator(generator);

// Set the output filename format as a printf-style string.
writer->SetFilePattern("%s/IM-0001-%04.4d.dcm");
// Set the directory to write the files into
writer->SetFilePrefix("/the/output/directory");

// Set the image type to Multi-planar Reformat.
// (forward slashes will be converted to backward slashes)
writer->SetImageType("DERIVED/SECONDARY/MPR");
writer->SetSeriesDescription("Sagittal Multi-planar Reformat");

// Set the 4x4 matrix that gives the position and orientation.
writer->SetPatientMatrix(patientMatrix);
```

4.3 Customizing the generators

At the present time, the vtkDICOMWriter has only three generators available: the vtkDICOMMRGenerator (for MR), the vtkDICOMCTGenerator (for CT), and the vtkDICOMSCGenerator (for Secondary Capture, e.g. screenshots). Writing a new generator class is the recommended method for adding support for a new modality to the vtkDICOM library.

4.4 Writing a raw data set

In addition to the vtkDICOMWriter, there is a class called the vtkDICOMCompiler that can write meta data and image data directly to a file without it being processed by a vtkDICOMGenerator. It can be used to efficiently perform such actions as changing the transfer syntax of the data or tweaking the meta data. By design, the vtkDICOMCompiler will take, as input, a meta data object that describes a series of images, and it will then write the files in the series one-by-one.

```
vtkSmartPointer<vtkDICOMCompiler> compiler =
  vtkSmartPointer<vtkDICOMCompiler>::New();
compiler->SetMetaData(meta);

const char *outputDirectory = "/some/directory";
int n = meta->GetNumberOfInstances();
for (int i = 0; i < n; i++)
  {
  char outputFile[256];
  sprintf(outputFile, "%s/IM-0001-%04.4d.dcm", outputDirectory, i+1);
  compiler->SetFileName(outputFile);
  compiler->SetIndex(i);
  compiler->WriteHeader();
  compiler->WritePixelData(rawPixelBufferForFile);
  }
```

5 Future Work

The vtkDICOMReader is also missing two key features that are required for correct image presentation: it does not provide the lookup table for images that have one, nor does it provide the overlay planes that are meant to be displayed with the images. Yet another omission is support for private private attribute dictionaries, which will be addressed in a future release. The vtkDICOMWriter will be extended so that it supports additional modalities, and might eventually be generalized to support all modalities.

It is also worth noting two features that are not scheduled at this time: support for the DICOM networking protocol, and inclusion of DICOM-specific image display classes for VTK. The former is something that we might add in the future if it becomes one of our internal requirements, while the latter is beyond the scope of this project and, if released, will be part of a separate project.

References

[1] Medical Imaging & Technology Alliance. DICOM Base Standard – 2011. Published by the Association of Electrical and Medical Imaging Equipment Manufacturers (NEMA), 2011.

Appendix A: Building the vtkDICOM library

The vtk-dicom project is on github, at the following url:

```
http://github.com/dgobbi/vtk-dicom
```

The best way to access the source clode is to click on the “Download ZIP” link on the right side of the web page. The following procedure can be used to build the vtkDICOM library, assuming that you have already built VTK with CMake. This example is for building the package on Linux or OS X in the bash shell.

```
$ # specify the directory where you built VTK, this is an example
$ export VTK_DIR=/Volumes/HD2/vtk-release-build/
$ unzip vtk-dicom-master.zip
$ mkdir vtk-dicom-master-build
$ cd vtk-dicom-master-build
$ cmake -D CMAKE_BUILD_TYPE:STRING=Release ../vtk-dicom-master
$ make
```

Optionally, the following cmake variables can be set with ccmake or CMakeSetup:

```
BUILD_EXAMPLES              ON
BUILD_PROGRAMS              ON
BUILD_SHARED_LIBS           OFF
BUILD_TESTING               ON
USE_DCMTK                   OFF
USE_GDCM                    OFF
VTK_DIR                     /Volumes/HD2/vtk-release-build/
```

The `USE_DCMTK` and `USE_GDCM` variables allow you to add the image decompression capabilities of either of these packages (do not specify both!) to the vtkDICOMReader. If you do not specify one of these packages, then the vtkDICOMReader will only be able to read uncompressed files.

If you want to use DCMTK, then download dcmtk-3.6.0 from http://dicom.offis.de/ and build it using the the instructions provided on the website. If possible, use the same compiler that you use for vtkDICOM.

If you want to use GDCM, it can be found at http://sourceforge.net/projects/gdcm/. Pay particular attention to the GDCM build instructions, especially the “make install” step, or the vtkDICOM library will not be able to link to the GDCM libraries. Additional information on GDCM can be found at http://gdcm.sourceforge.net/wiki/index.php/Main_Page.

The easiest way to use the vtkDICOM library in your own project is to add the following command block to the main CMakeList.txt file in your project:

```
find_package(DICOM QUIET)
if(DICOM_FOUND)
  include(${DICOM_USE_FILE})
endif()
set(VTK_DICOM_LIBRARIES vtkDICOM)
```

It is not recommended to try to use vtkDICOM (or VTK itself) within projects that are not built with cmake.