Large Field Visualization With Demand-Driven Calculation

Patrick J. Moran*
MRJ Technology Solutions
NASA Ames Research Center
pmoran@nas.nasa.gov

Chris Henze†
MRJ Technology Solutions
NASA Ames Research Center
chenze@nas.nasa.gov

Abstract

We present a system designed for the interactive definition and visualization of fields derived from large data sets: the *Demand-Driven Visualizer* (*DDV*). The system allows the user to write arbitrary expressions to define new fields, and then apply a variety of visualization techniques to the result. Expressions can include differential operators and numerous other built-in functions, all of which are evaluated at specific field locations completely on demand. The payoff of following a demand-driven design philosophy throughout becomes particularly evident when working with large time-series data, where the costs of eager evaluation alternatives can be prohibitive.

1 Introduction

In many scientific visualization applications, the data sets tend to be large. In computational fluid dynamics (CFD), for example, data can be on the order of one to hundreds of gigabytes in size. CFD data typically come in the form of meshes and fields defined in terms of the meshes. A *mesh* represents the locations of a discrete set of vertices in a domain and the organization of the vertices, for instance to form hexahedral cells. In some cases a mesh may consist of multiple, overlapping submeshes, referred to as *zones*. A *field* has a mesh and a discrete set of nodes where quantities such as density and momentum are represented. The location of the nodes is specified by the mesh. Visualization techniques start with a field and produce images which highlight various features in the domain. Some techniques, such as basic implementations of isosurfaces or volume rendering, require processing every cell or node in the field in order to produce an image. Many other techniques require field values only from small regions within the domain. For instance, a visualization displaying a cutting plane passing through the domain may require only that the field be sampled at points on the plane; or perhaps one may only require data near an aircraft model surface in order to apply LIC techniques [16] or to define contour curves or glyphs. Scenarios where an application touches only a small percentage of the whole data set are known as *sparse traversal* [5]. Data that varies with time tend to magnify the impact of sparse traversal, since sparse access can occur in both space and time. Furthermore, the implications of sparse traversal become more significant with time-series data since the data sets tend to be larger than in steady cases.

An important concept in computational fluid dynamics is that of derived fields. A *derived field* is a field whose values are computed in terms of one or more other fields. Derived fields come into play in simulation applications since programs which solve for field values typically compute only a particular set of fundamental solution values from which all other quantities can be derived. A typical set of fundamental solution variables is density, momentum and energy. There are numerous derived fields that a scientist may be interested in viewing. For instance, Table 1 lists the over 50 derived fields predefined by one CFD post-processing application: PLOT3D [20]. Derived fields are particularly a challenge when working with large data sets, since loading the fundamental solution values alone into main memory may already tax the resources of one's workstation. Furthermore, even if one can afford the memory necessary to store a derived field, much of the computation may be unused if the visualization does not access the whole field.

To address the challenges presented by large data in general and derived fields in particular, we present a visualization system based on a calculator paradigm, the *Demand-Driven Visualizer*. Using the system one can interactively specify derived fields and apply visualization techniques to those fields. The fields can be defined by arbitrary expressions or by any of the standard PLOT3D derived fields, yet the evaluation of derived quantities is completely demand driven (also known as *lazy evaluation*). At the user's option, the calculator can also evaluate and store the derived value over the whole field (*eager evaluation*), or cache lazily evaluated results at some instance in time (*lazy but thrifty evaluation*) for better performance. At the heart of the system is the Field Encapsulation Library (FEL) and a collection of visualization techniques known as the *VisTech Library*. FEL supports the dynamic construction and composition of arbitrary derived fields, and evaluation by lazy or eager methods. *DDV* provides the parsing to convert user expressions to FEL fields, and an interface where the user can interactively choose visualization techniques to apply to the results. The combination of the lazily evaluated derived fields driven by visualization techniques provides a powerful system for field analysis that is especially well suited for large data needs.

