UVF - UNIFIED VOLUME FORMAT: A GENERAL SYSTEM FOR EFFICIENT HANDLING OF LARGE VOLUMETRIC DATASETS

Jens Krüger

Kristin Potter

Rob S. MacLeod

Christopher Johnson

Scientific Computing and Imaging Institute
University of Utah

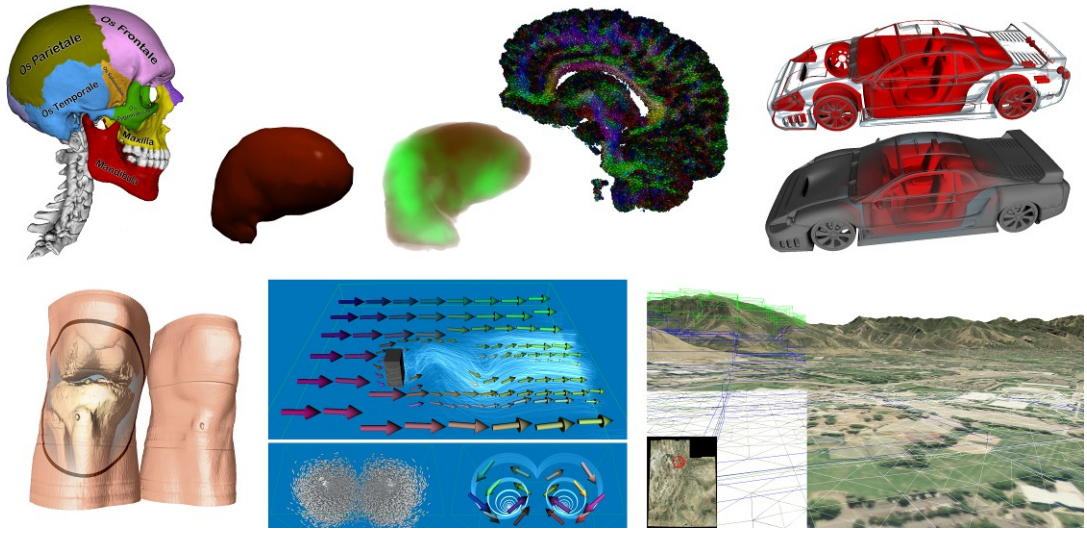

Figure 1. Some example datasets that are represented and stored in the UVF system. Examples include structured scalar, vector and tensor data as well as unstructured triangular and tetrahedral data. In these examples the sizes vary from a megabyte to multiple terabytes.

ABSTRACT

With the continual increase in computing power, volumetric datasets with sizes ranging from only a few megabytes to petascale are generated thousands of times per day. Such data may come from an ordinary source such as simple every-day medical imaging procedures, while larger datasets may be generated from cluster-based scientific simulations or measurements of large scale experiments. In computer science an incredible amount of work worldwide is put into the efficient visualization of these datasets. As researchers in the field of scientific visualization, we often have to face the task of handling very large data from various sources. This data usually comes in many different data formats. In medical imaging, the DICOM standard is well established, however, most research labs use their own data formats to store and process data. To simplify the task of reading the many different formats used with all of the different visualization programs, we present a system for the efficient handling of many types of large scientific datasets (see Figure 1 for just a few examples). While primarily targeted at structured volumetric data, UVF can store just about any type of structured and unstructured data. The system is composed of a file format specification with a reference implementation of a reader. It is not only a common, easy to implement format but also allows for efficient rendering of most datasets without the need to convert the data in memory.

KEYWORDS

scientific visualization, large datasets, petascale computing, interactive rendering

