Leopold-Franzens University Innsbruck
Institute of Computer Science

Computing Geodesics in Numerical Space Times

De computatione distantiarum brevissimarum in spatiis quattuor dimensionum numeris descriptis

Master Thesis

Marcel Ritter

Supervisor: O. Univ. Prof. Dr. Sabine Schindler

March 30, 2010

✝

In memoriam:

O. Univ. Prof. i.R. Dr. sub ausp. Manfred Ritter

Who never lost his curiosity, fascination and deep passion for science and
teaching even during difficult circumstances.
Although he was unable to support my work in this world I know his
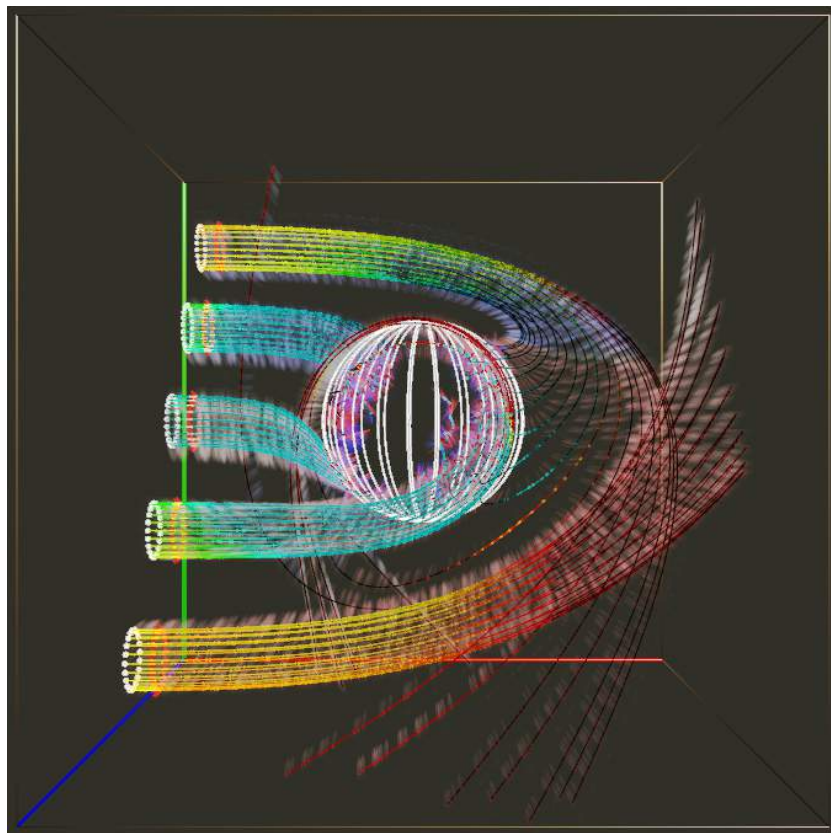guiding spirit was always by my side.

Figure 1: Geodesics in a Kerr metric.

Abstract:

Traditionally, tools to visualize geodesics in curved spacetimes of general relativity have been specialized solutions, either tailored to a certain spacetime or limited to certain kinds of numerical data. Utilizing a *Fiber Bundle* data model and the *Vish* visualization environment, this thesis aims to solve this problem by developing an approach that is independent of the underlying numerical data. My approach allows the combination of several visualization modules, and opens the possibility of applying the computation and visualization of integral lines more readily to other scientific domains. These domains include computational fluid dynamics and medical imaging.

Contents

Chapter 1

Introduction

The objective of this thesis is to provide a visualization framework suitable for exploring numerical spacetimes originating from astrophysical numerical relativity.

Numerical relativity is a very active research area. Having immense computing power available, numerical methods allow to solve the Einstein field equations, *chapter 2.2.1*, as is was not possible before. One specific application is the detection of gravitational waves. Gravitational waves have not yet been directly measured, but a Nobel Prize was given to Hulse and Taylor, [44], for finding a convincing indirect evidence. They measured timings in a binary pulsar. The observed frequency increase can only be explained by energy loss due to the emission of gravitational waves. Thus, the waves themselves have not yet been measured, but their effect on the emitting systems has been observed. Gravitational waves are ripples in spacetime curvature propagating with the speed of light. "Any mass in nonspherical, nonrectilinear motion produces gravitational waves (...), but gravitational waves are produced most copiously in events such as the coalescence of two compact stars, the merger of massive black holes, or the big bang." [36]

Detectors for gravitational waves have been built in the United States in Livingston, Louisiana, in Hanford, Washington [39], in Europe in Hannover, Germany [29], and near Pisa, Italy [24]. In order to detect a gravitational wave, relative distance changes in the order of 10^{-22} have to be measured.

Numerical analysis of situations involving strong gravity, such as the merging of black holes is of great importance. These simulations can be used to match the actually measured data of the gravitational wave detectors. Interactive $3D$ visualization techniques based purely on numerical data helps to analyze these simulations. This thesis focuses on the visualization of geodesics, the shortest (or longest) path between points in space (or spacetime). They are important indicators of the structure of spacetime.

Geodesics in curved spacetimes have been studied before, but most work is related to the visualization of analytic spacetimes, 1992 [2], 1997 [26], 1999 [22], 2001 [15] and 2004 [28]. Computing geodesics is required for ray-tracing black holes. In 1991 Corvin Zahn at the University of Tübingen implemented four dimensional ray-tracing in an analytic Schwarzschild space time, [63]. The most similar work was done already in 1992 by Steve Bryson, who implemented geodesic visualizations for exploration using a "boom mounted six degree of freedom head position sensitive stereo CRT system", [17]. For his setup the curved spacetimes could be given by closed formulas or also on simple uniform grids.

Geodesics were also analyzed in numerical spacetimes in the 2D (axisymmetric) era. The famous "pair of pants" picture contains the event horizon in a head-on collision, along with some geodesics. The corresponding movies were made in 1995 with great effort in TV resolution, see e.g. [43]. Werner Benger at the University of Innsbruck simulated a black hole by raytracing in 1996 [4]. Andrew Hamilton implemented a real time flight simulator for a charged black hole. He uses a projective technique to compute the paths of geodesics, [35].

Spacetimes visualized in this thesis are sampled on uniform grids. However, the developed infrastructure used in this work easily extends to adaptive mesh refinement (AMR) grids, which are currently used for numerical simulations of merging and colliding black holes, without changing the computation and visualization algorithms. Moreover, the rudimentary visualization of geodesics can be enhanced with other visualization modules, for instance to show the coordinate-acceleration, *equation (2.53)*, on the geodesics.

Inspired by Bryson's seeding methods for geodesics I developed a flexible technique for the creation of seeding geometries using basic operating blocks based on the theory of fiber bundles, that can be combined to a huge variety of seeding geometries.

In addition to the visualization and computation, other aspects have been addressed in this work:

- the data model

- high code re-usability

- high modularity

- provide an introduction to utilized software environments and libraries

The modern scientific world uses many computational methods in different scientific domains. The requirement of collaborations increases and thus

exchanging data becomes important. During my research visit at Louisiana State University I was taught that in 2005 hurricane Katrina forced scientific groups to exchange their data sets and couple different kinds of numeric simulations to predict the path of Katrina to provide warning and rescue to the public. For example, actual satellite data was coupled with simulation of the motion of air of the hurricane, which was then coupled with a simulation for wave propagation on the ocean. Such coupling is only possible if data can be exchanged without time-consuming data conversion processes between the different scientific groups. The simulation results were later gathered in a visualization combining several layers stemming from all the different domains, [14].

The thesis starts with a short theoretical part providing the necessary mathematical and physical background of general relativity, computational fluid dynamics and magnetic resonance imaging needed for the visualization and interpretation of the resulting illustrations, *chapter 2*.

Thereafter, some C++ programming techniques used for implementation are presented in *chapter 3*. Template meta programming is covered for enhancing overall source code re-usability.

Chapter 4 introduces and describes the concepts of the data model used throughout this thesis. Utilizing this data model made it possible to easily apply the developed algorithms to other applications such as medical imaging. It also enhanced source code re-usability because algorithms could be reused in situations that had not been possible without such a strong concept at the software foundation level.

Chapter 5 introduces the visualization environment that was used to implement the algorithms. It consists of a very flexible and modular system of software components that can be easily extended. During the implementation several parts of the software environments were refined and additional features where added to the software kernel.

The core work of the thesis is described in *chapter 6*. Concepts and algorithms for computing geodesics and streamlines are presented there and finally verified and utilized in *chapter 7*. The initial objective of computing geodesics was widened to the computation of integral lines. A simpler application scenario was chosen to develop the basic software infrastructure, which was later extended to the final application. Especially during the work on the foundations, it was possible to apply some of the new techniques in collaborative work that led to publications, *appendix C*.

