# Beyond the Visualization Pipeline: The Visualization Cascade

Werner Benger[1] and Georg Ritter[2] and
Marcel Ritter[1,3] and Wolfram Schoor[3]
[1]Center for Computation and Technology, Louisiana State University, USA
[2]Institute for Astro- and Particle Physics, University of Innsbruck
[3]Institute of Computer Science, University of Innsbruck, Austria
[4]Fraunhofer Institute for Factory Operation and Automation
werner@cct.lsu.edu, georg.ritter@uibk.ac.at,
marcel@cct.lsu.edu, Wolfram.Schoor@iff.fraunhofer.de

**Abstract**

The concept of a pipeline has become a quite common way of thinking about the process of visualizing data. In this article we discuss the inherent limits of this concept and argue for the need to expand this concept for achieving higher performance and convenience to the end user. While the traditional model of a visualization pipeline describes the execution of some data flow, it is most suitable for a static data-set. However for time-dependent data (e.g.) we intend visualizations to be as fast in time as they are in space. The pipeline model does not recognize similarity and repetition of operations which is essential to achieve the desired performance. The pipeline model thus needs to be extended to efficiently cover multiple traversals and caching of intermediate results, which we call the *Visualization Cascade*. It will be demonstrated in practice within its implementation in the VISH visualization environment.

## 1 Introduction

The process of visualizing data begins with the source data containing the information to be visualized and ends, finally, with a derived image representing the data. To arrive at the image the data needs processing, like being searched or filtered, depending on the nature of the data and the analysis requirements. It then must be mapped to graphical entities that are subsequently rendered into an image. In [Haber & McNabb, 1990] the authors identify and refine the general operations data undergoes in the process of creating visualization. The data flows in a pipeline through a chain of stages, as depicted in Figure 1, and finally, results in a representing image. The pipelined model they present, is known as the Haber-McNabb model of the visualization pipeline and

has been a widely successful concept for the design of visualization software. Several well known software packages have been built upon this idea, examples include AVS [Upson et al., 1989], VTK [Schroeder et al., 1997], IRIS Explorer [Foulser, 1995], OpenDX [Treinish, 1997] (for a more complete list see [A.A. Ahmed, 2007]).

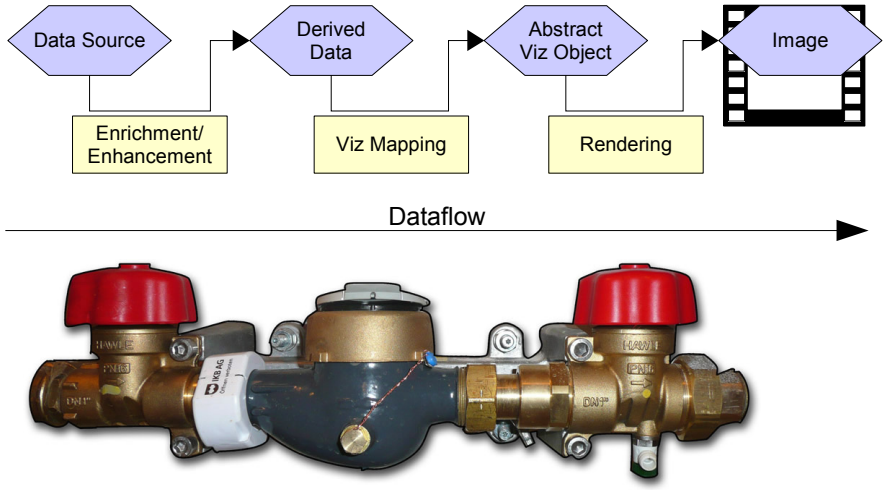


Figure 1: Visualization Pipeline: Data flows through the pipeline while operators modify the stream. They extract, filter, map and render data. Finally the pipeline outputs is an image or image stream.

While trying to understand the data through exploratory visualization, as outlined in [Upson et al., 1989, Card et al., 1999], a flexible and easy to use mechanism to enter the operation on the data into the software is needed. Exploring and trying to understand as much as possible from the data by inspecting them, includes making frequent changes to parameters of the visualization, sometimes even to the parameters or models used to create the data. Being able to easily change and adapt the parameters and, in addition, derive the visualization results fast, is of great importance as it helps to decrease the time needed for one investigative cycle. For some data, features might only become visible if it is possible to navigate interactively inside the parameter space.