The power of lazy evaluation becomes particularly apparent when working with large time-series data sets, where the shortcomings of eager derived field evaluation—memory consumption and unused calculations—are multiplied by the number of time steps in a simulation. Since typical simulations may have on the order of hundreds of time steps, these drawbacks are significant. In the *DDV* design field evaluation is completely driven by the visualization techniques, both in space and in time. The system automatically manages a working set of time steps, doing temporal interpolation if necessary.

In the following section we discuss some previous work related to the *Demand-Driven Visualizer*. Section 3 describes the key demand-driven fields used by *DDV*. Section 4 gives an overview of the interpreter language and the user interface of the system. In Section 5 we present results demonstrating some of the advantages of the *DDV* design. And finally, in Section 6 we conclude with some final thoughts.

2 Related Work

Field and mesh objects in the *Demand-Driven Visualizer* are provided by a C++ class library known as the Field Encapsulation Library (FEL). An initial version of FEL was presented at Visualization '96 [4]. Since then the library has been redesigned and com-

*Mail Stop T27A-2, Moffett Field, CA 94035
†Mail Stop T27A-1, Moffett Field, CA 94035

density	normalized density
stagnation density	normalized stagnation density
pressure	normalized pressure
stagnation pressure	normalized stagnation pressure
pressure coefficient	stagnation pressure coefficient
pitot pressure	pitot pressure ratio
dynamic pressure	temperature
normalized temperature	stagnation temperature
normalized stagnation temperature	enthalpy
normalized enthalpy	stagnation enthalpy
normalized stagnation enthalpy	internal energy
normalized internal energy	stagnation energy
normalized stagnation energy	kinetic energy
normalized kinetic energy	u velocity
v velocity	w velocity
velocity magnitude	mach number
speed of sound	cross flow velocity
divergence of velocity	x momentum
y momentum	z momentum
energy	entropy
entropy s1	x component of vorticity
y component of vorticity	z component of vorticity
vorticity magnitude	swirl
velocity cross vorticity magnitude	helicity
pressure gradient magnitude	density gradient magnitude
velocity	vorticity
momentum	perturbation velocity
velocity cross vorticity	pressure gradient
density gradient	

Table 1: Derived fields predefined in PLOT3D [20].

pletely rewritten in order to support a much wider variety of mesh and field types [10]. In particular, the features essential for large data handling – derived fields, differential operator fields, demand-driven evaluation, working set management of time-series data, and demand-paged data from disk [5] – were not available in the original version of FEL.

The calculator paradigm used in *DDV* is a relatively intuitive and easy-to-use interface for preparing data for visualization. FAST [3], for example, is a CFD visualization system which features a calculator module. In FAST the user can specify arbitrary expressions, including predefined fields such as those in Table 1, and use the resulting scalar and vector fields just as one would use the fundamental scalar and vector fields. The FAST calculator evaluates its results eagerly: new fields require allocating memory and computing the derived value over the whole field. There is little support for time-varying data in FAST.

An alternative paradigm for effectively specifying derivation functions and visualization techniques is data-flow. AVS [19], IBM Data Explorer [8, 1], IRIS Explorer [6], SCIRun [2, 12], and vtk [14] are all examples of data-flow implementations. Data flow systems in general can be classified as either push model or pull model. In a *push model* system, changes to one module cause it to push results downstream through the flow graph. AVS, Data Explorer and IRIS Explorer are examples of push model systems. In a *pull model* design, a change to one module results in data requests propagating upstream through the flow graph, where the appropriate data are processed and effectively pulled downstream. Vtk is an example of a pull model design. In SCIRun modules can operate in either pull mode or push mode [12].

For field visualizations where only a small subset of the field data is required, push model data-flow suffers from the drawbacks of eager evaluation: modules typically operate over the whole field even though ultimately only a relatively small amount of data need be processed, and potentially a large amount of memory must be allocated for buffering intermediate and final results. Memory usage problems can be ameliorated to a certain extent by more careful memory management techniques, or by designing modules that work with finer-grain units of data [18]. One solution to the wasteful computation problem is to introduce filter modules near the head of the flow graph which extract subsets of the data. Unfortunately, it can be difficult in some cases to anticipate what the appropriate subset should be. For example, if the downstream module is a particle tracer, then it may be hard to choose the subregion for computing a derived velocity field, because one would have to know *a priori* where the particles would go. Time-series data adds another dimension to the problem, since it may be difficult to anticipate where temporally a module may need data. For instance, a streakline module may require data over a range of times, including times intermediate to the given time steps (i.e., where temporal interpolation is necessary). One could also imagine scenarios where different modules in the same flow graph may need data at different temporal points in the data set.