A clarifying review of the available infrastructure was essential for the success of this work. While creating a tutorial and complementing programming and user documentation of the research I got the necessary insights to develop the key concepts of the visualization framework. Still there is no good

overall getting-started-documentation for someone who also might want to utilize the visualization and data environments as yet, because it is research software and no commercial environment. I decided to give quite detailed descriptions and source code excerpts throughout the thesis. Thus, many parts of the thesis can also be read as an introductory to the visualization environment *Vish* and the *Fiber Bundle* data library.

Chapter 2

Theoretical Background

This chapter briefly captures the mathematical and physical background of the thesis. It follows the descriptions of [55], [36], [6], [5], [38],[54], [47] and [40], which were adapted to the requirements of the thesis.

Following notations were used:

- Einstein summing convention is used. Same indices occurring at upper and lower position are summed up: $\alpha_\nu \beta^\nu = \alpha_1 \beta^1 + \alpha_2 \beta^2 + \ldots$

- Partial derivation is written as index: $\frac{\partial f}{\partial x} := f_{,x}$

2.1 Differential Geometry

2.1.1 Manifolds and Charts

Let S be a set. Then $\mathcal{P}(S)$ is the set of all subsets of S, called the **power set** of S. [49]

Let X be a set $\wedge \mathcal{P}$ be the power set. A subset $\tau \subseteq \mathcal{P}(X)$ is a **topology** if:

- arbitrary unions of elements of τ are contained in τ, i.e. if I is an arbitrary (also infinite) set of indices and $\forall i \in I : U_i \in \tau$ then $\bigcup_{i \in I} U_i \in \tau$,

- finite intersections of elements of τ are contained in τ, i.e. if U_0, U_1, ..., $U_n \in \tau$ then $\bigcap_{i=0}^{n} U_i \in \tau$ with $n = \mathbb{N}$,

- the empty set and the set X itself are contained in τ, i.e. $\emptyset, X \in \tau$

The pair (X, τ) of a set X together with a topology τ on this set is a **topological space**. The elements of a topological space are called points. [6]

Two topological spaces X, Y are **homeomorphic**, if there exists a bijective map $H : X \to Y$ such that open sets of X are mapped to open sets in Y and vice versa, i.e. the neighborhood relations must be sustained under this mapping.
H is called homeomorphism or topological map.

A topological space X is a **Hausdorff space** iff for any two distinct points $x, y \in X$ there exist two distinct neighborhoods U_x, U_y such that $U_x \cap U_y = \{\}$.

A **manifold** is a Hausdorff space that is locally homeomorphic to Euclidean space $\mathbb{R}^n$.

A manifold is a topological space which locally looks like the $\mathbb{R}^n$ with the usual topology [54].

A **chart** is a $\{x^\mu\}$ bijective and differentiable mapping (a diffeomorphism) from a point P of a manifold M to a n-tuple of scalar numbers.

$$
\begin{aligned}
q : M &\to \mathbb{R}^n \\
P &\mapsto \{(x^\mu(P)\}
\end{aligned}
\tag{2.1}
$$

The numbers $x^\mu(p)$ are called the coordinates of the point p in the chart $\{x^\mu\}$ and x^i is the ith coordinate function. [6] A chart that does not cover the complete manifold M is called a local chart.

Later, spherical and Cartesian charts are used to represent the metric of curved space times. The spherical chart $\{x^{\bar{\mu}}\}$, with $x^{\bar{1}} \equiv x$, $x^{\bar{2}} \equiv y$ and $x^{\bar{3}} \equiv z$, is transformed to Cartesian chart $\{x^\mu\}$, with $x^1 \equiv r$, $x^2 \equiv \theta$ and $x^3 \equiv \phi$:

$$
\begin{aligned}
x(r, \theta, \phi) &= r \cdot sin\theta \cdot cos\phi \\
y(r, \theta, \phi) &= r \cdot sin\theta \cdot sin\phi \\
z(r, \theta, \phi) &= r \cdot cos\theta
\end{aligned}
\tag{2.2}
$$

And the transformation from $\{x^\mu\}$ to $\{x^{\bar{\mu}}\}$ is:

$$
\begin{aligned}
r(x, y, z) &= \sqrt{x^2 + y^2 + z^2} \\
\theta(x, y, z) &= \arctan\left(\frac{\sqrt{x^2+y^2}}{z}\right) \\
\phi(x, y, z) &= \begin{cases}
\pi/2 & : x = 0, y > 0 \\
3\pi/2 & : x = 0, y < 0 \\
\arctan\frac{y}{x} & : x > 0 \\
\arctan\frac{y}{x} + \pi & : x < 0
\end{cases}
\end{aligned}
\tag{2.3}
$$

2.1.2   Curve

A **curve** is a mapping from a scalar $s \in \mathbb{R}$, the so called curve parameter, to a point on a manifold.

$$
\begin{aligned}
q : \mathbb{R} &\rightarrow M \\
s &\mapsto q(s)
\end{aligned}
\tag{2.4}
$$

To describe a curve in a certain chart $\{x^\mu\}$ the chart-function is used to extract a function for each coordinate of the chart:

$$
\begin{aligned}
q^\mu : \mathbb{R} &\rightarrow \mathbb{R} \\
s &\rightarrow q^\mu(s) = x^\mu(q(s)) \equiv x^\mu \circ q(s)
\end{aligned}
\tag{2.5}
$$

2.1.3   Tangential Vector

The **tangential vector** of a curve q(s) is defined by the operation $\frac{d}{ds}$:

$$
\frac{d}{ds} f(q(s)) := \lim_{h \to 0} \frac{f(q(s+h)) - f(q(s))}{h}
\tag{2.6}
$$

In a certain chart $\{x^\mu\}$:

$$
\begin{aligned}
\frac{d}{ds} f(q(s)) = \frac{d}{ds} f(x^0(q(s)), x^1(q(s)), \dots) &= \frac{\partial f}{\partial x^\mu} \frac{dx^\mu(q(s))}{ds} \\
&= \frac{dx^\mu(q(s))}{ds} \frac{\partial}{\partial x^\mu} f \\
&= \frac{dq^\mu(s)}{ds} \frac{\partial}{\partial x^\mu} f =: \dot{q}^\mu(s) \frac{\partial}{\partial x^\mu} f \\
&= f_{,\mu} \dot{q}^\mu
\end{aligned}
\tag{2.7}
$$

$$
\frac{d}{ds} = \dot{q}^\mu(s) \frac{\partial}{\partial x^\mu} =: \dot{q}
\tag{2.8}
$$

The functions $\dot{q}^\mu$ are the components of the tangential vector in the chart $\{x^\mu\}$. If f is a chart-function x^ν, then $\frac{d}{ds}$ is the νth component $\dot{q}^\nu$:

$$
\frac{d}{ds} x^\nu(q(s)) = \dot{q}^\mu(s) \frac{\partial x^\nu}{\partial x^\mu} = \dot{q}(s) \delta^\nu_\mu = \dot{q}^\nu(s)
\tag{2.9}
$$

The partial derivatives $\frac{\partial}{\partial x^\mu}$ span a vector space at a point P of a manifold M: the tangential vector space $T_P(M)$. Thus, the chart-functions x^μ induce a basis $\{\frac{\partial}{x^\mu}\}$ in $T_P(M)$.

$$
\partial_\mu := \frac{\partial}{\partial x^\mu}
\tag{2.10}
$$

Transformation

A tangential vector can be represented in different charts. For the basis of the tangential space $\{\partial_\mu\}$ and the charts $\{x^\mu\}$ and $\{x^{\bar\mu}\}$

$$\vec\partial \equiv \frac{\partial}{\partial x^{\bar\mu}} = \frac{\partial x^\mu}{\partial x^{\bar\mu}} \cdot \frac{\partial}{\partial x^\mu} =: \alpha_{\bar\mu}^\mu \cdot \vec\partial \tag{2.11}$$

with $\alpha_{\bar\mu}^\mu$ being the inner derivatives of the coordinate transformation:

$$\alpha_{\bar\mu}^\mu = \frac{\partial x^\mu(x^{\bar\mu})}{\partial x^{\bar\mu}} \quad and \quad \alpha_\mu^{\bar\mu} = \frac{\partial x^{\bar\mu}(x^\mu)}{\partial x^\mu}. \tag{2.12}$$

Now, the tangential vector can be written as:

$$v = v^\mu \partial_\mu = v^{\bar\mu} \partial_{\bar\mu} = v^{\bar\mu}(\alpha_{\bar\mu}^\mu \partial_\mu) \tag{2.13}$$

It follows:

$$v^\mu = \alpha_{\bar\mu}^\mu v^{\bar\mu} \quad \partial_\mu = \alpha_\mu^{\bar\mu} \partial_{\bar\mu} \tag{2.14}$$

To fully describe the chart transformation $\alpha_{\bar\mu}^\mu(x^\mu)$, $\alpha_\mu^{\bar\mu}(x^\mu)$, $\alpha_{\bar\mu}^\mu(x^{\bar\mu})$ and $\alpha_\mu^{\bar\mu}(x^{\bar\mu})$ are needed, which can be written as transformation matrices.