1. INTRODUCTION AND RELATED WORK

Mainly thanks to the evolution of the Internet, well-established standards for image raster and vector as well as audio data exist. Volumetric data, however, is hardly seen in web applications and thus no unified file format exists in the industry. Therefore, practically all scientific projects use custom file formats. The only area outside of scientific labs that routinely creates volumetric datasets is medical imaging. Thus, in the early nineties an open standard called Digital Imaging and Communications in Medicine (DICOM) [NEMA 1992] was proposed and is used today in practically all medical imaging devices. While this standard specifies a well-defined set of key/value pairs stored in a file together with the raster data of one or multiple slices of the data volume, many vendors add their own extension to the format. Often, system implementations are specialized for vendor-specific format rendering, leaving DICOM files incompatible amongst vendors. Furthermore, the DICOM standard does not attempt to store the data in an efficient manner for 3D image generation. Thus, most rendering systems load the possibly many DICOM files that make up a 3D volume into memory and generate a renderable representation out of the data. As long as the datasets are small enough to fit into main memory this does not pose a problem, except for the complex task of implementing a DICOM reader that is able to handle files from all vendors. However, if the size of the data reaches terabyte scale or beyond real-time data conversion is out of the question. Therefore, most scientific projects use custom file formats with simple, yet efficient storage that is easy to implement and allows for immediate rendering. Since the design of these data formats is usually not considered an integral part of the research very few open formats, let alone publications, exist. The well documented formats other than DICOM we were able to find are the NRRD format [Kindlmann 1998a] which is part of the teem [Kindlmann 1998b] framework, the Visualization Toolkit (VTK) File Formats [Kitware 1999], and HDF5 [The HDF Group 2008]. The alternative to these formats is to store mostly raw data to disk with some way to store the metadata, for example using text or XML based description files or a special file name convention to recognize the data.

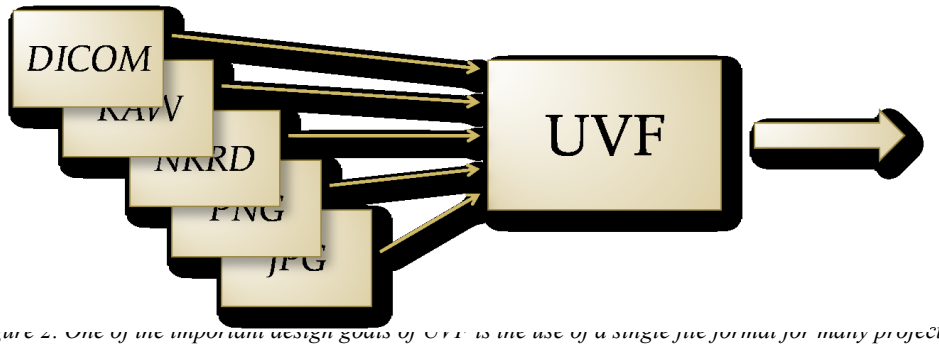

Figure 2: One of the important design goals of UVF is the use of a single file format for many projects and to implement a large variety of converters into this single format.

While all of the formats presented above have their rightful place in science and engineering, in many cutting edge research projects they have proven to be unsuitable for several reasons. First of all, formats such as DICOM or the VTK format and in particular HDF5 are fairly complex to implement while other formats such as file name based metadata or NRRD may be too simple to store the necessary data and metadata. Also, practically all of the formats are designed for easy data exchange only and do not consider the rendering subsystems. Facing these issues over and over again for each new project, we decided to design a new format that satisfies the needs not only of our applications but is also flexible enough to be useful to many other researchers worldwide. Furthermore, the format and the associated framework makes the transition from proprietary formats to the unified format easy by providing a set of data converters as well as an extendable API to easily add new converters for proprietary in-house formats (see Figure 2).

The remainder of this paper is structured as follows: In the next section we explain the basic design goal of UVF and the associated software implementation. In Section 3 we dive into the details of the UVF format

and in Section 4 we briefly describe the conversion from other well-known formats into UVF. Finally, we conclude the paper with a discussion and future work section.

2. DESIGN GOALS

The overall goals for this projects are to design a file format with an associated C++ framework that is easy to read (i.e. less than a thousand lines of code to read a file), accommodates the needs of practically all projects based on volumetric data, and is extensible while making sure not to break older projects in the future. Additionally, the format should also be able to store additional data other than only the structured volumes, such as acceleration structures, annotations, provenance etc.

