# Virtual-Reality Based Interactive Exploration of Multiresolution Data

Oliver Kreylos[1], E. Wes Bethel[2], Terry J. Ligocki[3], and Bernd Hamann[1]

[1] Center for Image Processing and Integrated Computing (CIPIC), Department of Computer Science, University of California, Davis, One Shields Avenue, Davis, CA 95616–8562, USA
[2] Applied Numerical Algorithms Group, National Energy Research and Scientific Computing Center (NERSC), Ernest Orlando Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA
[3] Visualization Group, National Energy Research and Scientific Computing Center (NERSC), Ernest Orlando Lawrence Berkeley National Laboratory, One Cyclotron Road, Berkeley, CA 94720, USA

**Summary.** We describe a system supporting the interactive exploration of three-dimensional scientific data sets in a virtual reality (VR) environment. This system aids a scientist in understanding a data set by interactively placing and manipulating visualization primitives, e. g., isosurfaces or streamlines, and thereby finding features in the data and understanding its overall structure.

We discuss how the requirement of interactivity influences the architecture of the visualization system, and how to adapt standard visualization techniques to work under real-time interaction constraints.

Though we have implemented our visualization system to work with multiple types of data sets structures – cartesian, tetrahedral, curvilinear-hexahedral and adaptive mesh refinement (AMR) – we will focus on AMR grids and show how their inherent multiresolution structure is useful for interactive visualization.

## 1 Introduction

Contemporary scientific research is performed by gathering and analyzing vast amounts of data. Regardless of whether this data is the result of real-world measurements or numerical simulations, extracting information from data becomes more and more difficult as the amount of data grows. Scientific visualization is becoming a major tool for research: The human visual system is unparalleled in its capacity to see patterns or detect features in data.

We concentrate on scalar- or vector-valued trivariate functions, i. e., functions of the type $f_s: \Omega \to \mathbf{R}$ or $f_v: \Omega \to \mathbf{R}^3$, where $\Omega \subset \mathbf{R}^3$ is some three-dimensional domain. We will also consider time-varying data sets, where the functions' domains are $\Omega \times [a, b]$, where $[a, b] \subset \mathbf{R}$ is some interval of time.

### 1.1 Visual Exploration

Typically, visualization is used for two different purposes: First, it can be used to gain understanding of phenomena underlying data; second, it can be

used to communicate this understanding to an audience. We concentrate on the first purpose, *visual exploration*. A scientist is typically confronted with a data set from an experiment or a simulation, and the task is to gain insight into the data. Often, especially when data is generated by a simulation, it is also necessary to show that the simulation system generated meaningful data at all; in this context, visual exploration becomes a debugging tool.

Visual exploration is most useful when applied early during data generation. Simulations generating large data sets typically run for long periods of time, even when using a supercomputer. If a visual exploration system can be used to visualize preliminary or evolving simulation results, it might be possible to detect errors in the simulation early, and to fix them by changing simulation parameters "on the fly." This interplay between visualization and simulation, also referred to as *computational steering*, can be very helpful in increasing the performance of a simulation system and its value for the overall scientific investigation process. To allow coupling visualization and simulation, the visualization system should be able to read the data generated by the simulation with as little pre-processing as possible. Since simulation systems create data in a variety of different formats (tetrahedral, cartesian, (curvilinear) hexahedral, AMR, etc.), the visualization system must support all these data structures directly, without having to resort to converting them to a standard format by re-sampling.

When working directly on the native data structure used by a simulation system, the debugging power of the visual exploration system is greatly increased. For example, an AMR simulation program creates the grid hierarchy without intervention [1], by considering certain properties of the data itself. If the visualization system can display the AMR data directly, it can also display the grid structure, providing the scientist with clues about the simulation process. We explain how to isolate the data set structure from the rest of the visualization program in Sec. 3.

### 1.2   Benefits of VR Methods

A standard visualization system that addresses both uses of visualization (exploration and communication) usually sacrifices interactivity for image quality. This fact can reduce the usefulness of a general-purpose system for visual exploration. For example, a data set might be the result of a numerical simulation of air flow around an aircraft wing, and the task is to determine whether the design fulfills the required aerodynamic objectives and how the wing design could be improved. In a standard visualization system, this task might be handled in the following way:

1. Place some visualization primitives, e. g., streamlines, at points considered "interesting."
2. Generate an image or multiple images.
3. If the imagery does not reveal anything exciting, repeat from step 1.

There are several problems with such an approach. First, a user has to specify at which points to place primitives. Specifying positions in 3D space using a 2D input device, e. g., a mouse, is awkward. Second, the process of image generation is time-consuming, and with placement of primitives essentially being a trial-and-error process, the overall time spent on analysis can grow prohibitively large. Even worse, merely seconds of delay between placing a primitive and seeing a resulting image can make intuitive exploration of a data set almost impossible. Third, even when an image reveals some features, it is only a 2D projection of a 3D data set and can be misleading concerning spatial relationships of features.
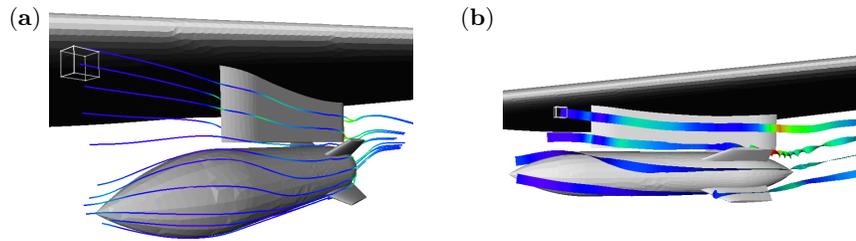
(a)                                          (b)

**Fig. 1.** Exploration of vector field on tetrahedral mesh with embedded geometry. (**a**) Visualization with streamlines (**b**) Visualization with streamribbons

VR can attack all three of these problems, making the exploration process much more productive. There are probably more contradicting definitions of VR in existence than there are VR researchers; but for the purposes of this paper, a system is considered a VR system if it offers at least the following functionalities:

- True 3D (stereoscopic) display with head tracking, i. e., support of a display that is dependent on a user's true eye position in space, updated at least at 30 Hz to create the illusion that the visualization is "real,"
- support of six-degree-of-freedom (6-DOF) input devices to allow a user to directly interact with and manipulate the visualization, and
- interactivity, meaning that the system's response time to user actions is small enough to seem immediate (within 0.05 s to 0.1 s).