In contrast to push model designs, pull model designs offer the potential of better performance in large data, sparse traversal scenarios. In a pull model system, each module can request just the data it needs from the one or more modules immediately upstream. For example, ImageVision [15] is a library for image processing with a pull model design, where operations can be applied to small tiles from much larger images. SCIRun [12] modules can request field values from upstream modules at individual points in space. Such pull model systems represent lazy evaluation embodied in a data-flow setting: the flow graph defines the operations to be applied to the data, but the operations are executed only at specific points or within specific regions, on demand.

Lazy evaluation techniques have also been employed in other visualization systems for large data. The Unsteady Flow Analysis Toolkit (UFAT) [7] is a system designed specifically for particle tracing through large, time-series data. UFAT computes derived field values on demand, but only for the velocity field. Cox and Ellsworth apply a demand-driven approach to the loading of data into main memory [5]. Using demand-paging techniques, they show good performance with large CFD data sets in sparse traversal scenarios, including cases where the fundamental solution data for a single time step are larger than the main memory of the target workstation.

3 FEL Fields

The *Demand-Driven Visualizer* builds upon five key field types in the Field Encapsulation Library (FEL): time-series fields, derived fields, differential operator fields, paged fields, and cached fields. The FEL field classes are defined within a common class hierarchy, and all fields inherit a standard interface defined by the `FEL_field` and `FEL_typed_field<T>` classes at the top of the hierarchy. The typed field class is written using C++ templates where the `T` parameter specifies the field node type, e.g., `float` for a scalar field. Each field instance also has a mesh which specifies the location and organization of the field node data, in FEL a field node is located at each vertex in the mesh. The field interface provides standard methods for accessing field values. An application can request node values at the vertices of a cell (`at_cell`), or at an arbitrary physical position (`at_phys_pos`). FEL uses a general definition for cell: vertices, edges, triangles, quadrilaterals, tetrahedra, and hexahedra are all cells. Calls to `at_cell` do not require spatial interpolation, calls to `at_phys_pos` do Field visualization applications written in terms of the standard "`at`" calls work with any field subclass. FEL field classes include `FEL_core_field<T>`, where the node data are stored in main memory, and other fields where node data may be synthesized on demand. We describe the five types of fields that figure most prominently in the *Demand-Driven Visualizer* design next.

3.1 Time-Series Fields

Large simulation data sets often come in the form of a time series, where each time step represents a snapshot of the field values in time. FEL represents time-series data via the class `FEL_time_series_field<T>`. Time-series fields support the interface common to all FEL fields, thus one can build arbitrary demand-driven fields for time-varying data just as one can for steady data. Visualization techniques request field values using the same arguments as in the steady case: cells and physical positions. Each argument contains a time representation, which is used by `FEL_time_series_field<T>` instances to select the appropriate time step data, or to select multiple time steps when temporal interpolation is necessary. The requirement that the time component of "`at`" call arguments be set is the only difference for the application programmer between using a steady or unsteady field.

`FEL_time_series_field<T>` instances load data for a particular time step on demand, using a callback function provided at construction time. Data are managed in memory using a working set approach, where the time steps are replaced when necessary using a least recently used policy. The size of the working set can be set by the user; thus one can trade-off memory usage for a greater likelihood that a desired time step will be in memory. The working set mechanism contained in `FEL_time_series_field<T>` makes it easier to design applications, such as the *Demand-Driven Visualizer*, for time series data that are much larger than workstation main memory.

3.2 Derived Fields

The derived field classes in FEL are all subclasses of `FEL_derived_field<T>`. For an application programmer, the construction of a derived field requires arguments specifying the fields to be derived from, and a mapping function to be used on demand to produce derived values. All the fields must be based on the same mesh. The *Demand-Driven Visualizer* utilizes several predefined derived field classes, such as `FEL_magnitude_field` and `FEL_sum_field`, where the mapping functions are provided by the library.