The pipeline concept has been found to be well suited and intuitive to understand when used to represent the flow of data in a user interface. The user can graphically construct a visualization pipeline by interconnecting different stages through attaching a pipeline to nodes. An early example of such a graphically programming interface has been implemented in ConMan [Haeberli, 1988], more modern examples include the Spiegel framework [Bischof et al., 2006], or the graphical user interface of LabVIEW [Johnson & Jennings, 2001] in which a data stream, mainly originating from measurements, can be connected to processing nodes or the gstreamer framework [Black et al., 2002] in which multi-

media data is handled this way.

Once the structure of the pipeline has been set up, it needs to be executed. Different schemes have been implemented to derive the final image. When using implicit execution, as applied in VTK [Schroeder et al., 1997], data is time stamped and only "upstream" nodes are executed, if demanded. Explicit execution, as used in the IRIS Explorer [Foulser, 1995], relies on external management of the data processing nodes to decide which needs re-computation.

The flow of data is initiated in two principal ways. In the "pull" case, a downstream receiver requests the data from an "upstream" node. In a "push" case, the "upstream" node would forward the data to the next stage in the pipeline as soon as it is available. Also a mixed version can be implemented, as they are independent.

If we want achieve full interactivity in the visualization of large data-sets we find that the current design of pipelines and their execution models are not well suited to meet the requirements. Response times to changes of parameters are not fast enough, as data travels too slowly through the whole pipeline to reach an interactive frame rate, as described in [Shen, 2006] for the case of time dependent data.

Not only for a change in the time parameter, but for any change made to a parameter of the visualization, the whole visualization pipeline has to be executed for every single frame. As this often exceeds the time required for an interactive frame rate, a common solution is to apply off-line rendering. Frames of an animation sequence are rendered for later viewing, but interactively exploring the data would be more desirable and would also possibly increase the chance of gaining further understanding of for example spatial-temporal features of the data.

Working toward the visualization challenges one ("Make the spatial and temporal resolution of visual displays indistinguishable from physical reality.") and four ("Optimize physical resources used to perform visual interactions."), as described in [Hibbard, 1999] and incorporating the user wishes for interactivity, here we present an extended pipeline model that utilizes the structure of modern graphics hardware to minimize the time needed to derive the final image. We propose that, by introducing a flexible caching mechanism, it is possible to increase re-usage of already processed data, especially in between different pipelines constructed for different parameter sets. In combination with a data storage model and utilizing GPU memory, full interactivity on a data-set of the size of 17 GB, containing 1.6 million points in 200 time steps [Benger, 2008], has been achieved.

## 2 The Visualization Cascade

The major drawback of the concept of the visualization pipeline is that it does not talk about caching of results. If an operation similar to an earlier is to be repeated, we would not want to have the entire pipeline to be traversed again. Only those sections that have changed shall be re-computed. A typical usage

scenario is running an animation of time-dependent data. The "conventional way" is to load each time step, pump it through the visualization pipeline to create a pixel frame for each time step, and then eventually watch the evolution of the data as a movie. Many features of a dynamic data-set are only appreciated when viewed at interactive speeds of e.g. 30 frames per second, but usually the traversal of the entire visualization pipeline is much slower than 1/30th of second.