With a VR system like this, the basic exploration loop remains the same; but with immediate response, a user is able to move a visualization primitive directly through a 3D data set until something interesting is visualized. The immediate update of displayed imagery, leading to an animation of a visualization primitive's behaviour, provides valuable clues as to where to move the primitive next in order to "close in" on a feature.

### 1.3   Real-time Constraints

The goal of implementing an interactive, real-time visualization system imposes several constraints on the visualization methods that can be supported. A severely limiting constraint is the *display constraint*, which requires that all imagery can be rendered at a frame rate of at least 30 Hz per eye. If the frame rate drops below 30 Hz, the display will appear "choppy," which can lead to eye strain and even motion sickness. Since users cannot hold their heads perfectly still, displayed stereo images have to be updated continuously, even if the visualization itself does not change. This forbids using very costly visualization methods, e. g., high-quality volume rendering [7]. Visualization methods of choice are the ones based on graphics primitives supported by available graphics hardware, e. g., isosurfaces, streamlines or streamribbons.

Even when considering only visualization methods supported by available graphics hardware, generating the visualizations can still be time-consuming. Though it is not necessary to update the visualization for every frame, should updating require more than 0.1 s, the system is "lagging," defeating the goal of interactivity. We refer to this constraint as the *update constraint*. This implies, for example, that a standard implementation of the Marching-Cubes (MC) isosurfacing algorithm [8–10] is not appropriate for visual exploration: Though the resulting isosurface triangulation might consist of a relatively small number of triangles, small enough to satisfy the display constraint, the time to generate the triangles might require seconds or minutes for large data sets, violating the update constraint. In Sec. 3.2, we explain how to adapt standard visualization methods to an interactive environment.

### 1.4   Related Work

Data visualization based on VR is not new. Several early important contributions were made by NASA scientists: The *NASA Virtual Wind Tunnel*, described, for example, by Bryson and Levit [5], is a visualization system for time-varying vector-valued data. Meyer and Globus [6] describe an interactive isosurface generator for scalar-valued data.

## 2   User Interface

User interfaces for virtual-reality environments are still an area of active research, and an interface paradigm agreed upon by most researchers has still to be developed. For our system, we implemented the simple gesture- and menu-driven user interface described in the following sections.

### 2.1   Available VR Devices

The design of the user interface is dependent on the underlying VR hardware, especially on the number of input devices and their degrees of freedom,

and the number and arrangement of buttons available. We implemented our system for two classes of VR hardware:

**Virtual Workbench** A virtual workbench is a semi-immersive VR system consisting of a large display screen, a pair of head-tracked shutter glasses, and three 6-DOF input devices: One *stylus* with a single button, and two *pinch gloves* with four buttons each, that can be activated by pinching the thumb and one of the other fingers.

**CAVE** A CAVE is a semi-immersive VR system consisting of one or more large display screens, a pair of head-tracked shutter glasses, and a single 6-DOF input device: A *wand* with three buttons and a pressure-sensitive joystick on it. CAVE systems come in different configurations, from a single-screen desk-like environment (ImmersaDesk) to a ten-by-ten-by-ten foot cube with five display screens as walls ("classic" CAVE). For the purposes of our system, we can treat all these identically, because they provide the same functionality in input devices, and the differences of the display hardware are hidden by a common application program interface, the *CAVE library*.

## 2.2   VR Toolkit

To simplify the task of writing VR applications, we implemented a VR toolkit to isolate the application program from the VR hardware and from basic interaction techniques. The toolkit's programming paradigm is similar to that of the Motif window environment toolkit: It supplies callback mechanisms, actions and simple widgets. The toolkit's architecture is shown in Fig. 2.
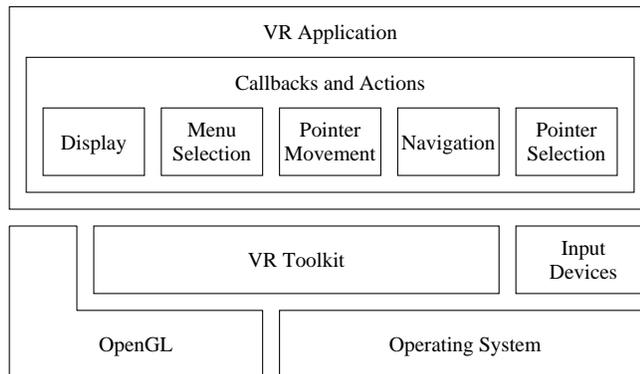
**Fig. 2.** Overview of the VR toolkit's architecture

Using this VR toolkit, we are able to port a VR application between several different classes of VR hardware without changing the application's source code. Currently, the VR toolkit is available for the two VR systems

listed above, and a version for desktop workstations is used for application testing purposes.

### 2.3   3D Pointer

The main selection and interaction is done with the *3D pointer*. This equivalent to a single-button mouse in a 2D window system is used to inquire data values at any point inside a data set, to place and drag new primitives, and to select existing ones. In the Virtual Workbench environment, we use the stylus as 3D mouse; in the CAVE environment, we use the wand and its leftmost button, the *mouse button*.