An important consequence of defining derived fields in terms of the base class `FEL_typed_field<T>`, rather than a more specific field type such as `FEL_core_field<T>`, is that derived fields can be constructed in terms of other derived fields. In general one can compose fields deriving from any field subclass. This also implies that one can build chains of derived fields to arbitrary lengths. The fact that one can construct new fields without needing to know the specific subclass of the fields being derived from makes it easier to build modular systems. For example, in the *Demand-Driven Visualizer*, derived fields can be composed incrementally as the interpreter traverses an expression parse tree.

The relationships between derived fields can be described using a directed graph. An application builds derived fields node by node, each newly constructed field adding a graph node and edges from previous nodes to the new node. The graphs are acyclic, thus derivation graphs are DAGs (directed acyclic graphs). The derivation graphs can also be thought of as flow graphs. Requests to a particular graph node cause requests to propagate upstream through the flow graph in a demand-pull manner. The data requests are fine-grain: `at_cell` calls require computation only at the nodes of a cell.

3.3 Differential Operator Fields

FEL contains field classes which compute the divergence, gradient or curl of an underlying field. Differential operator field values are computed on demand, similar to derived fields. The library provides classes for computing derivatives by first or second order methods[1] Other differential operators, such as the scalar or vector Laplacian, can also be represented in terms of the built-in operators. Temporal derivatives are not yet implemented in FEL.

As with derived fields, the field provided as a construction argument when building a differential operator field can be any subclass of `FEL_field`. Thus, differential operator fields can be composed into derivation chains just as derived fields are. Second-order differential operator fields are unlike subclasses of `FEL_derived_field<T>` in that they generate additional "`at`" calls on their underlying field in order to acquire a neighborhood of field values surrounding a given argument. For instance, a request for the gradient at a vertex requires field values at the adjacent vertices in the mesh in order to compute the necessary difference values. This expanding neighborhood of calls to fields upstream in the derivation graph is transparent to the end user of a differential operator field.

3.4 Paged Fields

With *paged fields*, the data are organized into page-sized blocks within files on disk. Blocks are automatically loaded into memory, on demand, by the paged field object. The pages are managed using working set techniques. The loading and management of blocks is transparent to the paged field user. The `FEL_paged_field<T>`

[1] Presently only first order methods are supported for unstructured meshes.

Operator	Description
`__add__`	addition (infix +)
`__div__`	division (infix /)
`__mul__`	multiplication (infix *)
`__neg__`	negation (unary -)
`__sub__`	subtraction (infix -)
`cross`	cross product
`curl1`	first-order vector curl
`curl2`	second-order vector curl
`div1`	first-order divergence
`div2`	second-order divergence
`dot`	dot product
`grad1`	first-order gradient
`grad2`	second-order gradient
`mag`	vector magnitude
`sqrt`	square root

Table 2: Field math operators defined in *DDV*.

class encapsulates the approach presented by Cox and Ellsworth at Visualization '97 [5].

3.5 Cached Fields

The derived and differential operator fields in FEL follow a maximally lazy strategy. No derived values are computed in advance, nor is any memory allocated for storing derived values. In cases where an application repeatedly requests values at the same locations in a field, the maximally lazy approach may not be the best, since the derived values would be recomputed at each request. On the other hand, eager evaluation may still not be the best choice, particularly in sparse traversal situations. FEL provides a hybrid approach via a field class: `FEL_cached_field<T>`. A cached field is constructed with another FEL field instance as an argument. Cached fields allocate the memory to store the whole field (at one instance in time) and mark each node with a special "unevaluated" value. For each "`at`" call, a cached field checks whether the requested node values have been evaluated already, and returns previously computed values if available. Node values requested for the first time are computed as in the uncached case, and stored for future reuse. The time component of the `at` call argument is ignored, thus it is inappropriate to use a cached field if the the underlying field is time varying and the time specified in all the `at` calls is not the same.