These basic goals lead to the following requirements, which UVF must support:

1. *virtually unlimited size volumes*
   i.e. data that fits into 64 bit address space
2. *arbitrary domain dimensions*
   such as 2D images, 3D volumes, 4D time dependent volumes, etc.
3. *arbitrary data dimensions*
   such as scalars, vectors, tensors, stochastic functions
4. *arbitrary data formats*
   such as 8-bit, 16-bit, float, double, as well as user defined formats
5. *spatial subdivision and level of detail must be considered by the file format*
6. *be extensible while persevering downward compatibility*
7. *platform independence*
8. *both the payload and the header are to be stored as binary*
9. *the data and the metadata should be kept in a single file*

While the reasons behind points 1-7 should be clear the last two points need some further explanation. The reasoning behind storing everything in binary including the header is twofold. On the one hand, this simplifies the implementation of the parser and increases its performance. On the other hand, this unifies the file and memory layout, eliminating the need for conversion. However, this renders our files non-human-readable, unless the human is familiar with a hex editor. But, since we are dealing with large datasets a direct inspection of the data by a human is not feasible.

The final design decision to put all the data and the header in a single large file is motivated by two reasons. First, this makes transfer of the data easier as header and data cannot be separated accidentally. Second, the reader can access all the data and metadata by simply mem-mapping the file and stepping through binary memory. This mem-mapping procedure requires the logical address space of the target system to be large enough to hold the entire file, which may pose a problem on 32-bit machines. While in the context of the processing of large datasets, these systems will probably die out fairly soon, the format still supports efficiently stepping through the data without having to mem-map it as a whole. We therefore have a special 32-bit implementation of the reader.

3. REALIZATION

In the previous section we discussed the general ideas, and we will now step into the implementation of the format. This will begin by starting at a high level perspective and then getting into more details.

3.1 Global File Structure

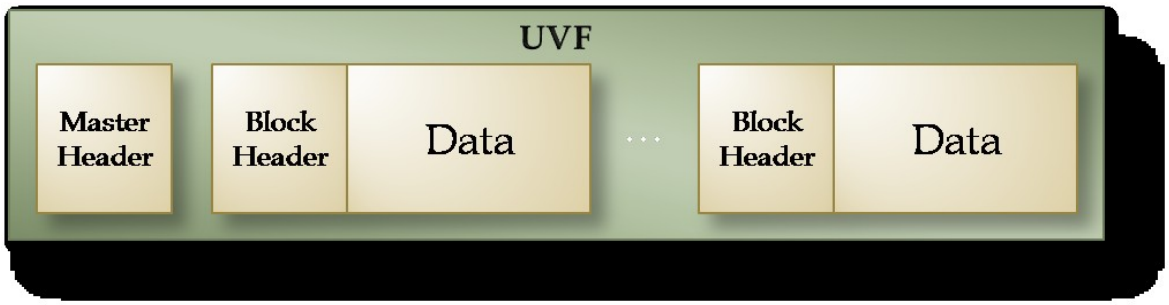

Figure 3.The basic structure of an UVF File

The global structure of a UVF file is depicted in Figure 3. As can be seen, the file starts with a *master header*. This very small data structure allows for the detection of the file type and contains very general information such as file version, checksum, endianess, and a pointer to the first data block. This master header is followed by a list of *block header*/*data* tuples. In this list each block header contains information about the associated data, and a pointer to the next block. The reasoning behind this basic structure is to allow for multiple datasets and data types to be stored in a single file, which is important for visualizations that require very different datasets to be loaded at once (see Figure 4. for an example). Additionally, the design of the file allows for applications that do not have the means to display or use one of the data tuples to simply skip over the unused data and advance to the next block. This not only makes future extensions easy to implement without breaking existing applications, it also allows for the implementation of mini-readers/writers. For example, a reader/writer subsystem that does not support the entire standard will still able to read any given file without failure; the reader may end up detecting that the file does not contain usable data but it will never fail to read it.

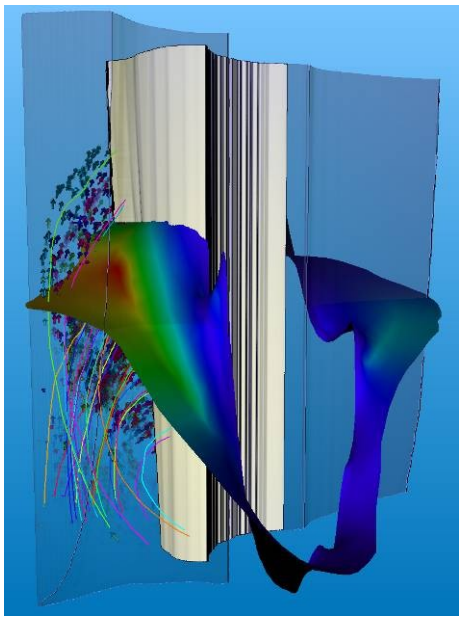

Figure 4. This probability density function requires a triangular mesh, a scalar-, and a vector-volume for visualization