### 2.4   Navigation

In order to explore large data sets, a user must be able to freely navigate through the data sets. It must be possible to get an overview of the whole data set, and it must also be possible to zoom in to a small region of space. Furthermore, navigation must be as intuitive as possible to not interfere with the exploration process. We chose to implement navigation by *direct manipulation*: Instead of using indirect navigation tools, e. g., scrollbars in 2D window systems, a user can directly "grab" a point in space using a 6-DOF input device. Grabbing will fix the data set's coordinate system with respect to the input device's coordinate system, and any movement (translation or rotation) of the input device while a point is grabbed will move the whole data set accordingly, allowing panning and rotating.

In the Virtual Workbench environment, we employ pinch gloves for navigation. A point can be grabbed with either hand, by pinching thumb and index finger. This two-handed navigation suits both left- and right-handed users, and it also allows to navigate with one hand even while the other hand is interacting with the data set using the stylus. In the CAVE environment, the wand is switched into navigation mode using its middle button, the *navigation button*.

Though it is possible to grab any point in space, inexperienced users will intuitively only grab parts of displayed geometry, either existing visualization primitives or models embedded in the data set. Experiments have shown that this is not a serious limitiation; users get adjusted to the navigation paradigm quickly (in a matter of minutes) and are able to pan/rotate data sets according to their wishes.

While navigation involving panning and rotating is very intuitive, zooming is more difficult to implement. This is probably due to the fact that zooming is not possible in the real world: While every child learns how to pick up an object and move it around to inspect it from all sides from a very early age on, there is no natural way to enlarge an object. We implemented zooming in different ways, depending on the available input devices. In an environment with two input devices, a user can grab one point in space with each device.

Then, pulling the two devices apart will zoom in, and pushing them together will zoom out. We decided to make the zoom factor proportional to the two device's distance[1]; following this principle, the apparent motion of the data set will relate to the input devices' motion in a natural way. In a single input device environment, we reserved a special button combination on the input device for switching into zoom mode. A user can zoom in by pulling the input device towards her, and zoom out by pushing the input device away.

In the Virtual Workbench environment, both pinch gloves are used for zooming. First, the user will grab one point with one glove, enabling panning and rotating. Then the user will grab another point with the second glove, enabling zooming. The data set's coordinate system is still fixed to the first glove while zooming; this enables panning, rotating and zooming simultaneously. In the CAVE environment, zooming is activated while the wand is already in navigation mode. Pressing the mouse button while the navigation button is held stops panning and rotating and enables zooming. Luckily, the two buttons are close enough to allow even inexperienced users to press both at the same time using one thumb. Still, the restriction to only one input device limits the usability of the system. Experiments have shown that the ambidexterous Virtual Workbench is more intuitive and makes it easier to navigate to a desired viewpoint.

Even in an environment with two devices, experiments show that inexperienced users will hardly use the zoom feature. Instead, they tend to pull a part of the data set they want to examine closer to their eyes, making it appear larger. This is a natural reaction, but it clashes with the limitations of the graphics system. Due to the stereo rendering, objects close to the user's eyes will be out of focus, and due to inaccuracies of head tracking, objects will not appear to be stable but move erratically. Correctly using the zoom feature to inspect an area more closely requires some training.

## 2.5   Higher-level Functions

To access higher-level program functions, e. g., selecting tools or changing visualization parameters, we provide a simple system of cascading popup menus. The main menu can be popped up at any position and orientation in space. While it is displayed, the 3D pointer is used to select menu entries or pop up submenus.

In the Virtual Workbench environment, the main menu is popped up by pinching thumb and middle finger of one hand. While the menu is displayed, it will move with the hand, allowing the user to move it to a position where it is convenient to select entries using the stylus and the other hand. In the CAVE environment, the rightmost wand button, the *menu button*, will pop up the main menu at the current wand position. As soon as the menu is

---

[1] $z = \|p_1 - p_2\|/d_0$, where $p_i$ are the device's positions, and $d_0$ is their initial distance at the time grabbing occured.

displayed, the wand is used to activate entries or popup submenus. Releasing the menu button selects the currently active menu item.

## 3   System Architecture

In order to fulfill the requirements of simulation debugging and computational steering, the visual exploration system has to be able to work as closely as possible with the applications generating the data it has to visualize. This means that the system must support a wide variety of data set structures. On the other hand, the system has to support a wide variety of different visualization primitives to address different needs. To make the system robust under changes or additions to both parts, it is necessary to isolate the generation of visualization primitives from the underlying data set structure, and to allow the two modules only to communicate through two well-defined interfaces. An overview of the complete system's architecture is shown in Fig. 3; the following sections describe the modules in detail.
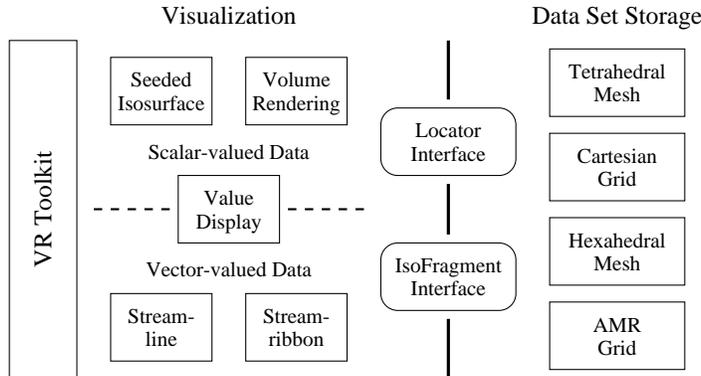
**Fig. 3.** Overview of the visual exploration system's architecture

### 3.1   Interfaces

The following sections describe the two interfaces connecting the generation of visualization primitives and the underlying data set structure.