In sparse traversal scenarios, cached fields provide amortized response time close to that of eager fields, without the wasteful computation drawback of eager evaluation. For demand-driven fields that are expensive to evaluate, in particular differential operator fields, cached fields can significantly improve performance when one can afford the memory.

4 Implementation

The *Demand-Driven Visualizer* provides a graphical interface allowing the user to interactively define and visualize arbitrary derived fields. In this section we provide a brief overview of the interpreter language used to express such fields, and the rapid application development language used to build the system — Python.

4.1 The Language

The *DDV* interface includes an interpreter window where the user can write and evaluate field expressions (see Figure 1). The interpreter in *DDV* is based on *Python* [9, 13], an interactive, interpreted language. A key feature of Python is its extensible, modular design. *DDV* provides an FEL module for Python which introduces mesh and field types into the interpreter environment. The types are first class, in other words, once the FEL module is imported one can use the mesh and field types just as one uses other built-in types. For instance, field types can be used in expressions, assignment statements, or passed as arguments to user-defined routines. Python parses expressions, using operator precedence similar that in the C language, building a parse tree internally. Table 2 lists the operators that can take field arguments in an FEL-extended Python. The interpreter traverses the parse tree, building FEL fields as directed by the tree. The demand-driven nature of FEL is essential here: the interpreter can traverse and build at interactive rates, even though the fields may be extremely large.

Data	Per Time Step	# Steps	Total
SSLV	599	1	599
DW	22	200	4391
F18	35	301	10652

Table 3: Data set sizes (MBytes).

4.2 Visualization Techniques

Once one has defined fields within the interpreter environment, the next step is to apply visualization techniques. *DDV* utilizes a C++ suite of visualization techniques known as the *Vistech Library* [17]. For each visualization technique, *DDV* provides a Python wrapper. Within the interpreter environment one can construct visualization instances and view the graphical output. Visualization instances can also be constructed via a menu-driven interface, described next.

4.3 The Graphical User Interface

The graphical user interface (GUI) of the *Demand-Driven Visualizer* is illustrated in Figure 1. The GUI is written using the `Tkinter` interface provided by Python. `Tkinter` is a wrapper around `Tcl/Tk` [11]; like `Tk`, `Tkinter` allows system designers to specify a graphical user interface in a windowing-system-independent manner. The *DDV* GUI gives the user choice: novice users can use the pull-down menus and buttons to construct and control visualization instances, while advanced users can use the interpreter command line alone to control the application. Python provides a universal language that supports both the specification of fields for visualization and the command and control of the application.

5 Results

To demonstrate the effectiveness of the *Demand-Driven Visualizer* with large data sets, we begin by quantifying the sparse data access patterns typical of many visualization techniques. Next, we show how the *DDV* exploits such patterns, avoiding the drawbacks of eager evaluation. The example derived fields and visualizations are computed for three CFD data sets: the space shuttle launch vehicle (SSLV), the delta wing (DW), and the F-18 fighter (F18). The SSLV data set is a steady simulation and has a mesh consisting of 113 zones. The delta wing and F-18 data sets represent time-varying single and multi-zone flow simulations, respectively. The delta wing mesh also varies with time, the F-18 mesh does not. Table 3 summaries the data set sizes.

A first step towards confirming that a lazy evaluation strategy will be effective is to verify that many visualization techniques require accessing or "touching" only a small percentage of the field values in a data set. Touching a small fraction of the data implies

Derived Field	Data	Eager		Lazy		Cached Lazy	
		Cons.	Visu.	Cons.	Visu.	Cons.	Visu.
`density`	SSLV	40.96	1.50	ε	1.75	ε	5.74
	DW	0.90	0.15	ε	0.19	ε	0.28
	F18	4.89	0.20	ε	0.25	ε	0.35
`pressure`	SSLV	41.94	1.50	ε	1.73	ε	6.01
	DW	0.96	0.15	ε	0.19	ε	0.28
	F18	4.99	0.20	ε	0.25	ε	0.35
`dot(grad2(pressure), velocity)`	SSLV	286.64	1.49	ε	10.51	ε	10.46
	DW	10.11	0.15	ε	1.22	ε	0.60
	F18	33.38	0.20	ε	1.73	ε	0.83
`vorticity_magnitude`	SSLV	341.05	1.50	ε	12.87	ε	11.06
	DW	12.86	0.15	ε	1.47	ε	0.68
	F18	39.66	0.20	ε	1.98	ε	0.92

