Fast Focus+Context Visualization of Large Scientific Data

Martin Gasser[*]

VRVis Research Center, Vienna, Austria

Abstract

Visualization of high-dimensional and time-dependent data, resulting from computational simulation, is a very challenging and resource-consuming task. Here, feature-based visualization approaches, which aim at a user-controlled reduction of the data shown at one instance of time, proof to be useful.

In this paper, we present a framework for interactive high- dimensional visualization and feature specification. We put special emphasis on the design of the *data access layer* and the performance optimization of the highly interactive parts of our system.

Keywords: interactive visualization, focus+context visualization, feature specification, large data, linked views

1 Introduction

CFD simulation (computational fluid dynamics) is a method frequently employed to gain insight into flow phenomena that cannot be experimentally analyzed (at least not without extraordinary efforts).

For example, oxidation processes in a catalytic converter are difficult to monitor in reality [4]. Typical questions would be how many toxic substances are present in the exhaust fumes at a certain point in time or how much soot is accumulated during its operation.

Besides the overhead for getting data samples that are accurate enough for the representation of subject state in an experiment, each experiment would require the construction of a new prototype. So it can be stated that computational flow simulation drastically shortens development cycles and allows the involved engineers to optimize their models before a real prototype is built.

To efficiently support users during analysis time, adequate visualization of the simulation results is needed. Complex physical and chemical processes imply large result data, since a large number of potentially interesting data dimensions is involved. Data size also depends on the duration and the temporal resolution of unsteady simulations. Typical sizes of data sets lie in the range from 250.000 to 1.000.000 data points, 15 data dimensions and 20 – 100 time steps. For example, a data set with 500.000 data points, 10 dimensions of high-precision data and 100 time steps would result in a file size of approximately 480MBytes. In the future, dataset sizes in a magnitude of $>$ 1.000.000 data points and noticeably more than hundred time steps can be expected, resulting in file sizes of several GBytes. Loading such a dataset entirely to system memory usually is not a practical solution, therefore sophisticated data access policies are a crucial requirement.

Visualization of high dimensional and large data is a very demanding procedure, requiring several problems to be solved. Besides the functional requirements posed to the software (e.g. loading/saving of feature specifications, cached data access), several non-functional-requirements had to be met as well:

- Visualization at interactive frame rates
- Open architecture for addition of new visualization methods
- Best possible exploitation of hardware resources

In this paper a framework is presented, which supports interactive visualization and analysis of complex data using multiple linked views, combining approaches from scientific visualization (SciVis) and information visualization (InfoVis).