### The Locator Interface

The most basic function of any visualization system is to evaluate the function represented by a given data set at any point inside its domain. For most types of data sets, this involves first locating a cell that contains the query point,

and then applying some interpolation scheme to calculate the value at the query point's position. To allow maximal efficiency, we defined the Locator interface to contain both functions separately:

- `bool locatePoint(p, bool traceHint)` prepares a locator to interpolate the data set's value at position `p`. The function's return value indicates whether the given point `p` is inside the data set's domain. If the return value is `false`, it is not possible to determine the data set's value. For some data set structures, locating a point "from scratch" might be very expensive; it might become much less difficult when a locator can use the fact that the new point it has to locate is close to the last located point. The parameter `traceHint` is a hint, provided by the visualization system, that the given point is close to the last point, and that using this information might be beneficial. Several typical visualization primitives, e.g., streamlines, evaluate a data set at a long sequence of points being very close to each other. In some cases, using the `traceHint` parameter resulted in speed-ups of an order of magnitude.
- `ValueType interpolateValue(void)` calculates the data set's value at the last located position.

With these two functions, iterators serve a dual purpose: They are not only used to evaluate a data set, but also to identify positions in a data set that can be passed between data structures and algorithms. In this way, they resemble the iterator mechanism used in the C++ Standard Template Library. The two steps involved in evaluating a data set at an arbitrary point are illustrated in Fig. 4.
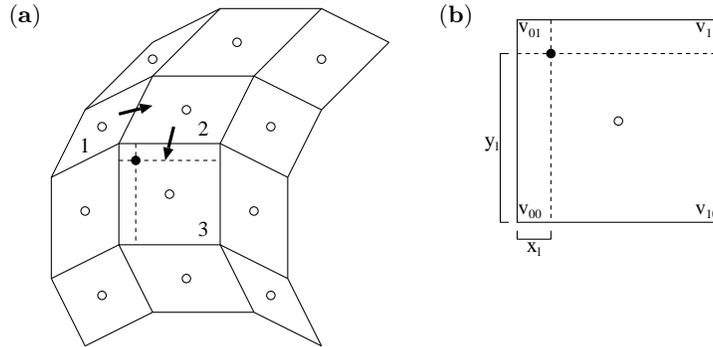


**Fig. 4.** Steps involved in evaluating a data set. (**a**) Locating cell containing query point (solid dot) and determining its local coordinates. Cell 1 is tried first because its centroid (hollow dots) is closest to the query point. The algorithm then crosses into cells 2 and finally 3 (**b**) Bilinear interpolation used to evaluate the cell at local coordinates $(x_l, y_l)$

**The IsoFragment Interface**

The IsoFragment interface has a more specific purpose than the Locator interface: It is used to generate seeded isosurfaces, as will be explained in Sec. 3.2. An IsoFragment identifies a cell in a data set, and it provides the following two functions:

- `setCell(Locator loc)` associates an IsoFragment with the data set cell containing the point most recently located by the Locator. This function is used to seed an isosurface.
- `expandSurface(ValueType isovalue, Queue expandNext)` creates the fragment of the surface connecting all points with value `isovalue` inside the associated cell, and puts all the cell's neighbours sharing faces that are intersected by the isosurface in the expansion queue.

## 3.2   Supported Visualization Primitives

The choices of visualization primitives for visual exploration systems are limited by the two constraints discussed in Sec. 1.3: The graphics system must be able to display all active primitives at a framerate of at least 30 Hz per eye (display constraint), and the visualization system must be able to update all primitives the user is interacting with at a rate of at least 10 Hz (update constraint). Some common primitives, e. g., streamlines, are almost directly applicable to visual exploration, whereas others, e. g., isosurfaces, have to be adapted to be usable.

**Data Value Display**

The most basic tool, hardly a visualization primitive, is to display the data value at the current 3D pointer location. It can be applied to both scalar- and vector-valued data sets. The only operation required by this tool is calculation of a function value for an arbitrary point, given in the data set's coordinate system. This operation is directly supported by the Locator interface described in Sec. 3.1. Since the 3D pointer is traced by the system at a rate of 30 Hz, it usually only moves a short distance between two point location requests. Communicating this fact to the Locator interface using the `traceHint` parameter mentioned in Sec. 3.1 can increase system performance, depending on the underlying data set structure.

**Primitives for Scalar-valued Data Sets**

The following primitives apply only to scalar-valued data sets, i. e., those defining functions $f: \Omega \to \mathbf{R}$.

**Seeded Isosurfaces** The most often used visualization primitive for scalar-valued data is the isosurface, connecting all points in space having identical function value [8]: For a scalar-valued function $f\colon \Omega \subset \mathbf{R}^3 \to \mathbf{R}$, the isosurface $I(v)$ for isovalue $v$ is defined by $I(v) = \{\, x \in \Omega \mid f(x) = v \,\}$. In standard systems, isosurfaces are typically generated using some variation of the Marching-Cubes algorithm [9]. In this algorithm, isosurface fragments are generated independently for each cell of the data set. This algorithm has a runtime proportional to the number of cells, and generally runs several seconds to minutes for large data sets.

To use isosurfaces in an interactive visualization system, we have to change their generation to satisfy both display and update constraint, see Sec. 1.3. The basic idea is to generate isosurfaces incrementally starting from a given point in space, instead of globally starting from a given function value. In our system, a user can move the 3D pointer to an arbitrary position in space and *seed* an isosurface there. A seeded isosurface is generated using the following main steps:

1. Determine the cell containing the 3D pointer and calculate the function value at the pointer's position. Store the found value as the isovalue to use, and put the found cell in the *expansion queue.*
2. Take the next cell from the expansion queue. Create an *isosurface fragment* for the stored isovalue inside the cell, and determine the cell faces intersected by the isosurface fragment. Put all cells sharing those faces that have not yet been visited back in the expansion queue. An illustration of this process is shown in Fig. 5.
3. While there are cells in the expansion queue, repeat from step 2.