One such transformation matrix is used in *section 7.3* and *section 7.4* to transform a metric given in spherical Coordinates to Cartesian coordinates. The according transformation matrix is:

$$\alpha_\mu^{\bar\mu}(x^\mu) = \begin{bmatrix} \frac{\partial r}{\partial x} = \frac{x}{\sqrt{x^2+y^2+z^2}} & \frac{\partial r}{\partial y} = \frac{y}{\sqrt{x^2+y^2+z^2}} & \frac{\partial r}{\partial z} = \frac{z}{\sqrt{x^2+y^2+z^2}} \\ \frac{\partial\theta}{\partial x} = \frac{xz}{\sqrt{x^2+y^2}(x^2+y^2+z^2)} & \frac{\partial\theta}{\partial y} = \frac{yz}{\sqrt{x^2+y^2}(x^2+y^2+z^2)} & \frac{\partial\theta}{\partial z} = -\frac{\sqrt{x^2+y^2}}{x^2+y^2+z^2} \\ \frac{\partial\phi}{\partial x} = \frac{-y}{x^2+y^2} & \frac{\partial\phi}{\partial y} = \frac{x}{x^2+y^2} & \frac{\partial\phi}{\partial z} = 0 \end{bmatrix} \tag{2.15}$$

2.1.4 Covector

Let $v \in T_P(M)$ be a vector and $f : M \to \mathbb{R}$ a real valued function on the manifold. Then applying the tangential vector v of f yields a number:

$$v(f) \in \mathbb{R} \tag{2.16}$$

Above we examined this expression for a fixed vector v and arbitrary function f. Alternatively, we can study it for arbitrary v and fixed function f. This way it defines the function df which maps a tangential vector v to a number $df(v) \in \mathbb{R}$:

$$\begin{aligned} df : T_P(M) &\to \mathbb{R} \\ v &\to df(v) := v(f) \end{aligned} \tag{2.17}$$

The function df is called **1-form**. 1-forms fulfill the vector space axioms:

$$(a \cdot df + b \cdot dg)(v) = v(a \cdot f + b \cdot g) = a \cdot v(f) + b \cdot v(g) = a \cdot df(v) + b \cdot dg(v) \quad (2.18)$$

and thus span a vector space, called the cotangential space $T_P^*(M)$. Its elements, calles **covectors**, are linear maps $T_P(M) \to \mathbb{R}$.

2.1.5 Tensor Field

Physical quantities are independent of an underlying coordinate system. This is known as the "principle of covariance". Mathematically, tensors are used to describe such quantities. A tensor can be written in any coordinate basis. The name tensor originates from its usage in continuum mechanics, where a tensor of rank two is used to describe stresses, [40] or [41].

An $m \times n$ **tensor** F is a multilinear mapping from Cartesian products of n tangential spaces $T_P(M)$ and m cotangential spaces $T_P^*(M)$ into $\mathbb{R}$ at some point P of a manifold:

$$F : (T_P(M))^n \times (T_P^*(M))^m \overset{\text{multilinear}}{\longrightarrow} \mathbb{R} \quad (2.19)$$

$m \times n$ is the rank of the tensor. The vector space spanned by tensors is called the tensor product space

$$(T_P^*(M))^{\otimes n} \otimes (T_P(M))^{\otimes m} := \{ (T_P(M))^n \times (T_P^*(M))^m \overset{\text{multilinear}}{\longrightarrow} \mathbb{R} \}, \quad (2.20)$$

whereby $X^{\otimes n}$ denotes the nth tensor product of the space X with itself. Due to the duality relations $T_P^*(M) \leftrightarrow T_P(M)$, the tensor product space can be seen as the space of linear functions that map elements from the dual tensor product space to a number:

$$(T_P^*(M))^{\otimes n} \otimes (T_P(M))^{\otimes m} = \{ (T_P(M))^{\otimes n} \otimes (T_P^*(M))^{\otimes m} \overset{\text{linear}}{\longrightarrow} \mathbb{R} \}, \quad (2.21)$$

With V, U two tensor product spaces, we see

$$V \otimes U = \{ V^* \times U^* \overset{\text{multilinear}}{\longrightarrow} \mathbb{R} \} = \{ V^* \otimes U^* \overset{\text{linear}}{\longrightarrow} \mathbb{R} \} \quad (2.22)$$

and

$$V^* \otimes U^* = \{ \overset{\text{multilinear}}{V \times U \quad \to \quad} \mathbb{R} \} = \{ \overset{\text{linear}}{V \otimes U \quad \to} \mathbb{R} \} \tag{2.23}$$

It follows from *equation (2.21)* that $V^* \otimes U^* \equiv (V \otimes U)^*$. [6]

The **tensor field** $\mathfrak{t}$ is a mapping from a point P of a manifold M to a tensor in the tangential space $T_P(M)$.

$$\mathfrak{t} : \mathcal{M} \quad \to \quad (T_P^*(M))^{\otimes n} \otimes (T_P(M))^{\otimes m} \tag{2.24}$$

Tensor fields are used in many physical and technical applications, for example, to describe distributions of temperature, velocity, stress or curvature in space and time.

A tensor field of rank 0×0 is a scalar field $M \to \mathbb{R}$. A tensor field of rank 0×1 is a vector field $v : M \to T_P(M)$. [6]
A metric field is of rank 2×0.

2.1.6 Metric

A metric G is a symmetric bilinear form on tangential vectors of a manifold. G is a bilinear mapping $T_P(M) \times T_P(M) \to \mathbb{R}$. With u, v, w being tangential vectors, λ a scalar:

$$\begin{aligned} G(u + \lambda \cdot w, v) &= G(u, v) + \lambda \cdot G(w, v) \\ G(u, v + \lambda \cdot w) &= G(u, v) + \lambda \cdot G(u, w) \\ G(u, v) &= G(v, u) \end{aligned} \tag{2.25}$$

In general relativity a metric tensor field is used to describe the curvature of spacetime. The metric at one point of a manifold is represented as a 4×4 matrix $g_{\mu\nu}$. Because of its symmetry it has 10 independent components. In this work the signature $(+, -, -, -)$ is used. Thus, the spatial components have a negative sign in contrast to the time components which are positive.

There exist three types of tangential vectors:

- $G(v, v) > 0 \leftrightarrow v$ is called time-like

- $G(w, w) = 0 \leftrightarrow w$ is called light-like

- $G(u, u) < 0 \leftrightarrow u$ is called space-like

The metric defines lengths of and angles between vectors. The length of a time-like tangential vector is

$$|v| := \sqrt{G(v,v)} \tag{2.26}$$

and of a space-like tangential vector it is

$$|u| := \sqrt{-G(u,u)}. \tag{2.27}$$

The angle between two space-like tangential vectors is

$$cos\alpha(u,v) := \frac{G(u,v)}{|u| \cdot |v|} = \frac{G(u,v)}{\sqrt{G(u,u) \cdot G(v,v)}} \tag{2.28}$$

A metric is written in a certain chart $\{x^\mu\}$ as follows:

$$G(u,v) = G(u^\mu \partial_\mu, v^\nu \partial_\nu) = u^\mu \cdot v^\nu \cdot G(\partial_\mu, \partial_\nu) =: u^\mu v^\nu g_{\mu\nu} \tag{2.29}$$

For example, the metric tensor for flat space times in Cartesian and spherical coordinates are:

$$g_{\mu\nu} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \quad \mathring{g}_{\mu\nu} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2 sin^2\theta \end{bmatrix} \tag{2.30}$$

If a metric is invertible the **co-metric** is defined via the Kronecker delta and is the inverse of the metric:

$$g_{\mu\nu} g^{\nu\lambda} = \delta^\lambda_\mu \tag{2.31}$$

2.1.7   Geodesic Equation and Christoffel Symbols

Geodesic Equation

A geodesic is a curve with extremal length on a manifold. The length of a curve along a parameter interval s can be computed by integration using the metric tensor of *equation (2.29)*:

$$\int_0^s \underbrace{g_{q(\sigma)}(\dot{q}(\sigma), \dot{q}(\sigma))}_{:= \mathcal{L}} d\sigma \tag{2.32}$$

To compute the extremum of the length we choose this expression as the Lagrange function and compute the total differential:

$$d\mathcal{L} = \frac{\partial \mathcal{L}}{\partial s} ds + \frac{\partial \mathcal{L}}{\partial q^k(s)} dq^k(s) + \frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} d\dot{q}^k(s) \tag{2.33}$$

with:

$$dq^k(s) = \frac{\partial q^k(s)}{\partial s}ds, \quad \frac{\partial q^k(s)}{\partial s} = \dot{q^k}(s) \quad and \quad q^k(s) = x^k(q(s)) \tag{2.34}$$

$$d\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial s} + \frac{\partial \mathcal{L}}{\partial x^k}\dot{q}^k(s) + \frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)}\ddot{q}^k(s) \right) ds \tag{2.35}$$