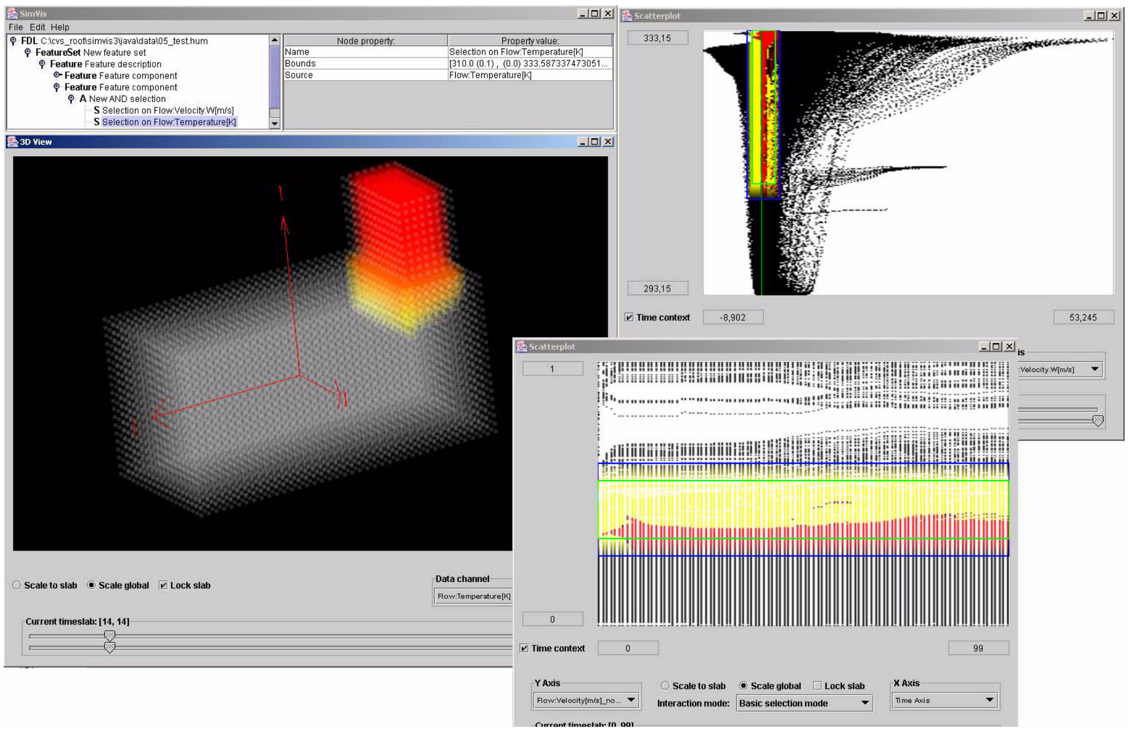
2 Concepts

Below we explain several concepts that were incorporated in our system.

2.1 Feature-Based Visualization

Due to the large overall amount of simulation data, not all the data can be shown simultaneously. *Data reduction* is a crucial task in many visualization systems. We support a feature based approach, enabling the user to interactively select essential data subsets he or she wants to view in special detail. So, the visual discrimination of data in focus from contextual data is supported by *Focus+Context* visualization.

[*]Martin.Gasser@VRVis.at



Node property:	Property value:
Name	Selection on Flow:Temperature[K]
Bounds	[310.0 (0.1) , (0.0) 333.587337473051...
Source	Flow:Temperature[K]

Figure 1: The SimVis application, showing the flow of hot gas through an object. Two scatterplots are used to mark regions of interest, which are then visualized in the 3D view as color coded points.

2.2 Linked Views

High dimensionality is a common challenge in information visualization (InfoVis) that can be effectively handled with multiple views that show the same data through different projections, allowing the user to discover data clusters and other interesting features with an explorative approach.

InfoVis usually is employed for the visualization of economic and other rather abstract data without any spatial dimensions. Scientific visualization (SciVis) usually deals with spatial data and the reconstruction and visual enhancement of sampled or synthetic geometry. By linking InfoVis and SciVis views, it is possible to locate a data item in the data domain as well as in 3D.

Linked InfoVis/SciVis-views can also substantially decrease the perceptual complexity of visualizations, especially when the data dimensionality increases. Multiple views can be used to localize data items in multiple projections of the data domain as well as in the spatial domain. A brushing widget can be used to mark regions of interest in one view, triggering an immediate update in the other views. The here employed technique is also called *Linking&Brushing* [3].

Figure 1 shows an overview of the application GUI. Two scatterplots and one 3D rendering view are used. The visualized dataset shows the flow of hot gas through sample geometry. The views show the data distribution of various dimensions.

2.3 Feature Definition

In addition to the already existing semi-automatic approaches for feature extraction (see Post et al. [11] for an overview), we follow a highly interactive approach. Users can *brush* [1, 12] data in the various views by directly pointing at the data with an appropriate device (e.g. mouse) and selecting the regions of interest. Simple brushes are defined by giving lower and upper bounds for data values.

Simple brushes can be logically combined (see section 3.1 for details) into more complex brushes (i.e. a set of data points that satisfy two or more constraints) and it is not only possible to say *if* something is important but also *how* important it is by employing a technique called *Smooth Brushing* [3].

Conceptually, a brush maps each (n-dimensional) data item to some *degree of interest* (DOI) [6], hence this map is called the *degree-of-interest* function. A possible way of how to implement this is a mapping from each data item to a value from the continuous unit interval $[0, 1]$, where 1 means maximal importance (or *full focus*) and 0 means no importance (or *context*).

During visualization, the views can perform focus+context discrimination by modulating the color and the opacity of data items with respect to their DOI values.