This algorithm will generate an isosurface for the function value at the 3D pointer's position. It can be used in an interactive system, because the expansion loop (step 2 in the algorithm) can be stopped at any time. To satisfy the update constraint, the system will start a timer before starting expansion, and will stop expanding as soon as the timer expires. To satisfy the display constraint, the loop will be stopped as soon as a certain amount of geometry has been generated. This amount depends on the underlying graphics system's performance; typical values are between 100,000 and 200,000 triangles. The described algorithm needs both interfaces described in Sec. 3.1. The Locator interface is needed to calculate the isovalue at the 3D pointer's position, and the two main operations in the algorithm, surface fragment generation and queueing of neighbouring cells, are provided by the IsoFragment interface.

The benefit of seeded isosurfaces is that they can always be generated at interactive rates, independently of data set size and performance of the underlying hardware. They also scale directly with hardware performance; on a faster CPU, the program will automatically generate larger isosurface parts. Furthermore, we found out that animation of isosurfaces, by moving
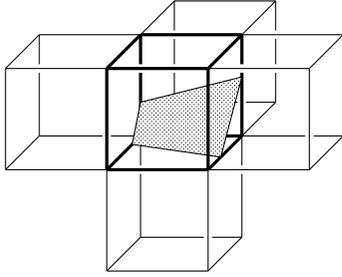
**Fig. 5.** Expanding a seeded isosurface. The IsoFragment interface generates a fragment of an isosurface inside a cell, and puts all neighbouring cells the isosurface continues into in an expansion queue

the 3D pointer while continuosly seeding, enables a user to quickly gain understanding of the behaviour of the data set in a region of space.

Their main drawback is, that in most cases only a part of the complete isosurface for a given isovalue is generated. This prohibits the user from getting a "big picture" overview of a complete data set. To offset this drawback, we decided to continue growing seeded isosurfaces up to the limits set by the display constraint once the user stops moving the 3D pointer. In this way, larger portions of an isosurface can be created. Figure 6 shows a seeded isosurface during animation and after the user stopped animation.
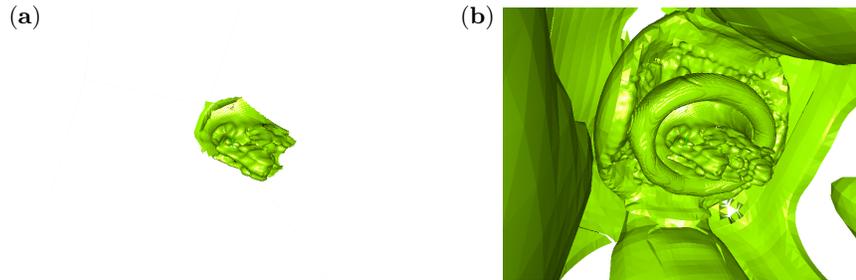
(**a**)                                                    (**b**)



**Fig. 6.** A seeded isosurface. (**a**) Isosurface while being animated (**b**) Isosurface after it has been released

### Primitives for Vector-valued Data Sets

The following primitives apply only to vector-valued data sets, i.e., those defining functions $f \colon \Omega \to \mathbf{R}^3$.

**Streamlines** Streamlines are probably the most commonly used visualization primitive for vector-valued functions. By definition, a streamline for a vector-valued function $f \colon \Omega \subset \mathbf{R}^3 \to \mathbf{R}^3$ is an integral curve $p \colon [t_0, t_1] \to \Omega$ defined by an initial value problem $p(t_0) = x_0$, $\dot{p}(t) = f\big(p(t)\big)$. A streamline directly visualizes the local vector directions of a vector-valued function. Fig. 1 (a) shows a visualization of a tetrahedral vector field using streamlines.

In our system, a user can create a streamline by selecting a starting point $x_0 \in \Omega$ inside the data set's domain. The points defining the streamline are then generated from the initial value problem using an adaptive step-size fourth-order Runge–Kutta method. This generation algorithm satisfies both real-time constraints; streamlines are generated one point at a time, and the generation can be interrupted when a computation timer runs out. As all interactive visualization primitives, the user can animate a streamline by moving the 3D pointer while the streamline is continuously regenerated. We found out that this animation is very helpful in understanding the behaviour of the function represented by a data set. By observing a streamline's "reaction" to moving the start point, a user can intuitively "home in" to critical points, and can get an understanding of the function's vector field topology [3].

When using a Runge–Kutta method to generate streamlines, the only functionality needed is to evaluate the given data set at a sequence of positions inside its domain. This functionality is provided by the Locator interface. We observe that subsequent evaluation positions in a Runge–Kutta computation are very close to each other. Thus, to maximize performance,our system uses the `traceHint` parameter to the `locatePoint` function to communicate this fact to the data set storage.

**Streamribbons** Streamlines as described above only visualize the local vector directions of a vector-valued functions. Often it is desirable to also directly visualize derived properties of a vector-valued function. Streamribbons are a generalization of streamlines that directly visualize the local vector direction and the local vorticity of a vector-valued function [4]. Instead of being a single curve, a streamribbon is rendered as a thin strip, whose rotation around its longitudinal axis is proportional to the dot product of the visualized function's local vorticity and the streamribbon's direction. Fig. 1 (b) shows a visualization of a tetrahedral vector field using streamribbons.

Streamribbons are generated similarly to streamlines: A fourth-order Runge–Kutta method is used to iteratively solve the defining initial value problem, and the data set's vorticity is evaluated directly through the Locator interface. As for streamlines, the computation can be interrupted at any time to satisfy the real-time constraints.

### 3.3   Data Set Structures

In this section, we describe the different data set structures implemented in our system, and how the two interfaces are implemented for each.

### Cartesian Grids

For our purposes, cartesian grids are the simplest data set structure. A cartesian grid consists of a three-dimensional rectangular arrangement of $i \times j \times k$ rectangular hexahedral cells, where each cell is of identical size $s_x \times s_y \times s_z$.

Locating a point inside a cartesian grid merely involves multiplication and modulo division operations, and interpolating the function value at a point inside a cell is done using trilinear interpolation.

To generate an isosurface fragment inside a cell, we use a standard Marching-Cubes case table; to enumerate all intersected neighbours of a cell, we use the implicit neighbourhood relation imposed by the grid structure.