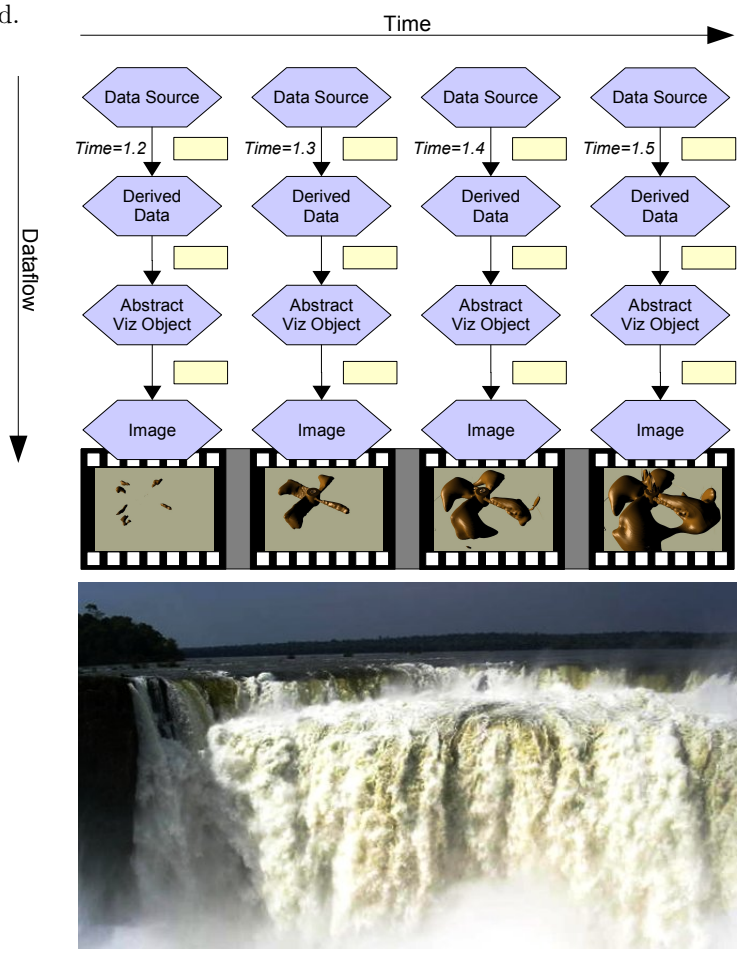


Figure 2: Exploring a full parameter space of some data-set requires parallel traversal of the data flow - resulting in a cascade rather than a single pipeline traversed repeatedly.

While speeding up the initial traversal of the pipeline might just be impossible, results of previously computed operations can be cached up to available RAM. We may consider the rendering of an animation as the execution of a sequence of multiple parallel visualization pipelines. The execution nodes of each

pipeline reside on the same level. At each such node we might need to cache some result of a previous operation. We may think of such a system as a cascade of data flowing down a water fall, in many parallel ways and intermediate levels where data reside to be cached.

Furthermore, data may eventually flow from one stream to another one, i.e. the "visualization pipeline" from one time frame may employ parts of a visualization pipeline from another time frame as well. Such may be the case when fusing data-sets given at different time intervals, for instance a data-set that is sampled at T=0.0, T=10.0 and another one at T=0.0, 1.0, 2.0, etc. If interpolation in time is not requested but rather the "most recent" timestep should be displayed, then at T=1.0 the coarse data-set at T=0.0 would be used, which does not require traversal of the viz pipeline for the coarse data-set all they way up to its source. A new computation will only be required when both data streams from the two pipelines will merge.

We may consider "time" as a parameter that is orthogonal to the flow of the visualization pipeline. It rather parameterizes the visualization pipeline (a linear, one-dimensional data flow) and unfolds it into multiple instances, thereby creating the visualization cascade (a two-dimensional flow of data). At each cascade level, there will be the essential decision when to re-execute the computation or to re-use existing data. This depends on additional parameters that have been changed since the last traversal. If the data at each level depend on "time" only, then there is no need for re-computation at all once data exist there already. However, there may be other parameters as well, such as depending on user interaction. For instance, when inspecting some time-dependent 3D data volume, the user-defined threshold level of an isosurface, or the range and color values of a colormap during volume rendering. In both cases, there is no need to reload data from disk when repeating an animation over a previous time range. However, in the case of the isosurface display, the computation of the geometry has to be re-executed. In the case of the dynamic volume rendering, there is not even a need to reload data on the graphics card, but only some texture maps might need to be updated when changing the colors.

While some parameters in the visualization cascade might not require requesting data up from the source, others might. For instance, changing the range of a volume rendering colormap or applying another filter (e.g. a non-linear filter) may require operating directly on the original data, and thus need to reload data and full pipeline traversal. We therefore have to distinguish between two classes of parameters: those that require re-computation, and those that do not. The efficiency of the visualization cascade will depend on proper choices at each node, to avoid unnecessary computation but still perform the essential ones. We will discuss our implementation in the next sections.

## 2.1 Data Result Caching

Each node within a visualization pipeline is an operation on the data stream. In an object-oriented environment, it is usually an object with data structures and member functions. It is straightforward and common use to store computational

data in here. Using this associated node-local space as cache for intermediate results is an option, but not an optimal one.

