Editors: Claudio T. Silva, csilva@cs.utah.edu
Joel E. Tohline, tohline@rouge.phys.lsu.edu

ON SAFARI IN THE FILE FORMAT JUNGLE—WHY CAN'T YOU VISUALIZE MY DATA?

By Werner Benger

Lots of glossy images and striking movies on the one side, lots of numbers and full hard disks on the other—seems like a natural pair. So why can't visualization people "just visualize" some data?

A typical visualization request goes something like this:

> Here, I have some data for you to visualize. The data format is very simple—it's just ASCII—so you can easily read it. And, there's plenty of time left: I don't need it until my presentation next week.

To the utmost disappointment of the interested party, this approach usually doesn't work.

The apparent expectation is that visualization is something like using a Web browser: here's the URL, just click on it and out pops a pretty image. Reality, however, differs markedly from this vision. When people see a good visualization or movie of some data, they typically have no idea about the effort behind it. For example, I once showed a scientist a 30-second animation of his data, and he asked how long it took to create it. I said it was pretty quick—only three weeks. He was totally puzzled by that answer, and expressed his astonishment: "For 30 seconds? Why did it take you so long?"

At the time, I was unprepared for this reaction and somewhat at a loss for words. Fortunately, in the meantime, I've heard rumors that at Industrial Light & Magic, a team of several hundred people worked for three months to create a three-second sequence for a *Star Wars* movie. So, now I have a better reply in similar situations: I point out that I'm still orders of magnitude faster than the George Lucas team. Of course, my result is not yet a Hollywood blockbuster.

Figures 1 and 2 show examples of visualizations I created from "easy to read" data, including the famous "just ASCII" format. These images have become quite popular, but the amount of work involved in preparing the data for the final visualization process—which takes only a fraction of a second—is far from obvious in the final results.

Getting Stuck in the File Format Swamp

So, what's the main hurdle that slows the process of "just visualizing" some data? First and foremost, there's the issue of file format and reading the data. Actually, this allegedly simple issue might well consume 90 percent or more of the data visualization time. In some cases, it's taken us several months to become able to read a particular data set. Only then could we finally press the "visualize and go" button.

"Just ASCII"

The "just ASCII" data approach isn't as simple as it appears to the data provider, who often assumes that a "human readable" file is also easily readable by a computer program. However, writing data is much easier than reading it.

Reading ASCII data requires some parsing routines, which are quite sensitive to minor changes in the input format. In one case, we'd finally implemented some data-reading routines, and then discovered that some newly provided data was unreadable. It took us quite some time to discover that the new data simply had one additional empty line in its format. Clearly, this was a triviality for the data provider, who didn't even think to tell us about the change. However, it was a big deal for the reading routines, which stumbled across this change in the most inconvenient moment.

It's mostly those simple things that, added together, are extremely time consuming when dealing with "just ASCII." Besides, parsing ASCII text data is extremely slow and requires lots of disk space. So what's the alternative? Just use binary data formats.

"Just Binary"

Because a binary file by itself is simply an agglomeration of bits and bytes, it's incomplete without an appropriate description of how to read it. So, the actual file format consists of two files: the raw data file and an email describing how to read it from the human who created it. The creator might also provide this information—the "metadata" describing the data—by other

 1521-9615/09/$26.00 © 2009 IEEE

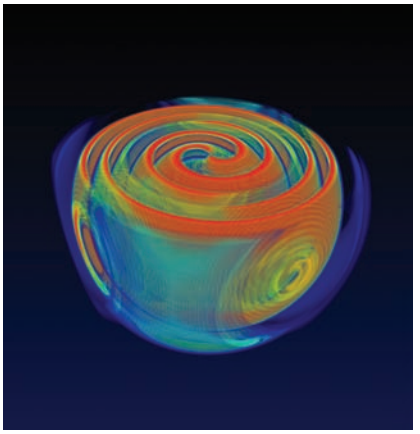

Figure 1. The outspiralling gravitational field resulting from the collision of two merging black holes, visualized as a 3D colored volume. I received the data on which this visualization is based in "just ASCII" format—a sequence of complex coefficients for spherical harmonic base functions. Preparing for this visualization required about 18 months of implementation effort.

means, such as a document or some source code.

In general, reading binary data is easier than reading ASCII because there's no parsing step. There's also less flexibility in the file variations: either the reading works or it fails—more or less—completely (but at least there's no fiddling with arbitrarily introduced empty lines). However, you might still have to deal repeatedly with supposedly solved problems, such as byte orderings (little-endian vs. big-endian) or different data types (int/float/double). Also, operating on "just binary" data is low level and requires the tedious reproduction of work that various I/O libraries already do for some established file formats.