To sum up, brushes are represented as logical combinations of ranges on the individual data dimensions. Like from simple range selections, a DOI can be derived from

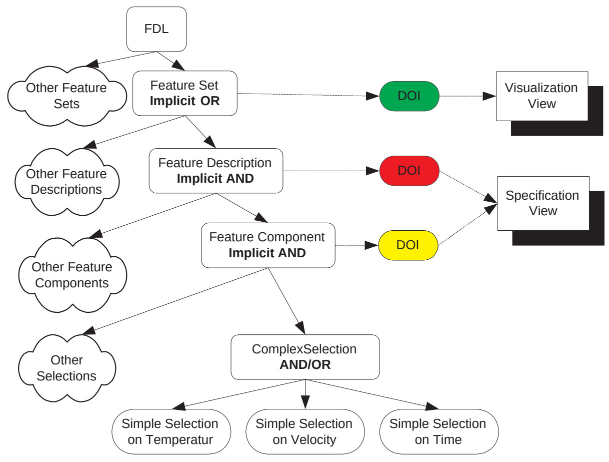


Figure 2: A example FDL tree with associated DOI

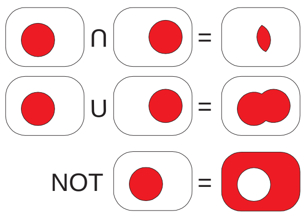


Figure 3: Possible set operations

the hierarchical combination of simple brushes, as well.

Since this hierarchical structure allows feature definitions of arbitrary complexity in a syntactically and semantically well-defined manner, it was called *Feature Definition Language* (FDL) [2]. In Figure 2, an example FDL tree is depicted.

Feature Definition Language

A typical FDL specification has a tree structure, as depicted in figure 2. The meaning of the different node types in the hierarchy can be best described in terms of logical set operations. Since we need not only a binary classification of features, membership and logical combinations were designed as fuzzy set operations (see section 3.1).

The bottom level of the hierarchy is formed by *SimpleSelections*. One SimpleSelection constrains the valid data range for data points in one dimension. The output of a SimpleSelection is a DOI function that classifies all data points that match the criterion as belonging to the feature.

A *ComplexSelection* can compute the intersection, the union, or the complement of sets of points, resulting in a DOI function that classifies all data points matching the combined criterions of the child nodes. For an illustration of the possible set operations, see figure 3. Since complex selections can be recursively combined, feature specifications of arbitrary complexity and dimensionality can be constructed.

Feature Components, *Feature Descriptions*, and *Feature Sets* are specializations of Complex Selections and were designed to provide additional support for the needs of users in analysis sessions.

Typically, a user specifies one feature in high-dimensional space by exploring the data in multiple projections of that space and brushing interesting structures in selected projections. A brushing of data items in one view is then immediately reflected in the other views by coloring the affected points. Individual *features* are identified with *Feature Descriptions*. Each view involved in the definition of a feature is bound to a *Feature Component*, providing a bidirectional mapping between views and FDL tree nodes. Additionally, the user may view the combination of *all* features she or he specified in a top level view (usually the 3D view), which is provided by a *Feature Set*.

In order to distinguish points which are marked only in the *current* view (but are not present in the combination of constraints as generated by other views) from points that are marked in all views, a *multi-level DOI* concept was introduced. Views used to specify features show two point sets as described by DOI functions, one resulting from the Feature Component which the view is bound to and one from the Feature Description that is build up by the individual components. View discrimination is then performed by coloring data points from the Feature Set in red tones, while data points from the Feature Component only are drawn in yellow tones (see the scatterplot views in Fig. 1 for an example).

External Representation

To store and load feature definitions, an external representation of the FDL has been designed. Due to the inherent hierarchical nature of the FDL, it has been implemented as an XML [17] application. Since XML was designed to be human readable, this realization enables power users to script feature definitions by directly manipulating the XML files with an ordinary text editor or a special XML editing application.

However, the average user does not formulate FDL statements directly. The views automatically produce feature definitions during user interaction. Sometimes, this is not fully appropriate (e.g., if known data boundaries have to be entered with high precision).

To provide mid-level feature manipulation without the need to start an external text editor, an additional view has been designed that shows the tree structure directly in a tree-like interface. Tree node attributes (like selection bounds or the different logical combinations) can be manipulated manually and entire subtrees can be moved with standard interaction techniques like copy&paste and drag&drop.