3.2 The Master Header

The basic layout of the master header is shown in Table 1. As most file formats do today, the UVF begins with so called *magic characters* to make file identification easy regardless of the extension; still we recommend the file have the extension ".UVF". This "magic" is followed by a Boolean variable indicating whether the rest of the file is in little or big endian. Since anything before this flag is 8-bit based (the chars as well as the Boolean itself) it will look the same regardless of the endianess of the system and thus can be read in the same way. To avoid permanent conversion during runtime for the rest of the file, if the endianess does not match the target system we propose to do a one-time in place conversion of the file, however this operation is optional. After the endianess flag a 64-bit version number of the format is stored, whereas the current proposal corresponds to the version number one. Although, we have designed the format to be entirely downward compatible, we use this counter as a concession to the fact that we may have overlooked something now or that some oddities may be introduced in future versions and to make sure a reader can address these issues. Next, up to three data fields store an optional checksum. Our current implementation supports CRC32 and MD5 hashes to guarantee the correctness of the file. From these fields **ulChecksumSemanticsEntry** is of particular interest as it uses the method of *semantics tables*, which is a concept that we use throughout the file. A semantics table is a table that is hardcoded into the reader and maps a 64-bit unsigned integer to a particular semantic. While these tables may grow in future versions, no value is ever to be removed from the table. Our format ensures that if the semantic table of a (outdated) reader does not have corresponding entry it can still continue to read the file. The semantics table for **ulChecksumSemanticsEntry** is given in Table 2.

The final value in the global header specifies the offset to the first data block in bytes. Since this value itself is a 64-bit unsigned int, a value of 8 indicates that the first data block starts right behind this offset value. The space in between the offset value and the first data block is reserved for future additions, whereas any addition in turn should be structured in a similar way to end with an offset counter to allow for even further additions, while not breaking the general concept of downward compatibility.

Table 1. The layout of the master header

Master Header				
FREQUENCY	TYPE	NAME	SEMANTIC	EXAMPLE
8	CHAR8	**wstrMagic**	8 magic chars to quickly recognize the file UVF-DATA	n/a
1	BOOL8	**bIsBigEndian**	denotes if the file is stored in big endian order	false
1	UINT64	**ulFileVersion**	the version of the file format	1
1	UINT64	**ulChecksumSemanticsEntry**	the checksum algorithm used to test the file	1 (CRC32)
1	UINT64	ulChecksumLength	the length in byte of the checksum, if **ulChecksumSemanticsEntry** is set to "0" (none) this field does not exist	32
ulChecksumLength	BYTE	**byChecksum**	the checksum: -the checksum computation starts right after this field - if **ulChecksumSemanticsEntry** is set to "0" (none) this field does not exist	0x3f52ac48
1	UINT64	**ulOffsetToFirstDataBlock**	offset to first Data Block in bytes	0 (no more data in header)
space for future extensions				

Table 2. The Checksum Semantics Table

Checksum Semantics Table	
ID	Semantic
0	None
1	CRC32
2	MD5

3.3 The Data blocks

All of the remaining data blocks are divided into a header containing the metadata and a raw data block. In the following we take a closer look at the header, which has a structure similar to the global header.

3.3.1 Block Header

The basic structure of the block header is depicted in Table 3. Its structure is similar to the global header with one extension block. It has a very short section that identifies the data block with a string for display to the user, a block semantic that identifies the data to the system (the block semantics table is shown in Table 5c), a compression scheme also stored as an entry into a semantics table, and finally a pointer to the next data block, if this value is set to zero then this is the last block in the file. Again this allows future extensions to be skipped by older readers. If a reader does not know the semantic of the block or does not support a compression scheme, it can simply skip the data block while the string allows the reader to notify the user of the contents of the skipped block.