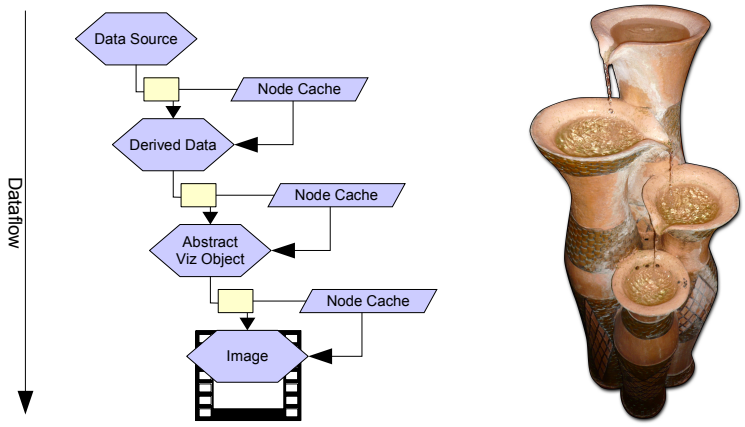


Figure 3: Caching of intermediate results of the data flow through a visualization pipeline allows to avoid repeating previously performed computations. Instead, final results may be retrieved from intermediate levels of the visualization *cascade*.

In our case we utlize Vish [Benger et al., 2007] as visualization environment, which allows using a so-called fiber bundle data model [Benger, 2004, Benger et al., 2006, Benger, 2008]. This model is a framework to handle a wide class of data for scientific visualization within the same structures. It intrinsically supports time dependency and thereby allows to store intermediate results within this data model (a data structure available at the data source) rather then the computational objects themselves.

This approach to store intermediate results directly at the data source (the so-called "Bundle") as extension to the source comes with various benefits:

1. The computational nodes are kept completely procedural; they never store any data itself, and may thus be utilized for any kind of data operation even stemming from different sources. Data are merely seen as parameters to the procedure, but not actually "transported" into the object.

2. Another instance of the same procedure would automatically recognize existing results, as it would store its results in the same location. In the purely object-oriented approach, objects would not know about the existence of other instances.

3. Since the data source is equipped with I/O methods, all intermediate results can be stored on disk and reloaded at a later instance; there is no requirement to equip each computational node with explicit functionality to store its own data.

4. With additional data added to the source, they are available to be inspected with other procedures or visualization modules. This may well lead to unexpected discovery and insight into the data itself, with no additional cost, but in a natural way. Additional data are just available.

Within VISH, the functionality of an *Operator Cache* is provided to attach any kind of data to a data source with minimal requirements. If the data source is a *fiber bundle*, then a more specific method can be applied.

## 2.2 Operator Cache

The "Operator Cache" is a C++ template class used to memorize the result of some computational operation as implemented by a node of a visualization pipeline (the "Operator"). This generic approach only fulfills the first property in the aforementioned list. Hereby the data source has to provide the property to be an "Intercube" object, as described in [Benger et al., 2007]. Basically this is an runtime-version of multiple inheritance, which allows to attach additional objects ("interfaces") to some container (the "intercube" holding many "interfaces"), e.g. an object providing data for visualization.

For instance, if we want to memorize a vector of doubles, then we simply instantiate the OperatorCache template over this data type:

```
typedef OperatorCache<std::vector<double> > OC_t;
```

Now given an InterCube object provided to a computational routine, one may retrieve an OperatorCache object that may be stored there. If not, we would need to create one anyway:

```
void VizNode::compute(InterCube &MyData)
{
        OC_t*OC = OC_t::retrieve( MyData, this );
        if (!OC) OC = new OC_t();
}
```

Note that the retrieve function basically has two parameters, the data object "MyData" and the visualization node. Thus, the operator cache can install a copy of the requested data with each data object and each visualization node. It is a unique place where the node may store data outside its own local memory, as illustrated in Figure 4

The Operator Cache is furthermore related to a set of variable values, a "ValueSet". Its purpose is to associate the OperatorCache with such a set of values. If any of these values has changed upon a repeated call of the viz node's compute function on the *same* data-set then the Operator Cache needs to be equipped with data from a new computation. For instance we might consider an operator that computes some isosurface. If the isolevel value or some maximum number of allowed triangles is changed, then the operator would need to execute the numerical routine again, otherwise it could just return the data stored in the Operator Cache. An "OperatorCache::unchanged()" member function checks if