### Tetrahedral Meshes

In tetrahedral meshes, locating a point involves finding the tetrahedron containing it, and calculating the point's barycentric coordinates inside that tetrahedron. To locate a point, we use the following two-stage approach:

1. Find the tetrahedron whose centroid is closest to the point in question. We implemented this by computing all cell centroids upon loading a data set, and storing them in an octree to later retrieve them using a closest-point query. If the `traceHint` parameter is set, skip this step and use the tetrahedron containing the last located point.
2. Determine the barycentric coordinates of the query point with respect to the current tetrahedron. If the query point is not inside the tetrahedron, i.e., at least one barycentric coordinate is negative, move to the neighbour sharing the face whose barycentric coordinate is smallest, and repeat this step. Neighbourhood relations between tetrahedra are stored explicitly in the data set, see Fig. 7; thus, finding the tetrahedron sharing a given face is an $O(1)$ operation.

This algorithm, illustrated in Fig. 8, works very well in practice. From-scratch point locations take on the order of $O(\log n)$ steps for $n$ tetrahedra to locate an initial tetrahedron, but they occur rather infrequently. Tracing a point from the initial tetrahedron to the final one containing it usually requires no more than two or three steps, which is sufficiently fast to generate streamlines or streamribbons interactively. Once a point is located, interpolation is done as a convex combination of vertex values using the query point's barycentric coordinates.
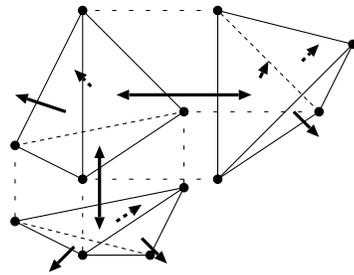


**Fig. 7.** "Exploded" view of a tetrahedral mesh. Each tetrahedron stores pointers to its four vertices and pointers to the up to four tetrahedra sharing its faces. This allows crossing of tetrahedron faces in $O(1)$ time during point tracing.
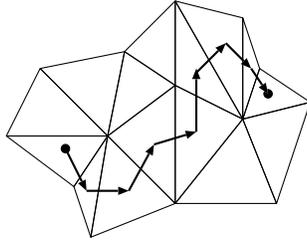
**Fig. 8.** Tracing a point in a tetrahedral mesh. If the point is not contained in the current tetrahedron, the face associated with the smallest (negative) barycentric weight is crossed

To generate isosurface fragments, we use a standard marching tetrahedron case table. Finding neighbours just involves traversing the explicit neighbour pointers stored with each tetrahedron.

### Curvilinear Hexahedral Meshes

Point location in curvilinear hexahedral meshes works very similarly to point location in tetrahedral meshes. We store cell centroids in an octree to quickly locate cells "close" to a point; then we perform the tracing loop to find the final cell containing the query point. Due to the curvilinear cells' almost-cubic structure, tracing usually takes even less steps than for tetrahedral meshes. Similarly to tetrahedral meshes, we store the neighbourhood information explicitly for each cell. This allows $O(1)$ traversal between neighbouring cells, and also allows "stitching" one or more curvilinear meshes together to form a larger one without having to treat traversal between different grids as a special case. An example where stitching is necessary is a fusion plasma data set, shown in Fig. 9, which closes around its $x$ and $z$ axes to form a hollow torus.

The difficult part in point location is calculating a point's local coordinates with respect to a cell. The transformation from cell coordinates to world coordinates is defined by trilinear interpolation of a cell's vertex positions; to transform the other way, the inversion of trilinear interpolation has to be calculated.

Let a hexahedral cell be defined by its eight vertices $\boldsymbol{v}_0, \ldots, \boldsymbol{v}_7 \in \Omega \subset \mathbf{R}^3$. Then trilinear interpolation converts a point $\boldsymbol{p} = (x, y, z)^\mathrm{T} \in [0, 1]^3$ in local cell coordinates to a point

$$
\begin{aligned}
\boldsymbol{p}_w = T(\boldsymbol{p}) = \quad & \boldsymbol{v}_0 \cdot (1-x)\,(1-y)\,(1-z) + \boldsymbol{v}_1 \cdot x\,(1-y)\,(1-z) \\
+\ & \boldsymbol{v}_2 \cdot (1-x)\quad y\quad (1-z) + \boldsymbol{v}_3 \cdot x \quad y \quad (1-z) \\
+\ & \boldsymbol{v}_4 \cdot (1-x)\,(1-y)\quad z\quad + \boldsymbol{v}_5 \cdot x\,(1-y)\quad z \\
+\ & \boldsymbol{v}_6 \cdot (1-x)\quad y \quad\ z\quad + \boldsymbol{v}_7 \cdot x \quad y \quad\ z
\end{aligned}
$$

To invert this transformation, we rewrite the vector equation $\boldsymbol{p}_w = T(\boldsymbol{p})$ as $T(\boldsymbol{p}) - \boldsymbol{p}_w = \boldsymbol{0}$ and solve for the unknown $\boldsymbol{p}$ using Newton–Raphson iteration [11]. The derivative of $T(\boldsymbol{p})$ needed for the Newton–Raphson method
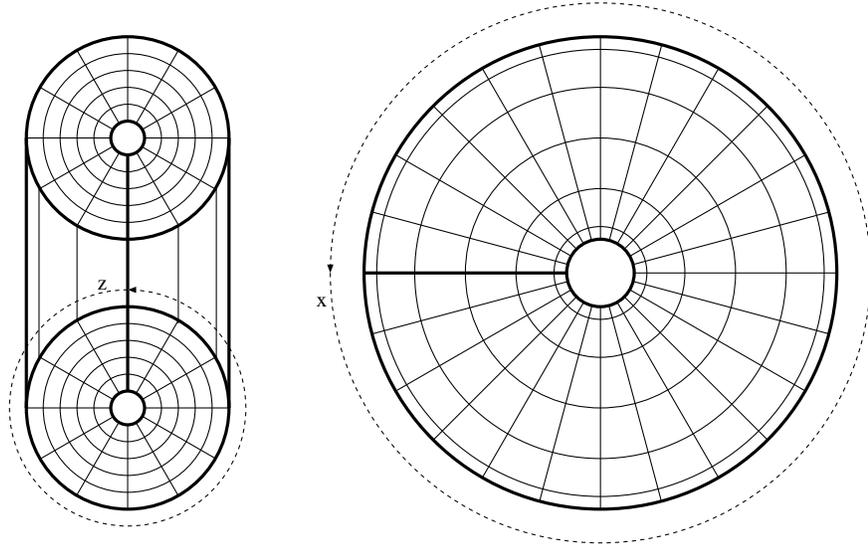
**Fig. 9.** A curvilinear hexahedral grid forming a hollow torus. Left: A crossection along the main torus axis; right: A view along the main torus axis.