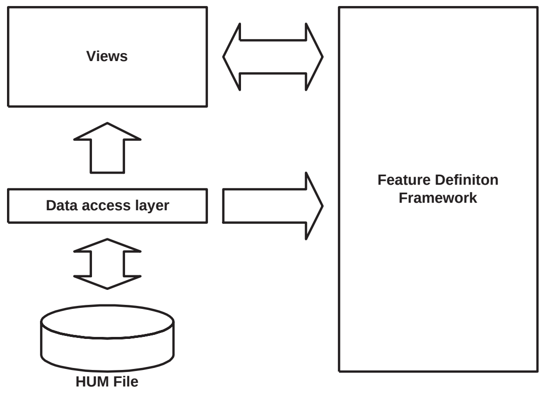
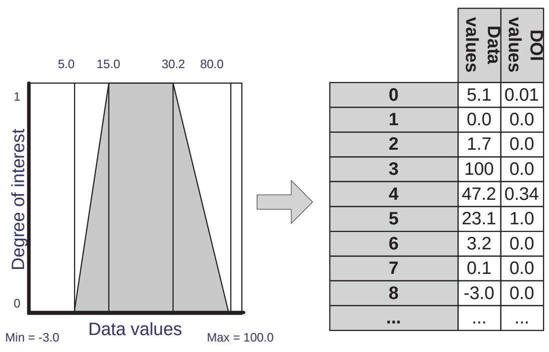


Figure 4: Main software components



	Data values	DOI values
0	5.1	0.01
1	0.0	0.0
2	1.7	0.0
3	100	0.0
4	47.2	0.34
5	23.1	1.0
6	3.2	0.0
7	0.1	0.0
8	-3.0	0.0
...	...	...

Figure 5: On the lowest level of the feature definition framework, each n-dimensional data point is mapped to a DOI value, according to its value in one dimension

2.4 Time Dependent Data

Special UI elements had to be designed to support the handling of time-dependent data. It is now possible to define a specific time interval in each view. Data from this interval is then visualized in the view, with successive time steps painted one over the other.

To allow more specific selections in the *time* dimension, we have added the time values for each data point as a first order dimension.

2.5 Data File Format

We have defined the *HUM* (High performance unstructured mesh) file format for storing the simulation data and the corresponding geometry. A separate converter application has been implemented to enable the conversion of data from our main industrial partner in this project to our own data format. The internal representation of the data (see section 3.3) is closely related to that format.

3 Software Architecture

Figure 4 shows the data flow between the main software components. The *Data Access Layer* implements transparent data access and caching. The *Views* and the *Feature Definition Framework* perform bidirectional communication, relying heavily on the data access services.

3.1 Feature Definition Framework

The FDL framework is built around a hierarchical tree structure which logically combines simple one-dimensional selections to more complex feature specifications following a descriptive and modular approach.

From each node in the tree, a degree of interest function can be recursively derived.

Each *SimpleSelection* (forming a leaf of the tree) defines a trapezoidal DOI function (see Fig. 5) that can be interpreted as a fuzzy membership function.

DOI functions derived from simple selections can be logically combined to selections of higher dimensionality. Logical operations are currently realized as fuzzy minimum t-norms (for AND) and maximum t-conorms (for OR) and the complement on one (for negations) [8].

$d1 \wedge d2 = min(d1, d2)$
$d2 \vee d2 = max(d1, d2)$
$\neg d = 1 - d$

3.2 Views

Each feature specification stores a reference to the view which created it and views can be opened and closed on demand. Changes to tree nodes fire events that are propagated up the tree, resulting in an update of each view associated to a tree node in the path from the affected node to the root. The views act as *Observers* [7] that are notified whenever changes occur in the observed objects (feature specifications). Hence the feature specification hierarchy can be seen as a hierarchic hub, forwarding update requests only to those views that are affected by it and realizing implicit linking of all views that belong to the current subtree.

In the following, a short overview of some of the views that we already integrated into our system, is given.

Scatterplot

A (2D) scatterplot shows two-dimensional projections of the data space. Two components of an n-dimensional data point are interpreted as coordinates in a Cartesian coordinate system. Figure 6 shows a scatterplot that draws the *temperature* values of the data points versus their *relative pressure* values. Points with medium relative pressure and high temperature have been classified as features with a smooth brushing widget.