The main part of the block header contains information about the volume data such as the dimension of the domain and the semantics of each of these dimensions, stored as an index into the Domain Semantics Table (see Table 5b). Next, a general affine transformation is stored to scale, move, rotate etc the volume. This information is followed by a vector, which specifies the size of the volume in voxels, and a vector that specifies the maximum size of a brick. Bricking is an integral part of the file format in that it allows bricks to be loaded from disk via a simple memcopy operation. Next, the brick overlap is stored followed by a set of parameters describing the LOD structure of the data. Again, to our best knowledge no other general volume format stores the LOD levels together with the dataset. Again, this allows for the memory manager of the application to fetch data using simple memcopy operations. This ability is particularly important for out-of-core rendering applications that need a low resolution, bricked representation, and would otherwise have to recreate this representation every time the dataset is loaded.

Next, the block header holds the information describing the actual data stored at each voxel. First the dimension, then the semantics (see Table 5a), and finally the data elements themselves are described. We allow for arbitrary data elements by defining the bit size of the mantissa and the signed bit, thus allowing

any user defined data format to be handled. The block ends with the obligatory offset to data pointer that – in contrast to the ulOffsetToNextDataBlock pointer - points to the data described by this header.

Table 3. The Block Header Layout

Block Header				
FREQUENCY	TYPE	NAME	SEMANTIC	EXAMPLE
1	UINT64	**ulStringLengthBlockID**	Length of the String that describes the content of this data block in a human readable form	85
ulStringLengthBlockID	CHAR8	**strBlockID**	A String that describes the content of this data block in a human readable format	Sequence of 1000 CT Scans of a pig heart each at 512^3 one image taken every tenth of a second
1	UINT64	**ulBlockSemantics**	index into "Block Semantics Table" describing the data block (all tables see next slide)	1 (volume data)
1	UINT64	**ulCompressionScheme**	index into the "Compression Algorithm Table"	0 (none)
1	UINT64	**ulOffsetToNextDataBlock**	offset to the NEXT Data Block in bytes, **zero if this is the last block**	size of data + size of reaming header
1	UINT64	**ulDomainDimension**	the dimension of the data domain	4
ulDomainDimension	UINT64	**ulDomainSemantics**	indices into "Domain Semantics Table" describing each domain direction	[0,1,2,3] (X,Y,Z,time)
(ulDomainDimension+1) * (ulDomainDimension+1)	DOUBLE	**ulDomainTransformation**	affine transformation applied to the domain (matrix is applied to all dimension that includes time etc.) to apply the transformation add one additional "1" coordinate to the vector and "dehomogenize" after the matrix multiplication	[0.01,0,0,0,0] [0,0.01,0,0,0] [0,0,0.01,0,0] [0,0,0,0.1,0] [0,0,0,0,1] (this example simply rescales space to "mm" and time to the "tenth of a second")
ulDomainDimension	UINT64	**ulDomainSize**	size of the dataset in each dimension	[512,512,512,1000]
ulDomainDimension	UINT64	**ulBrickSize**	the size of the bricks	[64,64,64,1]
ulDomainDimension	UINT64	**ulBrickOverlap**	overlap of the bricks	[4,4,6,1]
ulDomainDimension	UINT64	**ulLODDecFactor**	describes the factor each LOD level is scaled down compared to the previous one, a zero denotes no LOD for this component	[2,2,2,1] (MIP-map like LOD for the volume in space, no LOD in time)
ulDomainDimension	UINT64	**ulLODGroups**	indices to denote dependent LOD dimensions starting form zero to "*ulLODIndexCount*" the value for dimensions without LOD is ignored	[0,0,0,*] (x,y,z, dimension are always down sampled together) the implicit value *ulLODIndexCount* is thus 1 in this example
ulLODIndexCount	UINT64	**ulLODLevelCount**	how many levels of LOD are present, for each LOD index	[2]
1	UINT64	**ulElementDimension**	dimension of element (scalars & vectors = 1, tensors > 1)	1
ulElementDimension	UINT64	**ulElementDimensionSize**	number of components for each data element	1
ulElementDimensionSize* ulElementDimension	UINT64	**ulElementSemantic**	indices into "Element Semantics Table"	[1] (CT)
ulElementDimensionSize* ulElementDimension	UINT64	**ulElementBitSize**	bit sizes for the components of the data element	[16]
ulElementDimensionSize* ulElementDimension	UINT64	**ulElementMantissa**	bit sizes for the components of the data element	[16]
ulElementDimensionSize* ulElementDimension	BOOL8	**bSignedElement**	flags signed components the data element	[false] (with the above information this denotes an unsigned short)
1	UINT64	**ulOffsetToDataBlock**	offset to THIS Data Block in bytes	0 (no more data in header)
space for future extensions				