is given by the $3 \times 3$-matrix $D(\boldsymbol{p}) := (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}) \cdot T(\boldsymbol{p})$, given by

$$\frac{\partial}{\partial x}T(\boldsymbol{p}) = \begin{array}{l} (\boldsymbol{v}_1 - \boldsymbol{v}_0) \cdot (1 - y)\,(1 - z) + (\boldsymbol{v}_3 - \boldsymbol{v}_2) \cdot y\,(1 - z) \\ + (\boldsymbol{v}_5 - \boldsymbol{v}_4) \cdot (1 - y)\quad z \quad + (\boldsymbol{v}_7 - \boldsymbol{v}_6) \cdot y \quad z \end{array}$$

$$\frac{\partial}{\partial y}T(\boldsymbol{p}) = \begin{array}{l} (\boldsymbol{v}_2 - \boldsymbol{v}_0) \cdot (1 - x)\,(1 - z) + (\boldsymbol{v}_3 - \boldsymbol{v}_1) \cdot x\,(1 - z) \\ + (\boldsymbol{v}_6 - \boldsymbol{v}_4) \cdot (1 - x)\quad z \quad + (\boldsymbol{v}_7 - \boldsymbol{v}_5) \cdot x \quad z \end{array}$$

$$\frac{\partial}{\partial z}T(\boldsymbol{p}) = \begin{array}{l} (\boldsymbol{v}_4 - \boldsymbol{v}_0) \cdot (1 - x)\,(1 - y) + (\boldsymbol{v}_5 - \boldsymbol{v}_1) \cdot x\,(1 - y) \\ + (\boldsymbol{v}_6 - \boldsymbol{v}_2) \cdot (1 - x)\quad y \quad + (\boldsymbol{v}_7 - \boldsymbol{v}_3) \cdot x \quad y \end{array}$$

Using $T(\boldsymbol{p})$ and $D(\boldsymbol{p})$, the iteration step can now be written as $\boldsymbol{p}_{i+1} := \boldsymbol{p}_i - D(\boldsymbol{p})^{-1}T(\boldsymbol{p})$. As long as the cells are not too oddly shaped, the iteration will converge to a solution after only a few iterations.

Generating isosurface fragments works exactly as for cartesian grids. We use a Marching-Cubes case table to generate fragments, and the explicitly stored neighbourhood information to find neighbouring cells.

## AMR Grids

AMR grids [1] are difficult to handle for our visualization system due to their complicated structure, but they offer benefits for interactive visualization due to their inherent multiresolution structure. The AMR grids supported by our system consist of several *levels*, where each level consists of several cartesian

grids of identical cell sizes. The grids inside each level are non-overlapping, and the union of their domains forms the domain of the level. The domain of the whole data set is the domain of the first grid level. Let $L_i$ and $L_{i+1}$ be two adjacent levels; then the cell sizes of all grids in $L_i$ must be an integer multiple of the cell sizes of all grids in $L_{i+1}$. Furthermore, each cell of each grid in $L_{i+1}$ must be completely contained in a cell of some grid of $L_i$. This implies that the domain of level $L_{i+1}$ is a subset of the domain of level $L_i$. Figure 10 shows an illustration of an AMR data set's grid structure.
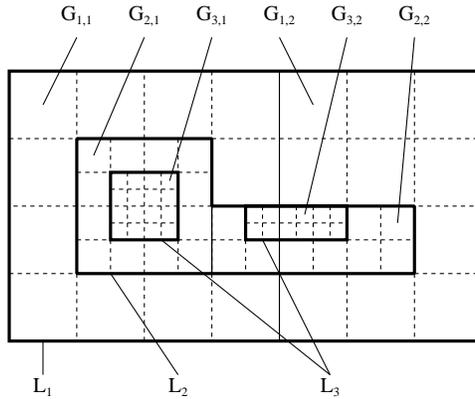


**Fig. 10.** Structure of an AMR grid with three levels $L_1$, $L_2$ and $L_3$. Each level $L_i$ consists of two grids $G_{i,1}$ and $G_{i,2}$

These restrictions stem from the AMR simulation algorithm: After each iteration step, those cells that have to be subdivided in order to guarantee certain error limits are subdivided independently, and all the subdivided cells are clustered together to form a set of cartesian grids.