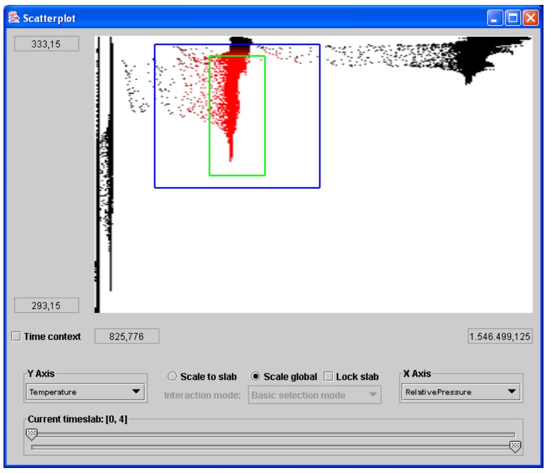


Figure 6: A cluster in the combined distribution of *temperature* and *pressure* is selected in a scatterplot

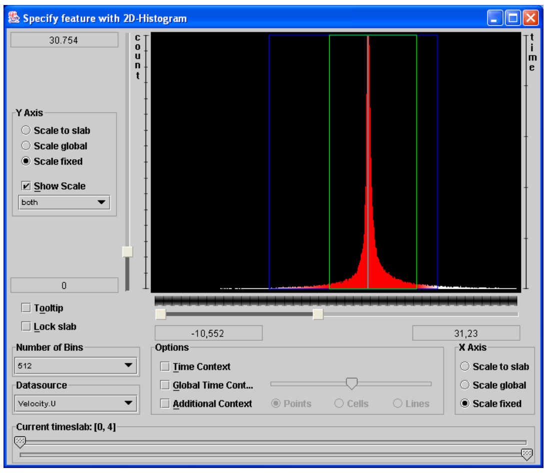


Figure 7: Peaks in the distribution of *velocity in x-direction* are selected in a histogram

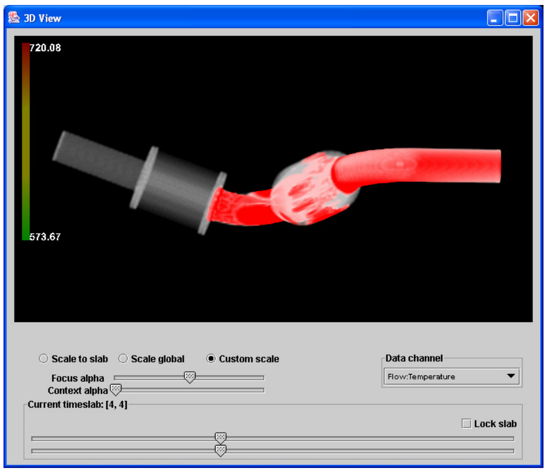


Figure 8: The 3D view, showing hot regions in a catalytic converter

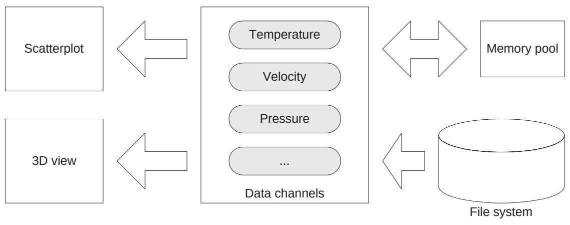


Figure 9: Overview of the data access layer

Histogram

A histogram counts the frequencies of occurrences of data values in a particular data dimension. Since this is not directly possible for continuous data, the data range has to be segmented into bins (small subranges) that represent a discretized version of the data space.

We implemented *TimeHistograms* as described in Kosara et al. [9]. TimeHistograms offer special support for time-dependent data, like visualization of trends or three-dimensional histograms with time as third axis (see figure 7).

3D View

The 3D view is, as opposed to the previous two InfoVis views, an extended SciVis view. It shows the distribution of the data points classified as belonging to a feature in 3D space but also gives hints about their position in data space via customizable color coding (e.g., the temperature range of points can be mapped to a color gradient from red to green).

Since continuous DOI functions are supported, the opacity of the displayed points is modulated by the corresponding degree-of-interest values. Data points with high DOI values are drawn opaque while a low degree of interest results in a rather translucent appearance, discriminating data *in focus* from *context* data.

In figure 8, a 3D view shows hot and fast regions in a simulation of a catalytic converter.

3.3 Data Access Layer

As already stated, the data access part of the application is of particular importance for the efficient operation of the software. Figure 9 illustrates the main system components and the data flow between them.