Table 5: Construction and visualization timings (in seconds) for four derived fields, ordered by increasing expense to evaluate (times designated ε are less than 1 millisecond). In all cases the time to construct a lazy field and apply a visualization technique is much less than the construction time alone for an eager field. The table also shows that caching can improve the performance of a visualization based on a lazy field that is expensive to evaluate, but caching can hinder performance when evaluation is cheap.

Derived Field *Visualization*	Data	Touched	
		> 0	> 1
`density` *Cutting plane*	SSLV	1.2	0.7
	DW	3.2	1.8
	F18	1.5	0.8
`divergence_of_velocity` *Cutting plane*	SSLV	2.4	2.0
	DW	6.2	4.8
	F18	3.0	2.3
`velocity` *Particle advection*	SSLV	0.1	0.1
	DW	1.1	0.7
	F18	0.4	0.3

Table 4: The percentages of nodes touched at least once, and more than once, for typical cutting plane and streamline visualizations. The numbers are typical of visualization algorithms with sparse traversal behavior.

that a large fraction is untouched, and a large fraction untouched implies a large amount of unused computation in an eager evaluation design. We also measure how many field nodes are touched more than once by the example visualization techniques. Cases where many nodes are touched more than once suggest opportunities where caching could make a significant improvement, since more values would be reused. Table 4 summarizes the measurements. The `density` and `divergence_of_velocity` scalar fields were visualized using a cutting plane sampling, and the `velocity` field was visualized via a particle advection technique. The percentages show the relatively low number of nodes touched, in most cases less than 5%. The exact statistics vary with the positioning of the cutting plane or particle advection rake. For example, with the SSLV data, a plane cutting between the shuttle and fuel tank passes through several fine detail meshes, increasing the touch counts. The SSLV counts in Table 4 are for such a plane position. The counts for the `divergence_of_velocity` field are for a plane in the same position as for the `density` field. Note that the touch counts for the divergence field are higher, since node values from a neighborhood surrounding the plane are required. Note too that the percentage of nodes touched more than once is over half the percentage of nodes touched at all, suggesting that caching derived results may improve performance.

The consequences of choosing a demand-driven design over an eager-evaluation design become apparent when we consider the times required to compute derived field visualizations. Table 5 summarizes the performance of four example fields using eager, lazy and cached-lazy evaluation techniques. In order to focus on the performance differences due to the different types of derived fields, the timings are for a single visualization technique, sampling with a cutting plane. The measurements were taken on an SGI Onyx2 workstation with one GByte of main memory and a 195MHz processor clock rate. The four derived fields include a trivial derived field, `density`, two commonly used non-trivial derived fields defined by PLOT3D [20], and a custom defined field (pressure gradient dotted with velocity) sometimes used for feature detection.

To isolate the costs of eager evaluation, the timings are broken down into construction and visualization contributions. In the case of every field and data set combination (i.e. every row in the table), the eager construction time dominates. In many cases the difference between the eager construction time and the visualization time using a lazily evaluated field is over an order of magnitude. Thus even in cases where the user desires multiple visualizations over the same derived field, the total time consumed using a lazily evaluated field would still be less than going the eager route. Note too that by taking the lazy evaluation approach the user also comes out ahead in terms of memory consumption. Eager fields require significant amounts of memory for storage. Furthermore, in some cases one may have to use yet more memory, at least temporarily, to store the intermediate fields used to compute a field defined in terms of other derived fields. In cases where the fundamental solution values alone consume much of the memory of one's workstation, not having to store derived values may make the difference between reasonable performance and thrashing.

Table 5 also lists the times to construct and utilize lazily evaluated fields where derived values are cached for reuse. The numbers show improvements when working with relatively expensive derived fields, but a downgrade in performance when caching is coupled with fields that are cheaper to evaluate. To decide whether caching will be effective, one has to consider not only the field evaluation cost, but also the field access patterns of the visualization technique, and the amount of memory available for caching. We are continuing to study these trade-offs as we further optimize the the performance of the field classes and visualization techniques.