Established File Formats

Many different file formats exist for data to be used for scientific visualization. These formats come with more or less adequate libraries and APIs, giving developers the opportunity to select their favorites. As it turns out, this freedom of choice is also exactly the problem: as of now, there's no generally accepted or standardized format that can deal with all cases. So, one specific data type can be written in multiple ways.

Typically, the file format is based on what the application developer plans to do with the data after processing it, such as targeting visualization using a specific software tool. For this specific scenario, visualizing the data is indeed close to a click-and-go solution. However, for software that doesn't understand this particular file format—but might provide some essential visualization feature—this format counts as "just binary," and making it cognizant of the actual format might require a fairly significant implementation effort.

If the chosen format is accompanied by a software library and API, you're in luck. But many times, you have to construct it from scratch, based on some PDF documentation or an email description. And, even with a software API, it might require a major programming effort as the library might be available only in Java or Fortran, whereas the target visualization software is written in C++ or C (or vice versa). Yet, even if the API is available in C/C++, it might be so complex to use that supporting it would be considerably time consuming.

The File Format Gap

Application developers who write scientific simulation software often choose to output data in their own, self-invented format instead of using an existing format. In so doing, they avoid an API's steep learning curve and can concentrate on solving physics problems rather than spending time on software engineering issues. Likewise, visualization software developers probably prefer to spend time developing visualization algorithms instead of supporting dozens of different file formats. After all, in an academic research environment, no one gets papers written for supporting another file format! The physicist writes papers for solving problems in physics, and the visualization researcher writes papers for new visualization algorithms—not for new file format support.

Nevertheless, understanding each other's data is a necessity, and bridging the gap between research interests and

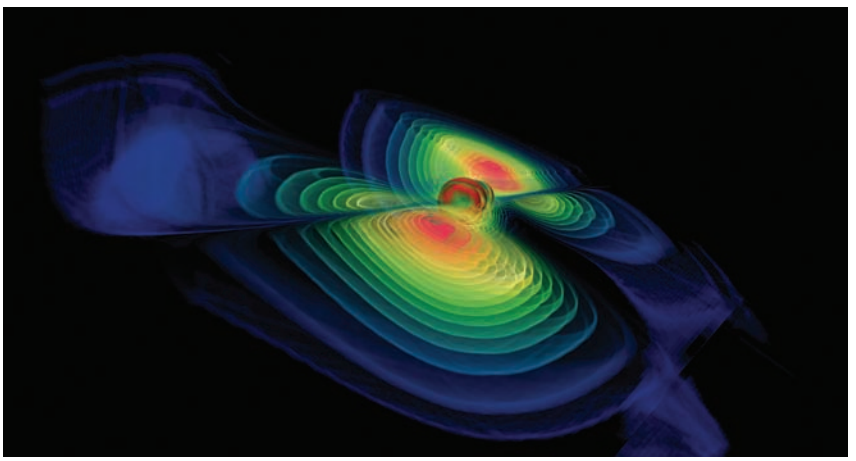

Figure 2. Gravitational waves from colliding black holes, with the apparent horizons of the merging black holes in the center. I received the apparent horizons as a table of multipole coefficients in "just ASCII" format—simple to read, yet the surfaces' 3D reconstruction required an implementation effort of about six months.

practical needs is an unavoidable requirement. To do this, both sides must realize that no one is enthusiastically interested in devoting time, effort, and resources toward supporting another file format. Application scientists often expect the visualization side to do all the data reading efforts and frequently state that, "I won't use your visualization application if it can't read my data." A valid and reasonable statement at the first sight, but again, the Web browser comparison comes to mind: visualization of datasets is expected to be as easy as browsing a Web site. This analogy does apply, but with an essential difference: Web site creators provide images in a format that the Web browser understands, be it JPEG or GIF or PNG. Web content creators wouldn't expect a Web browser to support their own homegrown file formats and wouldn't refuse to examine their own data until such support was offered by the browser developer.

Self-invented file formats frequently entail highly frustrating changes that aren't announced to the visualization side.

In the scientific visualization context, the application scientist is somewhat in the content creator's position, while the visualization person is in the position of writing the "data browser." The main difference is that well-established standards exist for images, but not for 3D data or for complex meshes or even for such simple cases as scalar fields on uniform grids. Actually, this context offers a particularly absurd situation: more complex datasets might have fewer file format issues because, for complex data types, only one file format might exist. In contrast, for a simple data type, there might be many competing format options.

Clearing a Path to a Common Data Model

Overcoming the gap between application scientists and visualization developers when it comes to file formats requires mutual investment from both sides. In a fruitful cooperation, application scientists provide the appropriate knowledge and human resources to make their data "readable." In a less fruitful collaboration, they provide data files with improper documentation on how to read them or reading routines that are specific to a software environment and not applicable to the visualization environment (such as using commercial libraries or complex functionality not available to the visualization). Surprisingly, the application scientists often don't even know their own data formats. They treat their simulation tools as black boxes: some else has written the code, so they can't answer questions about how to read the data files that they themselves provide. And, as mentioned earlier, self-invented file formats frequently entail silently introduced and highly frustrating changes that aren't announced to the visualization side. Such issues would be another reason to use established file formats for communicating data—if such formats existed.