In our data representation, data from one time step is stored in *Channels*. The channel class encapsulates an array of data values and provides bounds-checked accessor functions for those values. Multiple *Channels* can be collected in a *TimeChannel*. A *TimeChannel* organizes individual time steps, represented as channels and provides some meta data. Basically, one dimension can be associated with a *TimeChannel*.

Channels are implemented following a *Virtual Proxy* pattern [7], making transparent lazy loading (*activating*)

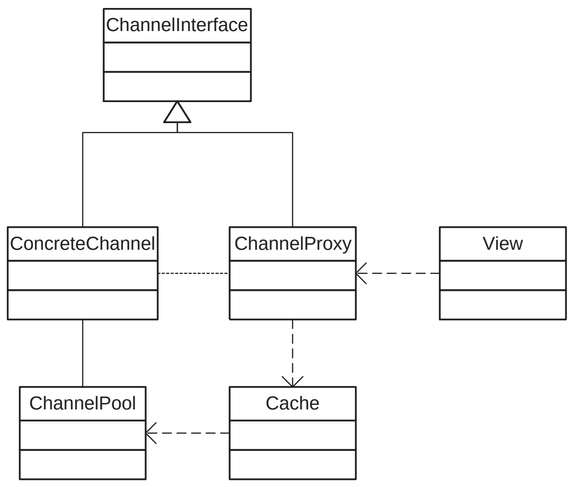


Figure 10: The components associated with the data activation process

of data possible. Figure 10 illustrates the data activation process and the associated components.

Due to the fact that it is impossible (or at least very expensive in terms of system resources) to leave all channels activated, a caching algorithm based on a LRU queue has been implemented.

The following requirements were the cornerstones for the design of the caching subsystem:

- Minimize the usage of the page replacement system provided by the OS
- Minimize the number of harddisk read accesses
- Minimize allocation/deallocation of large memory blocks

Activated *ChannelProxies* are stored in a global queue, with the last accessed proxy always at the head of the queue. Whenever a *ChannelProxy* is accessed, the caching layer ensures that it is either moved to the front of the LRU queue or activated and then inserted at the front of the queue. If the channel has been already activated, the function immediately returns.

The activation function itself tries to get the least recently used *ChannelProxy* from the queue and reuses its *ConcreteChannel*. If the channel stores read-only data from a file, the *Cache* can just reload the data if it is needed. Generated data (e.g., data that has been derived from existing data by applying a filter) is swapped out to another file if needed.

If the queue is empty – because all data channels are currently in use – the caching layer allocates an empty *ConcreteChannel* from the *ChannelPool*. The acquired *ConcreteChannel* is assigned to the *ChannelProxy*, which is then loaded with the requested data.

In the following, the caching algorithm is sketched.

```
if (not QueueIsEmpty()) {
    LRUProxy = RemoveLastQueueItem();
    SwapOut(LRUProxy);
    EmptyChannel = GetChannel(LRUProxy);
    AssignChannel(aProxy, EmptyChannel);
}
else {
    EmptyChannel = GetChannelFromPool();
    AssignChannel(aProxy, EmptyChannel);
}

SwapIn(aProxy);
```

4 Implementation issues

The application was implemented in C++ (Visual C++ .NET) and Java (j2sdk 1.4.2). We combined the advantages of C++ – mainly the high optimizability of the code – and the advantages of Java – rapid user interface design – as good as possible.

Essentially, all performance-critical parts were implemented in C++. These parts include low level data access, OpenGL rendering, DOI calculation, and data filtering. The user interface components were implemented in Java.

4.1 Java-native Binding

We implemented the binding of native C++ object instances to their corresponding Java objects with JNI [15] and an adapted proxy [7] pattern. All Java objects that have a native counterpart, act as proxies by just forwarding requests and caching values for succeeding read operations.

Basically, our concept is simple: A Java object stores the address of its corresponding C++ native object. Whenever a method of this object is called, another method with the same signature, but marked as *native*, is called; this method is implemented in plain C (since JNI is a C API). In this C function, a pointer to the corresponding C++ native object is constructed from the address stored in the Java object and the object property is set. The obvious drawback of this solution was the large coding overhead for the C wrapper functions (which could be easily avoided with a code generation utility).