Table 4. The Compression Semantics Table

Compression Semantics	
ID	Semantic
0	none
1	Volume Block Compression
10000	JPEG
10001	PNG

Tables 5: a) The Elements Semantics Table, b) The Domain Semantics Table, c) Block Semantics Table

a)

Element Semantics Table	
ID	Semantic
0	Undefined
1	General Vector Value
2	General Tensor Value
3	Symmetric Tensor Value
10000	Red
10001	Green
10002	Blue
10004	Alpha (Opacity)
20000	MR
20001	CT
30000	Time (Second)
30001	Mass (Kilogram)
30002	Electric Current (Ampere)
30003	Thermodynamic temperature (Kelvin)
30004	Amount of substance (Mole)
30005	Luminous Intensity (Candela)

b)

Domain Semantics		
ID	Semantic	Unit
0	X	Meter
1	Y	Meter
2	Z	Meter
3	Time	Seconds

c)

Block Semantics	
ID	Semantic
0	Empty
1	N-Dimensional Transfer function
2	Preview Image
3	Regular N-Dimensional Grid
4	8bit String Key/Value Pairs

3.3.1 Data

The data that follows the block header is stored in simple binary form. The following pseudo code explains the ordering of structured volumetric data:

for each ***ulLODIndexCount***
  for each Brick (inverse of ulDomainSemantics)
    for each ulDomainDimension (inverse of ulDomainSemantics)
      for each ulElementDimension (inverse of ulDomainSemantics)

This ordering is the same as in practically any other known volume format. It again is chosen such that all data can be moved via memcopy operations from disk to RAM, or directly to video RAM if a GPU is used to render the images. Data types are also stored in their canonical memcopy form, such as vertex lists, index lists and Key Value pairs. The exact specification for any of these data formats would fill too much space for this publication but can be downloaded from our website at ftp://datex.sci.utah.edu/UVF.

4. CONVERTERS

Along with the reference implementation of the UVF reader that can be downloaded from our website at ftp://datex.sci.utah.edu/UVF, we provide a set of converters to ease the burden of moving to the new UVF format. So far we can cnovert DICOM, DAT/RAW pairs, NRRD, PNG and JPEG stacks into UVF. The code

is released under a very liberal form of the BSD license to faciliate easy adoption even in comercial environments (see website for details).

5. CONCLUSION AND FUTURE WORK

In this paper we have presented a general format for storage of practically any kind of volumetric datasets. We focused on simplicity of the format and efficiency for the reader such that practically no post-processing on the mem-mapped data is needed for rendering. Within our department this data format has been successfully integrated into many different visualization systems, simplifying the means of data exchange with little implementation overhead. We hope that this publication and the software we are providing can bring a similar experience to other groups.

Currently we are investigating the integration of other general data into our file format, as well as provenance, history and undo information seem. In particular provenance becomes ever more important with the massive amount of datasets. Here we are looking into simple general description techniques for a provenance description that we could integrate into UVF such as the Open Provenance model proposed by Moreau et al. [2007]. We are also looking forward to feedback of other researchers suggesting data that we have not thought about yet.

ACKNOWLEDGEMENTS

The authors would like to thank the ClearView and ImageVis3D development teams for their thoughtful comments on the UVF format as well as the anonymous reviewers for bringing other established file formats to our attention.

REFERENCES

Gordon Kindlmann, NRRD, 1998, Nearly Raw Raster Data http://teem.sourceforge.net/nrrd/format.html

Gordon Kindlmann, Teem, 1998 http://teem.sourceforge.net

Kitware, 1999, The Visualization Toolkit File Formats, The VTK User's Guide
http://public.kitware.com/VTK/pdf/file-formats.pdf

Moreau et al. 2007, The Open Provenance Model
Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J. and Paulson, P
Technical Report 14979, University of Southampton

NEMA, 1992-2008, *Digital Imaging and Communications in Medicine (DICOM)*, http://medical.nema.org/

The HDF Group, 1988-2008, The HDF5 Format, http://hdf.ncsa.uiuc.edu/HDF5/