Bridging the File Format Gap

A big hurdle for establishing a standard file format for scientific data is the incompatibility of data structures themselves. Not all file formats can cover all data types that occur in scientific simulations and visualization. A standard file format that enables true interoperability across independently developed applications would need to cover all different cases within the same model. At this point, philosophical beliefs come into play: many people simply don't think this is possible at all. Rather, the mainstream approach appears to be "special problems need special solutions," which results in myriad specialized file formats and a list of special cases to be handled by some visualization application. Such beliefs are hard to overcome.

Many application developers focus on only their own application (which, of course, produces the papers) and seldom see things in a larger context. Nevertheless, there are a few attempts at establishing a common data model. One of the first common data model proponents was David Butler, who said, "a common language for the interchange of scientific data exists—this is the language of mathematics, which is common to all simulations—we just have to use it."[1] He then proposed to model data using mathematical concepts based on the theory of fiber bundles. These ideas have been successfully implemented in the IBM Data Explorer, which is available as the OpenDX open source tool.[2] OpenDX comes with its own file format, which is powerful enough to cover a wide range of data types.

OpenDX users are generally quite pleased with its flexibility. However, it's also complicated to use and its development has become dormant. So, despite a powerful internal data model, OpenDX doesn't provide state-of-the-art rendering and visualization techniques. On the other hand, the fiber

bundle data model—which can provide generic algorithms[3]—is mostly unknown to modern visualization software tools, which usually provide highly specialized visualization algorithms for specific data structures. The OpenDX file format in its generality isn't supported elsewhere. This means that support for a specific visualization application's specific file format is done on a tedious, case-by-case basis.

HDF5

Managed by the nonprofit HDF group, HDF5 (www.hdfgroup.org/HDF5) is a promising file format that was originally developed at the US National Center for Supercomputing Applications. HDF5

- provides many unique features,
- was designed for high performance,
- has a large user community,
- is in active development with long-term support plans,
- comes with a well-documented API, and, most importantly,
- provides a self-describing file format.

HDF5 is receiving increasing support from many applications. Data provided in HDF5 format solves the big problem of how to read data. Using the HDF5 tools, developers can easily determine the files' contents. The HDF5 library API provides neat, well-documented routines to perform these operations.

However, providing data in HDF5 doesn't solve the problem of being able to retrieve the data completely. It only shifts this problem to a higher level, from the syntax (how to read the data) to the semantics (how to interpret the data). That is, being able to read and interpret one HDF5 file doesn't mean you can interpret other arbitrary HDF5 files as well; it's not a data model per se, but rather a multidimensional arrays container (that also provides other highly useful features).

HDF5 is best compared to a container file format, such as an AVI movie file, which acts as an envelope for movies. But how the movie's image frames are actually stored depends on the video codec that's used. If such a codec isn't available when reading the AVI file, it's unreadable, even though it's an AVI file. The same is true with HDF5: the file format is flexible enough that you can write a specific data type in many different layouts. However, if the writing application and the reading application don't agree on how to lay out the data, the data are still unreadable, even though they're stored in HDF5 format.

The simpler a data type is, the more possibilities exist for different layouts. For example, for a 3D scalar field on a uniform grid, it might sound straightforward simply to store it as a 3D dataset in HDF5. However, a difficulty arises with the associated metadata, such as coordinate information, or the physical time associated with a time series. You can store such information via attributes, but doing so requires agreement on the attribute names. Introducing conventions is always a bottleneck in achieving interoperability. As it turns out, the fewer the required conventions, the less error-prone the implementations will be. Ideally, a file format should be keyword free—that is, all its semantic information should be contained in a logical layout that leads everyone in the same direction, without need for negotiating naming schemes for keywords and their meanings. Indeed, I remember weeklong discussions with colleagues on whether some attribute should be called "Size" or "size"; everyone had a strong preference for one or the other. Unfortunately, such decisions, which are based simply on matters of taste, can make files unreadable if the outcomes don't agree or they change over time. Therefore, it's best to avoid naming conventions entirely if possible. I tackled this challenge with fiber bundle HDF5 (F5; see www.fiberbundle.net), which I've been developing for my own visualization environment.

The F5 File Format

F5 arose out of the need to support many different data types and their properties in the numerical relativity context. Many quantities that are implicit in Euclidian geometry—such as coordinate systems and metric tensor fields—are handled explicitly in general relativity. Given this, a file format that can support this general case is likely to also support many other cases.