6 Conclusion

We have presented the *Demand-Driven Visualizer*, a system designed from the start to address large data visualization needs through demand-driven evaluation techniques. The system features a general, flexible interpreter where one can define arbitrary derived fields, yet still enjoy the advantages of lazy evaluation. Lazy evaluation excels in sparse traversal scenarios, i.e., in cases where an application touches a relatively small subset of the data. Many visualization techniques, such as particle tracing or visualizations defined on a surface within the domain, exhibit sparse traversal behavior. The issues addressed by a demand-driven design are especially pronounced with time-series data: eager evaluation with unsteady data can lead to a great deal of unused computation and memory consumption that can overwhelm most workstations.

It is important to note that key to the effectiveness of *DDV* are several demand-driven design features working in concert. Using a lazy evaluation model in some places and eager in others can lead to a counterproductive design. For example, demand-paging has been shown previously [5] to be an effective approach to large data visualization, but demand-paging coupled with eager derived field evaluation would be self defeating. Eager derived fields would force every page to be read in, and the memory consumption of the derived field would often outweigh the savings gained with paging. Since derived fields are often desired in simulation data visualization, it is important to take an approach that preserves and extends the benefits gained by other large data visualization techniques, both for steady and unsteady flow.

Acknowledgements

This work was supported by NASA contract NAS2-14303. We would like to thank Reynaldo Gomez, Neal Chaderjian and Ken Gee for making the space shuttle launch vehicle, delta wing and F-18 data sets available, respectively, for visualization studies. We would also like to thank Guido van Rossum and the Python community [13] for providing the application development language used by the *DDV*.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for a data analysis and visualization. In *Proceedings of Visualization '95*, pages 263–270. IEEE Computer Society Press, 1995.

[2] S. Parker amd D. Weinstein and C. Johnson. The SCIRun computational steering software system. In E. Arge, A. Bruaset, and H. Langtangen, editors, *Modern Software Tools for Scientific Computing*. Birkhäuser, 1997.

[3] G. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–24. IEEE Computer Society Press, October 1990.

[4] S. Bryson, D. Kenwright, and M. Gerald-Yamasaki. FEL: The field encapsulation library. In *Proceedings of Visualization '96*, pages 241–247. IEEE Computer Society Press, October 1996.

[5] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization '97*, pages 235–244. IEEE Computer Society Press, October 1997.

[6] IRIS Explorer Center. `http://www.nag.co.uk/Welcome_IEC.html`.

[7] D. Lane. UFAT: A particle tracer for time-dependent flow fields. In *Proceedings of Visualization '94*, pages 257–264. IEEE Computer Society Press, October 1994.

[8] B. Lucas et al. An architecture for a scientific visualization system. In *Proceedings of Visualization '92*, pages 107–114. IEEE Computer Society Press, 1992.

[9] M. Lutz. *Programming Python*. O'Reilly & Associates, Inc., 1996.

[10] P. Moran, C. Henze, and D. Ellsworth. The FEL 2.2 user guide. Technical report, National Aeronautics and Space Administration, 1999. NASA Technical Memorandum.

[11] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[12] S. Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999. *Feb. 1999 draft*.

[13] Python. `http://www.python.org`.

[14] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice-Hall Inc., New Jersey, second edition, 1997.

[15] SGI. ImageVision Technical Report. Technical report, Silicon Graphics Incorporated, 1995. `http://www.sgi.com/Technology/ImageVision/techreport`.