The partial derivative $\frac{\partial \mathcal{L}}{\partial s}$ is 0, since $\mathcal{L}$ is not dependent on s. For minimization we claim:

$$\int d\mathcal{L} = \int \left(\frac{\partial \mathcal{L}}{\partial x^k}\dot{q}^k(s) + \frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)}\ddot{q}^k(s) \right) ds = 0 \tag{2.36}$$

Now we use partial integration on the second term in the integral of *equation (2.36)* to further simplify the equation:

$$\begin{matrix} \int f' & g & = & f & g & - & \int g & f' \\ \int \ddot{q}^k(s) & \frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} & = & \dot{q}^k(s) & \frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} & - & \int \frac{d}{ds}\frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} & \dot{q}^k(s) \end{matrix} \tag{2.37}$$

Insertion of *equation (2.37)* in *equation (2.36)* yields:

$$\int \overbrace{\left(\frac{\partial \mathcal{L}}{\partial x^k} - \frac{d}{ds}\frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} \right)}^{= 0} \dot{q}^k(s)ds + \overbrace{\dot{q}^k(s)\frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)}}^{= const} = 0 \tag{2.38}$$

the last term in *equation (2.38)* is a constant term and thus must be 0. Also, the term in brackets () must be 0 because $\partial\mathcal{L}$ must be an extremum for any curve $q(s)$. This, simplifies the equation to:

$$\frac{\partial \mathcal{L}}{\partial x^k} - \frac{d}{ds}\frac{\partial \mathcal{L}}{\partial \dot{q}^k(s)} = 0 \tag{2.39}$$

We introduce coordinates and compute the terms of *equation (2.39)* using the Einstein notation :

$$\mathcal{L} = g_{q(s)}(\dot{q}(s), \dot{q}(s)) = g_{\mu\nu} \cdot \dot{q}^\mu(s) \cdot \dot{q}^\nu(s) \tag{2.40}$$

First term of *equation (2.39)*:

$$\frac{\partial \mathcal{L}}{\partial x^\alpha} = \mathcal{L}_{,\alpha} = g_{\mu\nu,\alpha}\dot{q}^\mu\dot{q}^\nu + g_{\mu\nu}\dot{q}^\mu_{,\alpha}\dot{q}^\nu + g_{\mu\nu}\dot{q}^\mu\dot{q}^\nu_{,\alpha} = g_{\mu\nu,\alpha}\dot{q}^\mu\dot{q}^\nu + 2g_{\mu\nu}\dot{q}^\mu_{,\alpha} \tag{2.41}$$

The metric tensor is symmetric. Thus coordinate indices may be flipped and the two terms containing $g_{\mu\nu}$ be added. The term finally vanishes because:

$$\dot{q}^\nu_{,\alpha} = \frac{\partial \dot{q}^\nu}{dx^\alpha} = 0 \quad \rightarrow \quad \frac{\mathcal{L}}{\partial x^\alpha} = g_{\mu\nu,\alpha} \dot{q}^\mu \dot{q}^\nu \tag{2.42}$$

Second term of *equation (2.39)*:

$$\frac{\partial \mathcal{L}}{\partial \dot{q}^\alpha(s)} = \overbrace{\frac{\partial g_{\mu\nu}}{\dot{q}^\alpha(s)}}^{=\,0} \dot{q}^\mu(s)\dot{q}^\mu(s) + g_{\mu\nu} \overbrace{\frac{\partial \dot{q}^\mu}{\dot{q}^\alpha(s)}}^{=\,\delta^\mu_\alpha} \dot{q}^\nu(s) + g_{\mu\nu}\dot{q}^\mu(s) \overbrace{\frac{\partial \dot{q}^\nu}{\dot{q}^\alpha(s)}}^{=\,\delta^\nu_\alpha} \tag{2.43}$$

Again, using the symmetry of $g_{\mu\nu}$:

$$\frac{\partial \mathcal{L}}{\partial \dot{q}^\alpha(s)} = g_{\alpha\nu}\dot{q}^\nu + g_{\mu\alpha}\dot{q}^\mu = 2g_{\mu\alpha}\dot{q}^\mu \tag{2.44}$$

$$\frac{d}{ds}\frac{\partial \mathcal{L}}{\partial \dot{q}^\alpha(s)} = 2\left(\frac{d}{ds}g_{\mu\alpha}\dot{q}^\mu + g_{\mu\alpha}\frac{d}{ds}\dot{q}^\mu\right) \tag{2.45}$$

$$\frac{d}{ds}g_{\mu\alpha} = \frac{\partial g_{\mu\alpha}}{\partial x^\nu}\frac{d}{ds}q^\nu = g_{\mu\alpha,\nu}\dot{q}^\nu \tag{2.46}$$

$$\frac{d}{ds}\dot{q}^\mu = \ddot{q}^\mu \tag{2.47}$$

$$\frac{d}{ds}\frac{\partial \mathcal{L}}{\partial \dot{q}^\alpha(s)} = 2(g_{\mu\alpha,\nu}\dot{q}^\nu\dot{q}^\mu + \ddot{q}^\mu\dot{q}^\mu) \tag{2.48}$$

All necessary terms are derived now. Insertion of *equation (2.46)* in *equation (2.45)* and further in *equation (2.39)* and *equation (2.42)* in *equation (2.39)*:

$$g_{\mu\nu,\alpha}\dot{q}^\mu\dot{q}^\nu - 2(g_{\mu\alpha,\nu}\dot{q}^\mu\dot{q}^\nu + g_{\mu\alpha}\ddot{q}^\mu) = 0 \tag{2.49}$$

$$-2g_{\mu\alpha}\ddot{q}^\mu - 2\left(g_{\mu\alpha,\nu} - \frac{1}{2}g_{\mu\nu,\alpha}\right)\dot{q}^\mu\dot{q}^\nu = 0 | \cdot \left(-\frac{1}{2}g^{\alpha\lambda}\right) \tag{2.50}$$

$$\ddot{q}^\mu + \underbrace{\frac{1}{2}g^{\alpha\lambda}(2g_{\mu\alpha,\nu} - g_{\mu\nu,\alpha})}_{=:\,A^\lambda_{\mu\nu}} \dot{q}^\mu\dot{q}^\nu = 0 \tag{2.51}$$

Introducing the abbreviation $A^\lambda_{\mu\nu}$, where only the symmetric part contributes:

$$-\ddot{q} = A^\lambda_{\mu\nu}\dot{q}^\mu\dot{q}^\nu = A^\lambda_{\nu\mu}\dot{q}^\mu\dot{q}^\nu = A^\lambda_{\nu\mu}\dot{q}^\nu\dot{q}^\mu = \underbrace{\frac{1}{2}(A^\lambda_{\mu\nu} + A^\lambda_{\nu\mu})}_{=: \Gamma^\lambda_{\mu\nu}}\ \dot{q}^\nu\dot{q}^\mu \qquad (2.52)$$

With $\Gamma^\lambda_{\mu\nu}$ called the **Christoffel Symbols**, the final geodesic equation is:

$$\ddot{q}^\lambda = \Gamma^\lambda_{\mu\nu}\dot{q}^\mu\dot{q}^\nu \qquad (2.53)$$

and Christoffel symbols:

$$\Gamma^\lambda_{\mu\nu} := \frac{1}{2}g^{\lambda\alpha}\left(\frac{1}{2}(2g_{\mu\alpha,\nu} - g_{\mu\nu,\alpha}) + \frac{1}{2}(2g_{\nu\alpha,\mu} - g_{\nu\mu,\alpha})\right) \qquad (2.54)$$

$$\Gamma^\lambda_{\mu\nu} := \frac{1}{2}g^{\lambda\alpha}(g_{\mu\alpha,\nu} + g_{\nu\alpha,\mu} - g_{\mu\nu,\alpha}) \qquad (2.55)$$

The geodesic $q(s)$ is an **integral line** because solving the *equation (2.53)* involves integration for computation. It is of second order since it is defined via a second order derivative. To solve for $q(s)$ the boundary conditions for $\dot{q}(0)$ and $q(0)$ are required. The second derivative $\ddot{q}$ can be interpreted as an acceleration. Since a geodesic represents an unaccelerated motion in curved spacetime this term is more precisely called **coordinate-acceleration**.