Six-Level Hierarchy

Inspired by Butler's initial (but abstract) considerations on using fiber bundles to model data and its successful implementation in OpenDX, F5 uses HDF5 to lay out data in a hierarchical structure with six levels, each with a specific semantic meaning:

1. Time slice
2. Grid name
3. Topological skeleton
4. Coordinate system
5. Field name
6. Field fragment (optional)

Only the fifth level and above contain actual datasets. Rather than using keywords, the metadata information is expressed by the placement of entries in the hierarchy. The topmost level defines a time series. The second level contains an arbitrary, user-chosen

name for the "Grid" geometric entity. The third level provides entries for the Grid's topological properties, such as its vertices, cells, or edges. The coordinate system level refers to data given in Cartesian coordinates, polar coordinates, and so on. The fifth level is for fields and consists of multidimensional datasets. In the sixth, optional level, these data sets can be fragmented into smaller parts, which are useful, for example, to handle output from a parallelized simulation code where each compute node deals only with a field subset for the global computational domain.

The data layout is largely keyword independent because the entry names are irrelevant. The only agreement is to store data on a six-level hierarchy and associate semantics with each level. In practice, the layout isn't completely keyword free, however, because a minimal attributes set still must be attached to some objects. Nevertheless, the intent is to keep the set of such "reserved words" as small as possible. We can therefore view the F5 model as an (intentionally) keyword-free version of the OpenDX model that groups "compatible" arrays together.

API Complexity

A major reason why previous attempts at establishing a common data model have failed is that their APIs exposed the data model's full complexity. Clearly, if you need to store only one simple data type, you'd prefer to avoid the considerable overhead that accompanies the entire generality of a common data model.

A complex, powerful, generic API is more intimidating than encouraging. The F5 model therefore comes with a library that is built upon HDF5 and provides a lightweight API for writing common data types in a simple way, such as with a single API call. Still, the API allows access to the deeper functions, so, for more complex—and currently unsupported—data types, you can directly use the underlying HDF5 functions to write the data in a layout compatible with the fiber bundle concept.

Supporting only this F5 format in the visualization application is much easier than supporting a dozen file formats. You therefore have to convert specific file formats into the F5 format before you can visualize it. You can do this in collaboration with the data developer; the HDF5 library itself is well documented and the F5 library simple enough that other people can quickly learn how to use it (in contrast to a complex visualization application).

Adding new file format support to a visualization application isn't as simple as writing "a reader"—that concept actually works only for simple cases where you can keep the entire data set in memory. Nowadays, however, with datasets much larger than the RAM available on local workstations or even visualization clusters, reading all the data at once isn't really an option. The objective instead is to read only those data that a specific visualization operation requires. For example, a data subset or the metadata alone are often sufficient, such as when you're displaying the bounding box of some geometry. HDF5 provides a clear separation of data and metadata, and provides mechanisms to retrieve only subsets for data arrays (hyperslabs). A visualization application that uses such features couldn't easily support another file format that doesn't provide similar functionality.

To visualize someone's data, you must first be able to read them. As with many things, such an allegedly simple task can evolve into a major time-consuming effort. Overnight "just do it" approaches are typically unrealistic and lead to mutual frustration. A successful visualization project requires investment from both parties—the data provider and the visualizer—and keeps in mind everyone's research priorities. No one wants to invest the time needed to implement file converters at the expense of research hours.

Reading data entails practical challenges, and the F5 format is an attempt to tackle them. Although F5 doesn't claim to be the ultimate solution, my hope is that its spirit and design philosophy will find support in similar attempts.

References

1. D.M. Butler and S. Bryson, "Vector Bundle Classes from Powerful Tool for Scientific Visualization," *Computers in Physics,* vol. 6, no. 6, 1992, pp. 576–584.
2. L.A. Treinish, "IBM DX Data Explorer Data Model," IBM Research, 1997; www.research.ibm.com/people/l/lloydt/dm/dx/dx_dm.htm.
3. W. Benger, "Colliding Galaxies, Rotating Neutron Stars and Black Holes—Visualizing High Dimensional Data Sets on Arbitrary Meshes," *New J. Physics,* vol. 10, 2008; http://stacks.iop.org/1367-2630/10/125004.

Werner Benger is a visualization researcher at the Center for Computation & Technology at Louisiana State University. Before joining CCT, he worked at the Zuse-Institute Berlin to develop the Amira (now Avizo) visualization software in collaboration with the Max Planck Institute for Gravitational Physics (Albert Einstein Institute) in Potsdam, Germany. His research interests include visualization of astrophysical phenomena, focusing on tensor fields. Benger has an MS in astronomy from the University of Innsbruck, Austria, and PhD in mathematics and computer science from the Free University Berlin. Contact him at werner@cct.lsu.edu.