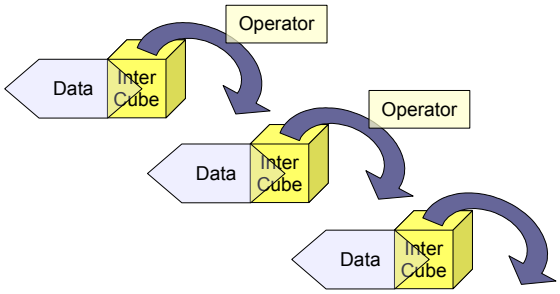


Figure 4: Data Storage in Intercubes

there are any such differences among the values stored with the OperatorCache and the current values had occured (it will automatically return "false" if the OperatorCache was newly created):

```
ValueSet*Changeables = new ValueSet();
        Changeables->addValue(IsoValue);
        Changeables->addValue(NumberOfAllowedTriangles);

        if (OC->unchanged( Changeables ) )
        {
                // do something with the data existing in OC
                return;
        }
        // compute new data and put them into the OC
```

If there had been changes, then following code is supposed to compute new data. There may be other parameters that do not require re-computation, such as another coloring of the resulting isosurface, see Figures 5, 6 and 7. These will be part of the visualization node, but not be added to the ValueSet that is used to inspect the Operator Cache. (The actual source code uses a slightly different syntax as it employs operator overloading to provide a more compact coding.)

## 2.3 Caching in the Fiber Bundle

When data are available in the fiber bundle, and results are storable in the fiber bundle, one would not employ the OperatorCache. Rather, any results will be stored directly in the incoming data structures. To depict how it works, we do not need to know the entire complexity of the full model. It suffices to know that there are objects called `Bundle` and `Grid`. A `Bundle` may be accessed with a floating point value and a string to yield a `Grid` object. Such a `Grid` object may represent a 3D data volume with scalar fields (which is to be identified via some string), or a triangular surface such as the result of an isosurface computation. The actual numerical routine "`Isosurface`" will require a `Grid` object, a field name, and a floating point value specifying the isolevel. The schematic code will look similar to this:

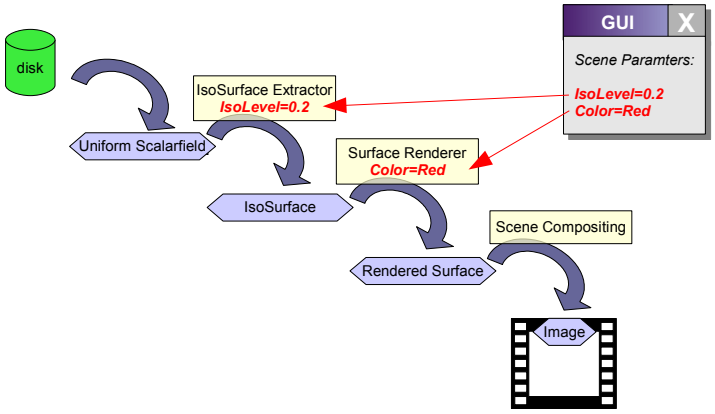


Figure 5: The execution flow: The first time, the complete cascade has to be executed. The operators read the data-set from disk, compute the isosurface, render the surface and composite it to the final image. The separation into operators is hidden and is not all seen by the user.

```
Grid VizNode::compute(Bundle&B, double time,
                      string Gridname, string Fieldname,
                      double Isolevel)
{
      // Construct a unique name for the computational result
string  IsosurfaceName = Gridname + Fieldname + Isolevel;
      // Check whether result already exists for the given time
Grid IsoSurface = B[ time ][ IsosurfaceName ];
     if (!IsoSurface)
     {
          // No, thus need to retrieve the data volume
     Grid DataVolume = B[ time ][ Gridname ];
          // and perform the actual computation
       IsoSurface = Compute( DataVolume, Fieldname, Isolevel);
          // finally store the resulting data in the bundle object
          B[ time ][ IsosurfaceName ] =  IsoSurface;
     }
     return IsoSurface;
}
```