Christoffel Symbols

To study the structure of Christoffel symbols four specific symbols are expanded. Here, it is assumed a 3D space. Consider a metric tensor g in xyz coordinates. The dimension of g is 3×3 and we have to compute $3\times3\times3 = 27$ Christoffel symbols. Expanding $\Gamma^\lambda_{\mu\nu}$ for $\lambda = x$, $\mu = x$ and $\nu = x$ for Γ^x_{xx} yields:

$$\begin{aligned}\Gamma^x_{xx} = \ \ \tfrac{1}{2}[\ \ &g^{xx}(g_{xx,x} + g_{xx,x} - g_{xx,x})+ \\ &g^{xy}(g_{xy,x} + g_{xy,x} - g_{xx,y})+ \\ &g^{xz}(g_{xz,x} + g_{xz,x} - g_{xx,z})]\end{aligned} \qquad (2.56)$$

$$\Gamma^x_{xx} = \frac{1}{2}\left(g^{xx}(g_{xx,x}) + g^{xy}(2g_{xy,x} - g_{xx,y}) + g^{xz}(2g_{xz,x} - g_{xx,z})\right) \qquad (2.57)$$

Next, three more Chirstoffel symbols are expanded varying just one index, e.g. $\lambda = y$, Γ^y_{xx} yields:

$$
\begin{aligned}
\Gamma^y_{xx} = \quad \tfrac{1}{2}[\quad & g^{yx}(g_{xx,x} + g_{xx,x} - g_{xx,x})+ \\
& g^{yy}(g_{xy,x} + g_{xy,x} - g_{xx,y})+ \\
& g^{yz}(g_{xz,x} + g_{xz,x} - g_{xx,z})]
\end{aligned}
\tag{2.58}
$$

$$
\Gamma^y_{xx} = \frac{1}{2}\left(g^{yx}(g_{xx,x}) + g^{yy}(2g_{xy,x} - g_{xx,y}) + g^{yz}(2g_{xz,x} - g_{xx,z})\right)
\tag{2.59}
$$

The Christoffel symbol with $\nu = y$, Γ^x_{xy} yields:

$$
\begin{aligned}
\Gamma^x_{xy} = \quad \tfrac{1}{2}[\quad & g^{xx}(g_{xx,y} + g_{yx,x} - g_{xy,x})+ \\
& g^{xy}(g_{xy,y} + g_{yy,x} - g_{xy,y})+ \\
& g^{xz}(g_{xz,y} + g_{yz,x} - g_{xy,z})]
\end{aligned}
\tag{2.60}
$$

The Christoffel symbol with $\mu = y$, Γ^x_{xy} yields:

$$
\begin{aligned}
\Gamma^x_{yx} = \quad \tfrac{1}{2}[\quad & g^{xx}(g_{yx,x} + g_{xx,y} - g_{yx,x})+ \\
& g^{xy}(g_{yy,x} + g_{xy,y} - g_{yx,y})+ \\
& g^{xz}(g_{yz,x} + g_{xz,y} - g_{yx,z})]
\end{aligned}
\tag{2.61}
$$

When comparing the terms of *equation (2.56)* and *equation (2.58)* it can be seen that the terms in between the round brackets () remain unchanged, since they depend on indices μ and ν only. The same holds when comparing the terms of *equation (2.60)* and *equation (2.61)*. Because of the symmetry of the metric tensor $g_{\mu\nu}$ (e.g. $g_{xy,x} == g_{yx,x}$) the terms in round brackets () are equal.

Because of the symmetry property of the metric tensor also the Christoffel Symbols are symmetrical. In the 3D case the number of independent components is reduced from 27 to 18, in 4D from 64 to 40.

2.1.8 Geodesic Deviation and Riemann Tensor

The geodesic deviation describes the change in separation of neighboring geodesics. The Riemann Tensor connects the deviation of the parallelism of neighbored geodesics to the curvature of the spacetime:

$$
K^\gamma_{\mu\delta\beta} = \Gamma^\gamma_{\beta\mu,\delta} - \Gamma^\gamma_{\delta\mu,\beta} + \Gamma^\kappa_{\beta\mu}\Gamma^\gamma_{\delta\kappa} - \Gamma^\kappa_{\delta\mu}\Gamma^\gamma_{\beta\kappa}
\tag{2.62}
$$

The Riemann tensor is of rank four[1] and has 4^4 components in $\mathbb{R}^4$. However, the number of independent components reduces to 20 when its symmetry properties are analyzed, [38].

---

[1]or (1,3)

2.1.9 Ricci Tensor and Scalar

The **Ricci tensor** is the only not vanishing contraction of the Riemann tensor:

$$R_{\mu\nu} = K^{\kappa}_{\mu\kappa\nu} \tag{2.63}$$

The Ricci tensor is symmetric and has 10 independent components in $\mathbb{R}^4$. Geometrically interpreted it sums up curvatures of orthogonal geodesics surfaces at an tangential vector $v \in T_P(M)$.

Further contraction of the Ricci tensor yields the so called **Ricci scalar**, which is an invariant of the Ricci tensor.

$$\mathfrak{R} = R^{\mu}_{\mu} \tag{2.64}$$

2.2   General Relativity

In 1916 Albert Einstein introduced his relativistic theory of gravitation, see
[25]. The general relativity describes gravity, which is the geometry of a four
dimensional spacetime. Due to the curved spacetime an object that experi-
enced acceleration of a gravitational force in classical mechanics is unaccel-
erated in general relativity. In fact it is just moving uniformly on a straight
line, on a geodesic, but in a curved spacetime. Trajectories of photons and
free falling objects are represented by geodesics.

Using differential geometry, spacetime is described by an infinitesimal
small line element specifying a distance between any two neighboring points.
A four dimensional metric, *section 2.1.6*, is used for that purpose:

$$ds^2 = g_{\mu\nu}dx^\mu dx^\nu \tag{2.65}$$

A metric tensor field describes the curvature at every point of the spacetime
manifold, *section 2.1.6*.

2.2.1   Einstein Field Equation

The heart of the general relativity theory is the equation that correlates mat-
ter and energy with the spacetime curvature. It is known as the Einstein field
equation. When a distribution of matter is given, the spacetime curvature
can be computed by this equation.

$$R_{\mu\nu} - \frac{1}{2}g_{\mu\nu}\mathfrak{R} = 8\pi G T_{\mu\nu} \tag{2.66}$$

The left hand side of *equation (2.66)* describes the curvature of spacetime
using the *Ricci* tensor and the *Ricci* scalar, *section 2.1.9*. The right hand
side describes the distribution of matter by a tensor of rank two.

$T_{\mu\nu}$ is called the energy-momentum-stress tensor and at it full complexity
captures a scalar field for energy density, a vector field for the motion of
matter, a scalar field for pressure, a space-like tensor field for stress and a
vector field for energy flux.

The *equation (2.66)* is a coupled system of differential equations of second
order, which to the full extend can only be solved numerically. However, some
analytic solutions exist that make use of strong simplifications.

For example, for solutions in pure vacuum the equation simplifies to
$R_{\mu\nu} = 0$. The Schwarzschild metric and the Kerr metric are such analytic
vacuum solutions of the Einstein field equation.

2.2.2 Schwarzschild Metric

"The solution of the field equations, which describe the field outside of a spherical symmetric mass distribution, was found by Karl Schwarzschild only two months after Einstein published his field equations. ... From this solution he derived the precession of the perihelion of Mercury and the bending of light rays at the surface of the sun." [54]

The metric is expressed in spherical coordinates, with m being the mass expressed as a length:

$$m(\text{in cm}) := \frac{G}{c^2} M(\text{in g}) \tag{2.67}$$

$$g = \left(1 - \frac{2m}{r}\right) dt^2 - \frac{dr^2}{1 - 2m/r} - r^2(d\theta^2 + sin^2\theta d\phi^2) \tag{2.68}$$

To write g in matrix form the equation is expanded and the terms that are multiplied by the squared derivatives are extracted:

$$g = \begin{bmatrix} 1 - \frac{2m}{r} & 0 & 0 & 0 \\ 0 & -\left(1 - \frac{2m}{r}\right)^{-1} & 0 & 0 \\ 0 & 0 & -r^2 & 0 \\ 0 & 0 & 0 & -r^2 sin^2\theta \end{bmatrix} \tag{2.69}$$

The according Christoffel symbols in polar coordinates are, [36]:

$$
\begin{aligned}
&\Gamma^t_{tr} = (m/r^2)(1 - 2m/r)^{-1} &\quad& \Gamma^\theta_{r\theta} = 1/r \\
&\Gamma^r_{tt} = (m/r^2)(1 - 2m/r) &\quad& \Gamma^\theta_{\phi\phi} = -cos\theta sin\theta \\
&\Gamma^r_{rr} = -(m/r^2)(1 - 2m/r)^{-1} &\quad& \Gamma^\phi_{r\phi} = 1/r \\
&\Gamma^r_{\theta\theta} = -(r - 2m) &\quad& \Gamma^\phi_{\theta\phi} = cot\theta \\
&\Gamma^r_{\phi\phi} = -(r - 2m)sin^2\theta
\end{aligned}
\tag{2.70}
$$

Important properties of the Schwarzschild Metric:

- It is time independent and, thus, a stationary and static metric field.

- It is spherically symmetric.