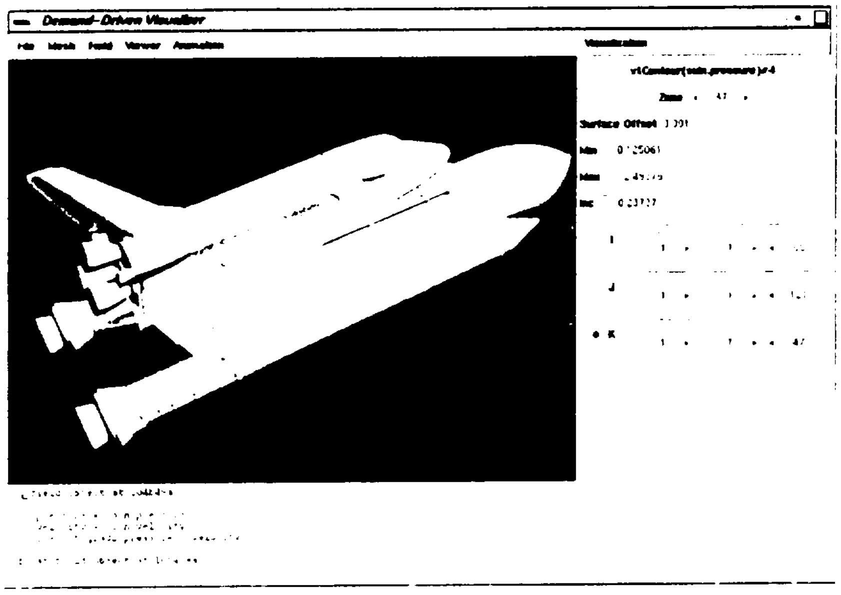
[16] H.-W. Shen and D. Kao. UFLIC: A line integral convolution algorithm for visualizing unsteady flows. In *Proceedings of Visualization '97*, pages 317–322. IEEE Computer Society Press, October 1997.

[17] H.-W. Shen, T. Sandstrom, D. Kenwright, and L.-J. Chiang. *VisTech Library User and Programmer Guide*. National Aeronautics and Space Administration, 1999.

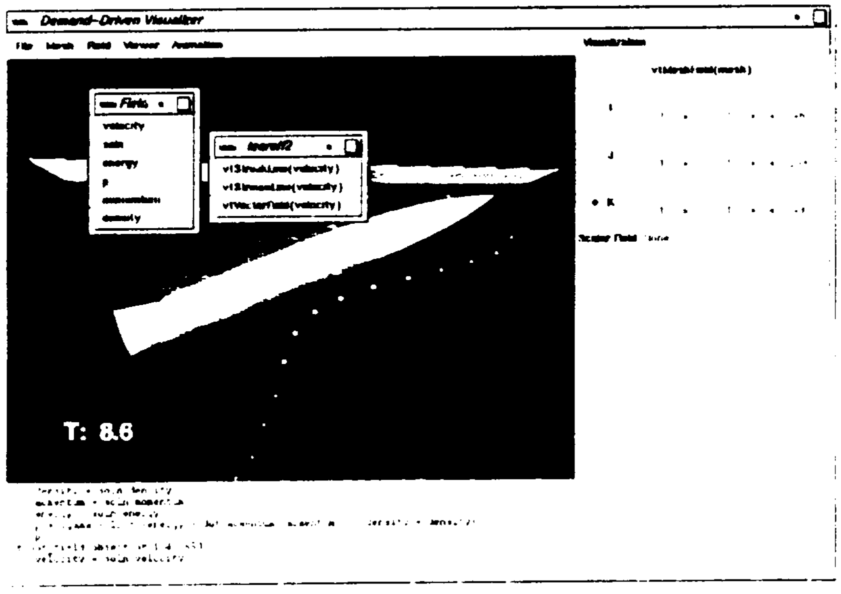
[18] D. Song and E. Golin. Fine-grain visualization in data flow environments. In *Proceedings of Visualization '93*, pages 126–133, October 1993.

[19] C. Upson et al. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, July 1989.

[20] P. Walatka, P. Buning, L. Pierce, and P. Elson. *PLOT3D User's Manual*. National Aeronautics and Space Administration, July 1992. NASA Technical Memorandum 101067.



[illegible] graphical user interface and the SSFA data set. The contours on the shuttle body are for pressure [illegible]



```
density = soln.density
momentum = soln.momentum
energy = soln.energy
velocity = soln.velocity
```

[illegible] DW data set. The [illegible] (left) [illegible] the fields currently [illegible] [illegible] that are [illegible] the fields [illegible]