The problem in locating points in AMR grids is to find the smallest cell in the hierarchy containing the query point. We implemented this by overlaying the AMR grid's domain with an octree structure. Octree leaves are restricted to only contain cells from a single grid of the most refined grid level whose domain intersects the octree leaves. Octree nodes not satisfying this property are subdivided until their children do, see Fig. 11. From this construction, we know that every point inside an octree leaf must be located in the level and grid associated with the leaf, and that it cannot be overlayed by a grid on a more refined level. Thus, locating a point inside an octree leaf is reduced to location in cartesian grids. When the `tracingHint` parameter to the `locatePoint` function is `true`, the program first checks if the new point position is inside the same octree leaf as the old one. In that case, the point is relocated using the cartesian grid algorithm. Otherwise, the program will traverse the octree structure upwards from the leaf to find an interior node containing the new position, and then downward again to find a leaf

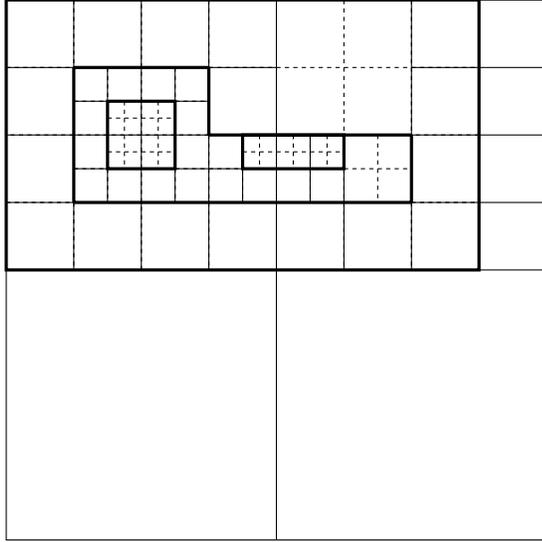containing it. Inside this leaf, the point is finally located using the cartesian grid algorithm again.



**Fig. 11.** AMR grid with overlayed octree structure used to locate points. Octree leaves only contain complete cells from a single grid of the most refined level whose domain intersects the octree leaf

Creating isosurfaces inside AMR grids is even more complicated. Generating the isosurface fragment itself is done by using a Marching-Cubes case table, since all finest-level cells are cartesian grid cells. The problem arises when cell neighbours have to be enumerated. As long as a cell is not touching the border of the octree leaf containing it, its neighbours are determined as for cartesian grids. Otherwise, the octree is traversed upwards and then downwards again to find all cells sharing the originating cell's traversed face. Depending on the local refinement, the possible configurations can be one neighbour of identical or larger size, or several smaller neighbours. Both cases are handled by the recursive octree traversal mechanism.

The major problem with this approach is that isosurfaces can exhibit "cracks" when they cross boundaries between differently refined grids. This problem might be fixed in the future by using the grid structures and algorithms described in [2]

**Multiresolution Visualization** AMR grids in the form described here are inherently a multiresolution representation of their underlying functions. Since grid cells that are overlayed by grids in more refined levels still have meaningful values, usually generated by some subsampling technique, one can visualize a lower-resolution version of an AMR grid by just ignoring several of the more refined levels. This can be done on-the-fly by tagging each octree node with the level of the grid they are contained in. This way,

interior nodes can be treated like leaves when their grid level is at least as refined as the maximum level that should be considered at the selected level of resolution. The other parts of the AMR grid code do not have to be changed to accomodate multiresolution visualization in this way.

## 4     Conclusions and Future Work

We presented an interactive VR visual exploration system that can be used by scientists to explore large data sets of different structures. It is specifically designed to work closely with the simulation systems generating the visualized data, to allow exploration of preliminary or evolving data and to enable "computational steering." We discussed how the requirement of interactivity has influenced the architecture of our system and the choice of supported visualization primitives and their implementation. We described how to isolate the two major system components, visualization and data set storage, from each other by introducing small, well-defined interfaces, and how these interfaces are implemented for the supported data set structures. We concentrated on visualizing AMR data, and on how to exploit the inherent multiresolution structure of AMR grids for interactive visualization.

The main areas for future work are increasing the range of visualization tools available, especially by adding localized volume rendering, implementing a crack-free isosurface algorithm for AMR grids, and implementing a feedback mechanism that allows our system to be used for computational steering, by visualizing "live" data from a running simulation and allowing to change simulation parameters from within the visualization program.

## 5     Acknowledgments

## References

1. Berger, M., and Colella, P., *Local Adaptive Mesh Refinement for Shock Hydrodynamics*, in: Journal of Computational Physics, 82:64–84, May 1989. Lawrence Livermore Laboratory Report No. UCRL–97196
2. Weber, G. H., Kreylos, O., Ligocki, T. J., Shalf J. M., Hagen H., Hamann, B., and Joy, K. I., *Extraction of Crack-Free Isosurfaces from Adaptive Mesh Refinement Data*, Proceedings of the Joint EUROGRAPHICS and IEEE TCVG Symposium on Visualization, Ascona, Switzerland, May 28–31, 2001, Springer Verlag, Wien, Austria, May 2001
3. Helman, J. L., and Hesselink, L., *Representation and Display of Vector Field Topology in Fluid Flow Data Sets*, in: Computer 22(8) (1989), pp. 27–36
4. Ueng, S.-K., Sikorski, C., and Ma, K.-L., *Efficient Streamline, Streamribbon, and Streamtube Constructions on Unstructured Grids*, in: IEEE Transactions on Visualization and Computer Graphics 2(2) (1996), pp. 100–110
5. Bryson, S. and Levit, C., *The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows*, in: Proc. of Visualization '91 (1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 17–24
6. Meyer, T. and Globus, A., *Direct Manipulation of Isosurfaces and Cutting Planes in Virtual Environments*, technical report CS–93–54 (1993), Brown University, Providence, RI
7. Drebin, R. A., Carpenter, L. and Hanrahan, P., *Volume Rendering*, in: Proc. SIGGRAPH '88 (1988), pp. 65–74
8. Bloomenthal, J., *Polygonization of Implicit Surfaces*, in: Computer Aided Geometric Design 5(4) (1988), pp. 341–356
9. Lorensen, W. E. and Cline, H. E., *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, in: Proc. of SIGGRAPH '87 (1987), pp. 163–169
10. Nielson, G. M., and Hamann, B., *The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes*, in: Proc. of Visualization '91, (1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 83–91
11. Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C*, 2nd ed. (1992), Cambridge University Press, Cambridge, MA