- It has a singularity at $r = 0$ and $r = 2m$ (Schwarzschild radius).

The radius of a static star is always outside the Schwarzschild radius. "The Schwarzschild radius of the sun, for instance, is $2GM/c^2 = 2.95$ km - much smaller than the radius of the solar surface 6.96×10^5 km." [36].

The singularity at $r = 0$ is a physical singularity at the center point. Here, curvature and mass becomes infinite.

The Schwarzschild radius is a singularity induced by the coordinate system. Other coordinates, such as Eddington-Finkelstein coordinates can be used to eliminate this singularity. However, it still has a physical meaning. Light rays or particles passing this radius towards the center cannot escape the heavy mass. The surface at the Schwarzschild radius is called the event horizon of a black hole.

Properties of the Schwarzschild metric are explored by visualizing geodesics in *section 7.3*.

2.2.3 Kerr Metric

The Kerr metric was discovered by Roy Kerr in 1963. They are a generalization of the Schwarzschild metric by rotation. It can describe the actual endstate of collapsed astrophysical objects quite well. Besides the mass now an additional parameter, the angular momentum, controls the spacetime geometry. The so called "no hair theorem" states that at the endstate of a collapse only three quantities are conserved: the mass, the angular momentum and the electric charge, see [20].

"Yet the evidence of both theoretical investigation and numerical simulations is that the endstate of *any* realistic gravitational collapse that proceeds far enough is remarkably simple, analogous in many ways to the special case of spherical collapse. ...

From the perspective of a distant observer, the endstate is indistinguishable from a time-independent Kerr black hole characterized by just mass M and angular momentum J, with a horizon that conceals the singularity in it. ...

Kerr black holes thus provide the cleanest connection between fundamental gravitational physics and realistic astrophysics."[36]

The Kerr metric is formulated in Boyer-Lindquist coordinates, with:

$$a = J/M$$
$$\Delta = r^2 - 2Mr + a^2 \tag{2.71}$$
$$\rho^2 = r^2 + a^2 cos\theta$$

The parameter a is called the Kerr parameter. If $a = 0$ then the Boyer-Lindquist coordinates are equivalent to the Schwarzschild coordinates. The metric is:

$$g = \frac{\Delta}{\rho^2}[dt - asin^2\theta d\phi]^2 - \frac{sin^2\theta}{\rho^2}[(r^2 + a^2)d\phi - adt]^2 - \frac{\rho^2}{\Delta}dr^2 - \rho^2 d\theta^2 \tag{2.72}$$

Written in components:

$$
\begin{aligned}
(t,t): \quad & \tfrac{\Delta}{\rho^2}dt^2 - \tfrac{sin^2\theta}{\rho^2}a^2 dt^2 & \to g_{tt} = \quad & \tfrac{1}{\rho^2}(\Delta - a^2 sin^2\theta) \\
(r,r): \quad & -\tfrac{\rho^2}{\Delta}dr^2 & \to g_{rr} = \quad & -\tfrac{\rho^2}{\Delta} \\
(\theta,\theta): \quad & -\rho^2 d\theta^2 & \to g_{\theta\theta} = \quad & -\rho^2 \\
(\phi,\phi): \quad & (\tfrac{\Delta}{\rho^2}a^2 sin^4\theta - \tfrac{sin^2\theta}{\rho^2}(r^2+a^2)^2)d\theta^2 & \to g_{\phi\phi} = \quad & \tfrac{sin^2\theta}{\rho^2}(\Delta a^2 sin^2\theta - \\
& & & \quad -(r^2+a^2)^2) \\
(\phi,t): \quad & (-\tfrac{\Delta}{\rho^2}2a sin^2\theta + \tfrac{sin^2\theta}{\rho^2}2(r^2+a^2)a)d\phi dt & \to g_{\phi t} = \quad & \tfrac{sin^2\theta}{\rho^2}2a\cdot \\
& & & \quad \cdot\big((r^2+a^2)-\Delta\big)
\end{aligned}
$$

$$(2.73)$$

Written in matrix form:

$$
g = \begin{bmatrix}
g_{tt} & 0 & 0 & g_{\phi t} \\
0 & g_{rr} & 0 & 0 \\
0 & 0 & g_{\theta\theta} & 0 \\
g_{t\phi} & 0 & 0 & g_{\phi\phi}
\end{bmatrix}
\tag{2.74}
$$

Important properties of the Kerr metric:

- For $r \gg M$ and $r \gg a$ the metric as asymptotically flat, a flat spacetime far from the black hole.

- It is stationary axisymmetric. As the Schwarzschild metric it is independent of t. Also it is independent on ϕ.

- It reduces to Schwarzschild when $a = 0$.

- It has a singularity at $\rho = 0$ ($r = 0$ and $\theta = \pi/2$) and at $\Delta = 0$.

Similar to the Schwarzschild metric the real physical singularity is at $\rho = 0$, where curvature gets infinite. In contrast to the Schwarzschild metric the singularity has the shape of a ring. The second singularity is a coordinate singularity. Δ vanishes at $r_\pm = M \pm \sqrt{M^2 - a^2}$. The radius r_+ relates to the event horizon in the Schwarzschild case.

The radius of the horizon r_+ exists only for $a <= M$ and thus the angular is limited. When $a = m$ then the metric is called an extreme Kerr black hole.

Properties of the Kerr metric are explored by visualizing geodesics in *section 7.4.*

2.3   Fluid Dynamics

Computational fluid dynamics (CFD) uses numerical techniques to solve the equations describing the mechanics of fluids, such as the Navier Stokes equations. These equations are second order partial differential equations which cannot by solved analytically when, for example, complex boundary geometries have to be captured.

With increasing computing power simulation results grow in resolution and size in respect to time and space. Typical results of simulations are tensor fields describing pressure, velocity, density and stress. The field that captures the motion of fluid particles is the velocity field.

Mathematically, it is a time dependent vector field with the vector $v \in T_P(M)$ being an element of the tangential space at a point P of a manifold M.

Based on such a vector field integral lines $q(s) \subset M$ can be computed. In CFD four different types of integral lines are used for exploring the velocity field, [16], also included in *appendix C*:

- **Path lines** or *trajectories* follow the evolution of a test particle as it is dragged around by the vector field over time.

- **Stream lines** or *field lines* represent the instantaneous direction of the vector field (no time evolution). They are identical to path lines if the vector field is constant over time.

- **Streak lines** represent the trace of repeatedly emitted particles from the same location, such as the trail of smoke.

- **Material lines** or *time lines* depict the location of a set of particles, initially positioned along a seed line, under the flow of the vector field.

Figure 2.1 and *figure 2.2* illustrate these different types of integral lines in a vector field.

In this thesis stream lines were visualized as a pre-stage to geodesics. With $\mathfrak{v}$ denoting the vector field, stream lines are defined as:

$$\dot{q}(s) = \mathfrak{v}(q(s)) \tag{2.75}$$

The stream line is a integration line of first order and, thus, one boundary condition $q(0)$ is required to solve for $q(s)$. The numerical computation of streamlines is described in *section 6.2* and results are shown in *section 7.1* and *section 7.2*.
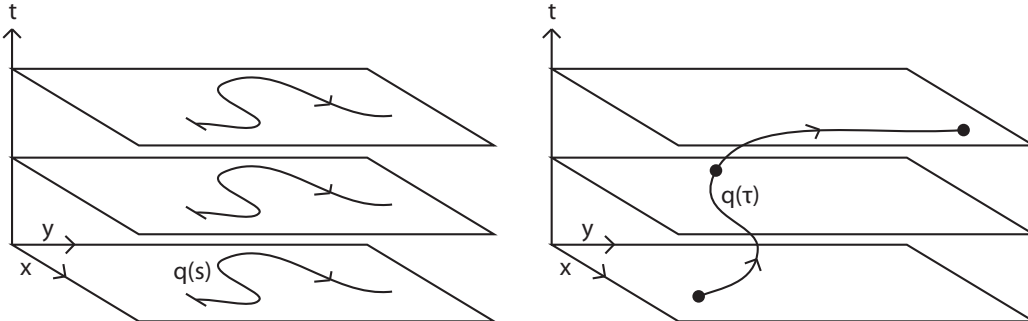
Figure 2.1: *Left:* Stream line in a vector field. The stream line follows the vector field at a frozen instance of time. The stream line parameter is not dependent on the time. *Right:* Path line in a vector field. It represents the trajectory of a particle moving through the vector field over time. If the vector field is stationary the path line is also a stream line.
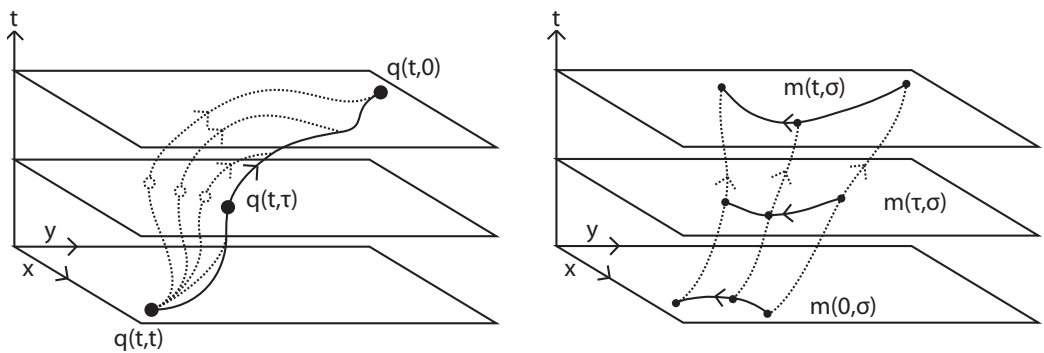
Figure 2.2: *Left:* Streak lines are the connection of particles continuously emitted from the same location over time in a vector field. It is the connection of endpoints of the path lines of each particle. *Right:* Material lines are the time evolution of lines over time in a vector field. Each point on the line corresponds to a particle of one path line.