Since it was a requirement to use OpenGL for the rendering part of the views, we needed a way of how to natively render to Java UI components. Since the JNI API did not expose native UI handles from the AWT before the arrival of JDK1.3, the first prototype of the system used the GL4Java API. Due to the developement stop of this OpenGL implementation Java we soon decided to move to a less proprietary solution. The JAWT interface (Java Abstract Windowing Toolkit native interface) introduced calls that returned handles to native windowing resources with the release of JDK1.3, which made it possible to create OpenGL rendering contexts [13].

5 Performance Optimizations

5.1 High Level Optimizations

The first version of the software was strictly synchronous, resulting in a blocking behaviour due to the fact that 90% of the computing time was spent in the calculation of the DOI values and mouse interaction continuously triggered a recomputation of the DOI values. Large datasets left the system completely unusable because mouse interaction was impossible.

Our solution was to decouple the DOI computation from the user interface in a main thread and a DOI computation thread. The following pseudo code segment illustrates the algorithm:

```
function Recompute(Node) {
    CancelRunningThreads();
    SetPathToRootDirty(Node);
    RunInThread(RecomputeRec(Root));
}

function RecomputeRec(TreeNode) {
    foreach (child of TreeNode) {
        if IsDirty(child) then
            RecomputeRec(child);
    }
    ComputeValues(TreeNode);
    SetClean(TreeNode);
}
```

During the recursive depth-first traversal of the tree, each completed node fires an update event for the registered views.

5.2 Algorithmic Optimizations

An important optimization came with the algorithm in the previous section: Only affected tree paths are recomputed because only the path from the updated node to the root is set dirty.

The fundamental DOI functions in the simple selections are computed by calculating the value of the trapezoidal function for each data value in the corresponding data dimension. A straightforward implementation would loop through all the values and calculate the DOI value for each of them. Since the trapezoidal function can be divided into 4 regions we can split up the algorithm as well. In a preprocessing step, we have to sort the data along each dimension v_k. This can be efficiently done with double-indexing via a permutation array p_k, so that $v_k[p_k[i]] \leq v_k[p_k[i+1]]$ is true for each dimension. Now we can perform binary search on each data dimension and find the indices of the bounds. Then we can construct the trapezoidal function with two linear interpolations and one constant period.

```
Brush(DOI, {lowerSoft, lower,
                upper, upperSoft}) {

    IndexLowerSoft = FindFirstGreaterThan(
        SortedData,lowerSoft);
    IndexLower = FindFirstGreaterThan(
        SortedData,lower);
    IndexUpper = FindLastSmallerThan(
        SortedData,upper);
    IndexUpperSoft = FindLastSmallerThan(
        SortedData,upperSoft);

    InterpolateZeroToOne(
        DOI,IndexLowerSoft,IndexLower-1);

    SetToOne(DOI,IndexLower,IndexUpper);

    InterpolateOneToZero(
        DOI,IndexUpper+1,IndexUpperSoft);
}
```

This yields a runtime of $4 * O(log_2(n))$ (versus $O(n)$ with the straightforward approach).

5.3 Low Level Programming Optimizations

DOI computation

Due to the fact that the DOI computation subsystem is based mainly on componentwise vector arithmetics, it was possible to highly optimize that part of the software by taking into account standard optimization techniques (loop unrolling, code inlining) as well as special features of the hardware the system was developed for. Our current production environment has the following technical details:

- OS: Windows XP Professional
- Intel Pentium 4, 2.8GHz
- 333 MHz FSB
- Hyperthreading support enabled
- Geforce FX 5900
- 2GB RAM
- 2x80GB S-ATA RAID0

Intel processors from the Pentium III upwards support a powerful SIMD instruction set (Streaming SIMD Extensions [5]) that can perform calculations on four 32 bit float values simultaneously. It was possible to heavily optimize the calculation of the DOI function based on this extension, since it packs 4 operations into one.

The Pentium IV supports the *hyperthreading* technology [14] that offers thread-level parallelism, enabling multi-threaded applications to make much better use of CPU resources than single-threaded applications. Therefore it was mandatory to implement the performance-critical parts of our software in a multi-threaded fashion. A quite obvious way how to do this was to partition the set of affected time steps per brushing operation in two parts and let two threads compute the DOI function in parallel.