Note that in case an IsoSurface Grid already exists, there is no need to request a DataVolume object. The operation of requesting such might be effortsome, including slow disk access (on-demand loading), numerical computation of the source field, network access, etc. Never are any data actually stored in the `VizNode` object itself. In this version, a new geometry is stored for each time step and each isolevel value. Since these are floating point values, this may well need to an immense number of surfaces that are stored when exploring the parameter space of time and isolevel. Therefore, some appropriate memory

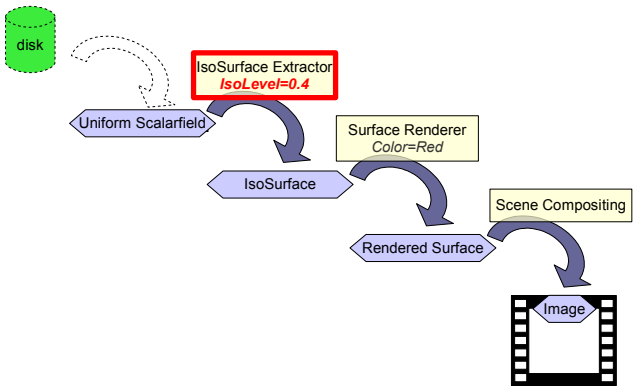


Figure 6: The execution flow: Resulting data flow when changing the isosurface level parameter. The data-set need not be read from disk again.

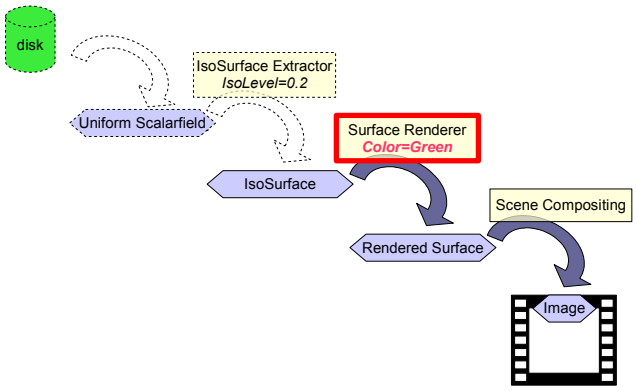


Figure 7: The execution flow: Resulting data flow when changing the color of the iso surface. The isosurface need not be recomputed again.

management that discards old objects that have not been accessed for a long time will be mandatory.

## 2.4 OpenGL Caching

The final stage of producing pixels using modern graphics hardware is loading data onto the memory of the graphics card (GPU). Once all data that are required for rendering are transferred to the GPU, pixel generation will be as fast as possible. Via means of OpenGL, large data at the GPU are modeled as *Display Lists, Textures* and *VertexBuffer Objects*. Framebuffer objects might fall into this category as well, but we did not consider them yet. While the graphics memory is limited, it may still provide enough space to also store objects that are not visible in the currently viewed frame but a previous one. Re-utilizing

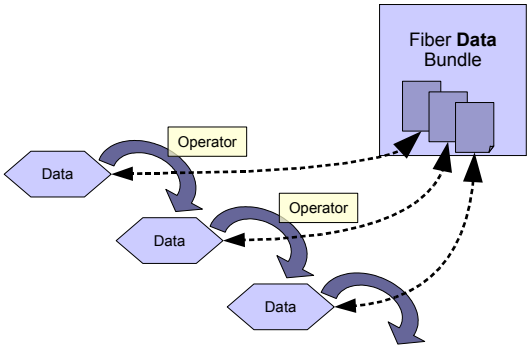


Figure 8: Data Storage in the Fiber Bundle

objects already stored in GPU memory is much faster than re-loading objects from RAM. We may expect so even in case the graphics driver is placing some object into RAM itself.