2.4 Medical Imaging

An application where tensor fields are used stems from medicine. The technology developed in magnetic resonance imaging (MRI) allows to capture the diffusion of hydrogen in human tissue. The resulting tensor fields describe the speed of diffusion in all spatial directions.

The aim of analyzing the anisotropic diffusion is to identify different kinds of brain matters, such as white matter (elongated neurons) which has a high directionality in contrast to grey matter (glial cells) which has very little directionality.

As stated in [9], tumor regions are dominated by grey matter and thus are regions with low directionality. Tensor field visualization can be used to reveal such regions.

Finding accurate and robust visualization techniques could play an important role in diagnostics of brain tumor diseases using MRI scans. Early and precise diagnostic and correct treatment is of great importance to possibly increase the survival rate which, at the moment, is of about 30% only, see [9].

Mathematically the anisotropic diffusion is described by a scalar density function $\Phi(x,t)$, e.g. the time-dependent concentration of water molecules:

$$\frac{\partial \Phi(x,t)}{\partial t} = \nabla \cdot \mathfrak{D}[\nabla \Phi(x,t)] \qquad (2.76)$$

Here, $\mathfrak{D}$ is called the flux and is a function of the concentration gradient $\nabla \Phi(x,t)$. The flux $\mathfrak{D}(v)$ can be expanded into a Taylor series in the Euclidean chart $\{x^\mu\}$ with chart-functions $x^i = e^i$:

$$\mathfrak{D}(v) = D_i(v)e^i = \left(D_i(0) + D_{i,j}(0)v^j + \frac{1}{2}D_{ij,k}(0)v^j v^k + ... \right) e^i \quad (2.77)$$

The constant flux $D_i(0)$ vanishes in *equation (2.76)* and the second term, the $D_{i,j}(0)$ is a tensor of rank two. The tensor is symmetric and positive definite, since particles move forward only ($D_{i,j}(0)v^j > 0$). $D_{i,j}(0)$ is called the **diffusion tensor**.

The diffusion tensor is comparable to the metric tensor in general relativity, see *equation (2.29)*, which is also symmetric and positive definite. The geodesic *equation (2.53)* can be utilized, exchanging the metric with the diffusion tensor, to compute geodesics in the diffusion field.

The geodesics then represent lines that follow the extremal diffusion speed. In *section 7.5* such spatial geodesics where computed in a numerical diffusion tensor field stemming from a MRI scan of a human brain.

Chapter 3

Implementation Concepts

Software can be implemented in a many different ways. Especially C++ provides a lot of flexibility on how problems can be solved and algorithms can be implemented.

This chapter shortly describes how C++ template meta programming can enhance code re-usability, what problems were encountered utilizing the standard template library and it introduces a mechanism to equip classes with interfaces at runtime.

3.1 Type Traits

The introduction of templates to C++ opened a wide range of new programming concepts. Todd Veldhuizen described how to implement programs evaluated at compile time using template meta programming[1], see [59]. Template meta programming theoretically[2] provides a programming language inside C++ that is Touring complete (see [60]) and executes at compile time. Functional programming techniques can be fully utilized to build template programs.

Geoffrey Furnish discusses the pros and cons of using C++ for development of numerical algorithms, see [31]. According to him, C++ code often fails the programming paradigm of high re-usability because of its flexibility to define data structures and classes. He recognized that exchanging and reusing code is easier, for example, in FORTRAN when only limited data structures, like predefined arrays, were used. However, one can use the power, flexibility and speed of C++ while still developing highly reusable code. He

---

[1]The first meta program was written by Erwin Unruh and presented at a C++ standard committee meeting in 1994: `http://www.erwin-unruh.de`

[2]Theoretically because of limited compilers, such as limited recursion depths.

discusses and demonstrates the utilization of template meta programming to create numerical algorithms independent on data container types.

Meta programming is based on template specializations. *Listing 3.1* shows a simple example how a template function and specialization can be used to return different strings dependent on a type.

Listing 3.1: Simple example of using template specialization to create `string`s dependent on the template class type `T`.

```
1  template<class T> string getTypeString()
2  {
3      return "Unknown";
4  }
5
6  class MyType{ /*...*/ };
7
8  template<> string getTypeString<MyType>()
9  {
10     return "MyType";
11 }
12
13 /* ... in the main program ... */
14
15 string a = getTypeString<Foo>();
16 string b = getTypeString<MyType>();
17
18 /*...*/
```

Line 1 defines the general template function that will be called with any type not specialized. A specialization of the custom type `MyType` is shown in *line 8*. In this example string `a` will contain `"Unknown"` and string `b` `"MyType"`.

The general template definition can be interpreted as a definition of an interface. It ensures a valid call of the function with any type. When using template classes or functions in such an interface defining way they are called *Type Traits* in terms of C++ meta programming.

Type Traits can be used to add functionality to certain class types without modifying the classes' own definitions. Thus, defined classes implemented in an external library can be equipped with functions in the own code, like described in *section 5.2.2*, where functions for converting from and to a `string` can be provided for any types by a *Type Trait*.

Functions can be gathered in template classed as well. Member functions defined in the general template class are then similar to *pure virtual* functions in classical object oriented programming.

This mechanism can be applied similar to polymorphism in classical ob-

ject oriented programming. The next source code listing,*listing 3.2*, shows an example. The "virtual" function is called from within another template class dependent on the template class type parameter.

Listing 3.2: Example of gathering encapsulated functionality of different types in a unified interface template class enhancing code re-usability. A new `Computer` doing some work can be add by introducing the according `Functor` template specialization.

```cpp
template <class T>
struct Functor
{
    void doIt()
    {
        cout << "Do_what?" << endl;
    };
};

template <>
struct Functor<LaptopType>
{
    void doIt()
    {
        doSomethingLaptopTypeSpecific();
    };
};

template <>
struct Functor<WorkstationType>
{
    void doIt()
    {
        doSomethingWorkstationTypeSpecific();
    };
};

template<class T>
struct Computer
{
/* ... */
    void work()
    {
/* ... */
        Functor<T> Operator;
        Operator.doIt();
    }
};

/* ... */
```

```
41
42  Computer<LaptopType> LT;
43  Computer<WorkstationType> WT;
44  −
45      LT. work ( ) ;
46      WT. work ( ) ;
```

Here, the template class `Functor` defines an interface by its member function `doIt`, *line 1*. Two specializations operating on different types encapsulate its functionality in their functions, *line 10* and *line 19*. From within the `work` function of the template class `Computer` the `Functor` is then used to do the computation dependent on the template parameter `T`. Thus, the two different computers declared in *line 41* can operate using the same function interface call `work`, *line 45*

I used a similar technique to implement different line integration algorithms for different types of integral lines, see *section 6.2*. The core step integration is provided by template specializations, while the main loop and the code for reading and writing data is completely reused. I implemented a *KDTree* using a template callback structure. This made the search algorithm independent on the result data container. Any data structure can be filled with the results. Just a few lines providing a template specialization have to be added, see *section 5.4.2*.

The source code for templates is expanded, inserted and possibly inlined during compilation when being declared. Thus, the compiler can create highly efficient code. Such meta-programming provides coding flexibility and can reduce lines of codes tremendously when applied thoughtfully.

A good guide to C++ meta programming can be found in [58].

3.2   STL Encapsulation

Container classes of the standard template library (STL) [52] are utilized especially in the *Fiber Bundle* library, *chapter 4*. Though, the STL provides a beautiful concept to iterate over its template container classes, these iterators fail when classes are implemented over different dynamically linkable objects, such as *dll*s. Microsoft operating systems do not allow STL objects to be passed across library boundaries due to memory allocation issues. This prevents their use in programming interfaces, [48].