3D rendering

Due to the large amount of points to be rendered for one time step, simple immediate mode rendering of GL_POINTS is too slow. Through the employment of *Vertex Arrays* it is possible to render mesh geometries with 1.000.000 geometry cells and more at interactive frame rates. We also used the *Vertex Buffer Objects (VBO)* extension [16] that provides fine grained control over the vertex array memory management and allows for tuning of the caching policies for the vertex/color/index data arrays according to their update frequencies.

Since we modulate the opacity of the rendered fragments with the DOI function, the hardware z-buffer cannot be used and rendering in back-to-front order is necessary to support hardware based alpha blending. The sorting is performed with an hybrid QuickSort/HeapSort algorithm that has running time $O(n \cdot log(n))$ in the worst case (IntroSort, see [10]). Although the sorting performance decreases when very large geometries are rendered, data sizes of up to 250.000 geometry cells can be rendered interactively ($\sim$ 15 frames/second) with this algorithm. An approach based on using an approximate ordering based on BucketSort (running time $O(n)$) during user interaction and exact sorting in idle periods has been considered but not yet been implemented.

6 Conclusions

We have developed a system for the visualization and analysis of high-dimensional and time-dependent data. It allows users to interactively explore data space and specify features with multiple views. We have shown how it is possible to handle large data sets in interactive applications on common PC hardware. Furthermore we have defined an extensible and scalable feature definition framework with a graphical as well as a text-based user interface. This generic approach allows a wide range of applications in the fields of scientific visualization (especially flow visualization) as well as information visualization.

We have already proven the high usability of our system in everyday life of researchers from the field of automotive engineering [4]. In the future, we hope that we can intensify our efforts in this direction as well as in other application areas, such as the visualization of rather abstract data describing stock exchange courses and other economic data.

7 Acknowledgements

This work has been carried out as part of the basic research on visualization at the VRVis Research Center in Vienna, Austria (http://www.VRVis.at/vis/), which is partly funded by an Austrian research program called KPlus. The author would like to thank Helmut Doleisch, Robert Kosara, and Helwig Hauser for supervision of this work.

References

[1] R. Becker and W. Cleveland. Brushing scatterplots. *Technometrics*, 29(2):127–142, 1987.

[2] H. Doleisch, M. Gasser, and H. Hauser. Interactive feature specification for focus+context visualization of complex simulation data. In *Proc. of the 5th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2003)*, pages 239–248, Grenoble, France, May 2003.

[3] H. Doleisch and H. Hauser. Smooth brushing for focus+context visualization of simulation data in 3D. In *Journal of WSCG*, volume 10, pages 147–154, Plzen, 2002.

[4] H. Doleisch, M. Mayer, M. Gasser, R. Wanker, and H. Hauser. Case study: Visual analysis of complex, time-dependent simulation results of a diesel exhaust system. In *Proc. of the 6th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2004)*, Konstanz, Germany, 2004.

[5] Intel Streaming SIMD Extensions. See URL: `http://x86.ddj.com/articles/sse_pt1/simd1.htm`.

[6] G. W. Furnas. Generalized fisheye views. In *Proc. of the ACM CHI '86 Conf. on Human Factors in Computing Systems*, pages 16–23, 1986.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] E. P. Klement, R. Mesiar, and E. Pap. *Triangular Norms*, volume 8 of *Trends in Logic*. Kluwer Academic Publishers, Dordrecht, 2000.

[9] R. Kosara, F. Bendix, and H. Hauser. Timehistograms for large, time-dependent data. In *Proc. of the 6th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2004)*, Konstanz, Germany, 2004.

[10] David R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993.

[11] F.H. Post, B. Vrolijk, H. Hauser, R.S. Laramee, and H. Doleisch. Feature extraction and visualization of flow fields. In *Eurographics State of the Art Reports*, pages 69–100, 2002.

[12] M. O. Ward. XmdvTool: Integrating multiple methods for visualizing multivariate data. In *Proc. of IEEE Visualization '94*, pages 326–336.

[13] The awt native interface. See URL: `http://java.sun.com/j2se/1.3/docs/guide/awt/AWT_Native_Interface.html`.

[14] Hyper-threading technology architecture and microarchitecture. See URL: `http://www.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf`.

[15] The java native interface. See URL: `http://java.sun.com/j2se/1.3/docs/guide/jni/`.

[16] Specification of arb_vertex_buffer_object. See URL: `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt`.

[17] Webpage of the world wide web consortium about xml. See URL: `http://www.w3.org/XML/`.