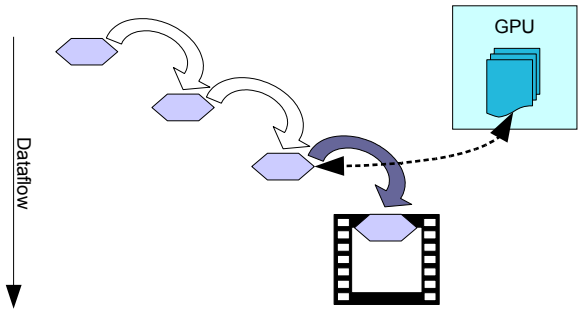


Figure 9: Caching on the GPU

In contrast to caching RAM data like those on the fiber bundle, GPU data is only available through some handle within an OpenGL context. It cannot be stored with the data source. We thus utilize a management system for the OpenGL handle identifiers that is associated with a viewer, called the "`GLCache`". The GLCache is a mapping from certain keys to an OpenGL identifier object (internally just an integer). This mapping is three-dimensional and of the structure:

```
GLuint DisplayList = GLCache[ Intercube ] [ typeid ] [ ValueSet ];
```

Hereby, a given `GLCache` object, a DisplayList identifier can be retrieved by specifying

1. an arbitrary Intercube object
2. an intrinsic C++ type ID
3. a set of values

The functionality is similar to the OperatorCache, where an InterCube and a visualization node is utilized to specify a location of the OperatorCache, plus a set of values used to determine whether re-creation of the data is required. Here, the storage location of the cached objects is provided by the `GLCache`, a parameter that is provided to a visualization object's render routine. The InterCube object (which, for instance, is available with each Grid or Field object within a fiber bundle data-set) is used to find a unique storage location within the `GLCache`. The `typeid` will be the type of the rendering object, such that multiple instances of the same rendering functionality will automatically be able to share their OpenGL objects. The set of values will contain all those rendering parameters which require re-creation of the OpenGL object. A typical rendering code will (schematically) look like this:

```
void VizNode::render(GLCache Context, Grid G)
{
ValueSet VS;
    // assign cacheable variables into the value set
InterCube&C = G;
GLuint DisplayList = GLCache[ Intercube ] [ typeid(this) ] [ VS ];
if (!DisplayList)
   {
       DisplayList = glGenLists(1);
       glNewList(DisplayList, COMPILE_AND_EXEC);
       // do actual rendering of grid data G
       glEndList();
       GLCache[ Intercube ] [ typeid(this) ] [ VS ] = DisplayList;
   }
   else
       glCallList(DisplayList);
}
```

A similar synopsis will be applied for OpenGL object types others than display lists. Some automatic discarding mechanism to ditch unused objects will be required here as well. Note that in case an OpenGL object already exists for a given input data-set (here a "`Grid`" object), then there is no need to actually request the internal data of such an object and to traverse the associated visualization pipeline up to its source, such as shown in Figure 10.

# 3 Conclusion

An universal caching mechanism has been presented. It can be used to extend the visualization pipeline model (as defined by [Haber & McNabb, 1990]) to maximize the reuse of computed data and thereby minimizing the response time of an interactive visualization to parameter changes. The mechanism can relate computed data for all parameters of the visualization, which make fast and easy navigation in the whole parameter space possible. Special emphasis

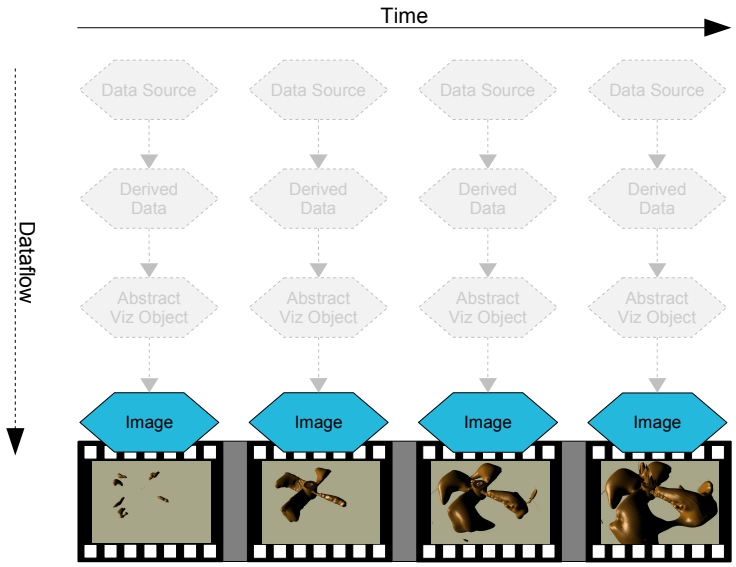


Figure 10: A GPU cached visualization cascade provides the animation without expensive data flow.

is given to the time parameter and time depended data. The implementation demonstration utilizes the Vish framework and also the fiber-bundle model. By incorporating the described GPU caching mechanism full interactivity when browsing an astrophysical data set of 17GB - containing 1.6 million points in 200 time steps - has been achieved. This visualization was run on a Linux 64 bit workstation equipped with a eight 1.6GHz cores (only one of which was used by Vish), 8 GB of RAM and a Geforce Quattro FX5600 graphics card with 1.5GB of GPU memory. The caching mechanism accelerated the visualization to a achieve interactive rates of 30 frames per second when traversing in time in addition to arbitrary spatial camera movement. The first access of the data including reading from disk and processing data in contrast required a couple of seconds for each newly accessed time step. It was also possible to maintain the interactive frame rate while changing parameters such as color-maps and density shape-functions used for volume rendering.

# 4 Acknowledgements

This cooperation research work was supported by the DFG (SCHO 1346/1-1). This research employed resources of the Center for Computation & Technology at Louisiana State University, which is supported by funding from the Louisiana legislature's Information Technology Initiative. Portions of this work were supported by NSF/EPSCoR Award No. EPS-0701491 (CyberTools).

# References


[A.A. Ahmed, 2007] A.A. Ahmed, e. a. (2007). Automatic visualization pipeline formation for medical datasets on grid computing environment.

[Benger, 2004] Benger, W. (2004). *Visualization of General Relativistic Tensor Fields via a Fiber Bundle Data Model*. PhD thesis, FU Berlin.

[Benger, 2008] Benger, W. (2008). Colliding galaxies, rotating neutron stars and merging black holes - visualising high dimensional data sets on arbitrary meshes. *New Journal of Physics*, 10. URL: `http://stacks.iop.org/1367-2630/10/125004`.

[Benger et al., 2007] Benger, W., Ritter, G., & Heinzl, R. (2007). The concepts of vish. In $4^{th}$ *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007* (pp. 26–39).: Berlin, Lehmanns Media-LOB.de.

[Benger et al., 2006] Benger, W., Venkataraman, S., Long, A., Allen, G., Beck, S. D., Brodowicz, M., MacLaren, J., & Seidel, E. (2006). Visualizing katrina - merging computer simulations with observations. In *Workshop on state-of-the-art in scientific and parallel computing, Umeå, Sweden, June 18-21, 2006* (pp. 340–350).: Lecture Notes in Computer Science (LNCS), Springer Verlag.

[Bischof et al., 2006] Bischof, H.-P., Dale, E., & Peterson, T. (2006). Spiegel - a visualization framework for large and small scale systems. In *MSV* (pp. 199–205).

[Black et al., 2002] Black, A. P., Huang, J., Koster, R., Walpole, J., & Pu, C. (2002). Infopipes: an abstraction for multimedia streaming. *Multimedia Syst.*, 8(5), 406–419.

[Card et al., 1999] Card, S. K., Mackinlay, J. D., & Shneiderman, B. (1999). Using vision to think. (pp. 579–581).

[Foulser, 1995] Foulser, D. (1995). Iris explorer: a framework for investigation. *SIGGRAPH Comput. Graph.*, 29(2), 13–16.

[Haber & McNabb, 1990] Haber, R. & McNabb, D. A. (1990). Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*.

[Haeberli, 1988] Haeberli, P. E. (1988). Conman: a visual programming language for interactive graphics. *SIGGRAPH Comput. Graph.*, 22(4), 103–111.

[Hibbard, 1999] Hibbard, B. (1999). Top ten visualization problems. *SIGGRAPH Comput. Graph.*, 33(2), 21–22.

[Johnson & Jennings, 2001] Johnson, G. W. & Jennings, R. (2001). *LabVIEW Graphical Programming*. McGraw-Hill Professional.

[Schroeder et al., 1997] Schroeder, W., Martin, K., & Lorensen, B. (1997). *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall.

[Shen, 2006] Shen, H. (2006). Visualization of large scale time-varying scientific data. *J. of Physics: Conf. Series*, 46, 535–544.

[Treinish, 1997] Treinish, L. A. (1997). Data explorer data model. `http://www.research.ibm.com/people/l/lloydt/dm/dx/dx_dm.htm`.

[Upson et al., 1989] Upson, C., Thomas Faulhaber, J., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., & van Dam, A. (1989). The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9(4), 30–42.