To overcome this problem the *Fiber Bundle* library uses its own iterator classes based on callbacks and wrapping the STL iteration. Such iterators are, for example, used to iterate over *Fiber Slice*s, *Fiber Grid*s or field fragments. In all that cases an iterator base class declaring a virtual `apply` function is provided. The iterator base class has to be derived and implemented and can then be utilized by calling the according `iterate` function. A simple example on how to iterate over field fragments is given in *listing 4.3*.

3.3   Reference Pointers

Smart pointers are a basic design element in modern C++ software development. They take care about their memory deallocation themselves. Native pointers in C++ are not exception save. Smart pointers solve this issue. Reference counting pointers are smart pointers that keep track of the number of references, pointers or handles to a resource. They are a typical approach.

Utilizing reference counting pointers reduces misuses and errors in memory management of the software. They can be implemented using template programming. Operator overloading enables a syntax similar to programming with native pointers.

The software libraries described in *chapter 4* and *chapter 5* are based on a memory core library, `MemCore`, providing strong and weak reference counting pointers with the following features, see [6]:

1. reference counting: keep objects alive exactly as long as at least one pointer refers to it

2. weak and strong pointers:allow back links and circular references

3. support of the up casting operation, conversion from child class to base class (going "up" in the class hierarchy), like it is always possible with native C++ pointers

4. support of the downcasting operation: conversion from base class to child class (going "down" in the class hierarchy), possibly revealing a `NULL` pointer, similar to the `dynamic cast<>` pointer conversion in C++

5. constant objects

6. multiple inheritance

7. thread-safety

A strong pointer will keep an object alive, whereas a weak pointer only recognizes the death of an object. Strong and weak pointers are complementing concepts and allow circular references. In this implementation, a strong pointer is a weak pointer via inheritance relationships.

Any class can be enabled for reference counting by deriving from the `ReferenceBase` class:

```
template<class T>
class MyClass : public MemCore::ReferenceBase<MyClass<T> >
{
    RefPtr<HerClass<T> > element;

public:
    MyClass( RefPtr<HerClass<T> >& elementP )
    : element(elementP) {}

/* ... */

};

/* ... */
RefPtr< HerClass<int> > herFoo = new HerClass<int >();

RefPtr< MyClass<int> > myFoo = new MyClass<int >( herFoo );
```

A reference pointer object can now be created by using the template `RefPtr<>` and the standard `new`, *line 15* and *line 17*. Objects are efficiently passed by a reference of a reference pointer, *line 7* and *line 17*.

A `RefPtr<>` can be checked to be valid similar to a native C++ pointer. This concept is used throughout the implemented software for error checking:

```
RefPtr< MyClass<int> > foo = someContainer->getData();

if (!foo)
{
    puts("error");
```

```
6     return;
7 }
```

Chapter 4

Modeling of Scientific Data

Computational methods are used more and more in the scientific world. Huge amounts of data are produced every day. But as the number of applications increases also the diversity of data handling increases and often not much care is taken about data exchange-ability.

Modern research often forces to split huge tasks over several researchers or research groups that have to collaborate. Data diversity can be a big burden to overcome and many hours are spent on data conversion and handling instead of concentrating on real research problems.

However, there exist concepts that can be used to model any kind of scientific data transparently, efficiently and sustainable. Using and applying such a data model is of great advantage especially in collaborative research projects.

Already in 1989 David M. Butler introduced and suggested the concept fiber bundles to describe data in a unified way, see[19] and [18].

Inspired by this idea Werner Benger developed the *Fiber Bundle* library for his needs in visualizations stemming from numerical relativity [6]. Data in numerical relativity tend to be quite complex since they often are formulated in different coordinate systems on non trivial manifolds such as adaptively refining meshes. Data sets tend to be huge, with several hundreds of Terabytes of data produced. Numerical relativity is located on the edge of possibilities in super-computing technologies.

With this main application in mind he included information such as topology and coordinate representation in a strong and flexible systematic approach applicable to likely all data occurring in computational scientific domains.

4.1   Fiber Bundle Data Model

The *Fiber Bundle* is based on Butler's fiber bundle concept. Here, data is separated in base space and in fiber spaces where the base space describes the topological and geometrical data and the fibers the data on the geometry. The data model captures all the semantics of the data. A vector field is not just a collection of floating point numbers. Rather, it is an array of tangential vectors in a coordinate system representation on a certain topology of a geometrical object at a certain time. All this information is systematically organized to handle and also store data. The meta information is captured in a hierarchical layer approach. The actual data sets are hosted in so called data fields, that represent the *fibers* of the bundle.

The two main goals of the *Fiber Bundle* approach are [6]:

- abstract the geometrical description of spatial objects from their numerical representation in a specific coordinate

- abstract the physical computation domain from its underlying discretization scheme

This allows to formulate algorithms independent of the underlying grid objects. The line integration algorithms I developed in this thesis are basically formulated grid independently by utilizing the `LocalFromWorldPoint` class, see *chapter 5.4*. Still some algorithms cannot be formulated in such a way.

Also a mapping to the file format *F5* exists that is utilized by the *Fiber Bundle* library for reading and writing. An isolated C-library provides this functionality. The file format is based on HDF5[1] which is developed and supported by the NCSA[2] HDF5 group for over a decade now. It is stemming from high performance scientific computing applications and addresses many needs regarding fast and parallel data access or sustainable archiving, see [33]. For more in depth information about the *Fiber Bundle* data model, the library and the *F5* file format, see [6], [49] and [8].

I omitted the mathematical definitions of fiber bundles here because they might discourage and confuse. The concept itself is simple and straightforward. For the sake of completeness, I added the mathematical definitions to the appendix, see *appendix B*.

---

[1]Hierarchical Data Format Version 5

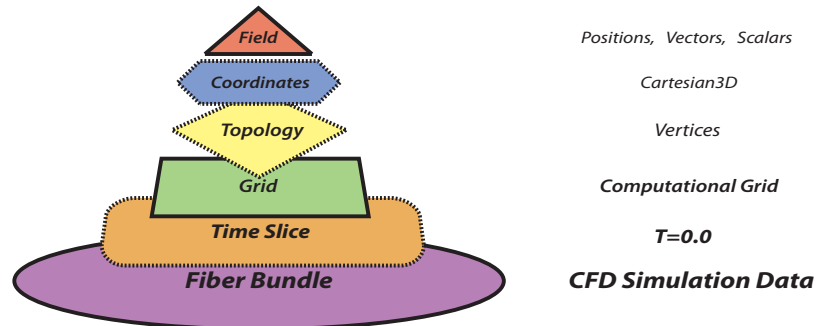[2]National Center for Supercomputing Applications

Figure 4.1: *Left:* The hierarchical layers of the *Fiber Bunlde* data model. Each layer can be compared to a directory in a file system, a container hosting several objects of the next hierarchy layer. Layers bordered by dashed lines are hidden to an application user completely and to a developer as far as as possible. *Right:* A simple example of a CFD data set. The positions of the computational grid are stored as a vertex field in Cartesian Coordinates. Two data fields (Vectors and Scalars) are stored on a grid object at time $T = 0$.

4.2   The Hierarchy Levels

To store data and its properties the *Fiber Bundle* library uses a hierarchical organization. Each layer adds information to the numerical data that is hosted on the end leaves, the data fields.

Figure 4.1 illustrates the hierarchical layers covering information regarding time, grid objects or manifolds, topology and refinement, coordinate representation and numerical data. The layers one mainly operates with are *Fiber Slice*, *Fiber Grid* and *Fiber Field*.

4.2.1   Fiber Bundle

Usually data can be separated in a spatial domain Σ and in one oriented time coordinate $\mathbb{R}$. The whole data space is thus covered by the Cartesian product of $(\Sigma \times \mathbb{R})$.

One time slice is a collection of all data related to one certain instance of time. The `Bundle` provides a map, implemented using a STL `std::map<double, RefPtr<Slice>>`, from a scalar time value to a time slice object. In the *Fiber Bundle* library a slice can be retrieved, for example, by
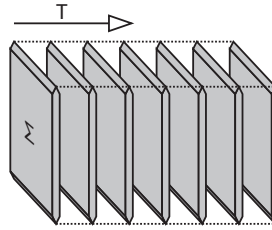
Figure 4.2: The collection of all time slices form a *Fiber Bundle.*

using the () operator of the `Bundle` class:

```
RefPtr<Slice> mySlice = myBundle( 0.553 );
```

Also, the [] operator can be used for access. In that case a new time slice will be created automatically when no slice at the given time is found.

The `Bundle` class provides a number of functions and convenient functions, for example, to retrieve the next and previous *Fiber Slice* or to extract a *Fiber Grid* object of the next or previous time slice.

In contrast to common praxis in many simulations floating point values are used to identify time slices instead of integer time steps. Float values capture time in a more natural way and allow to store, for example, datasets stemming from different time discretization in one bundle. This would be difficult if datasets are accessed by integer time steps but having different absolute time resolutions.

Generally, a slice does not need be time but could be a different floating point parameter describing something else. Even slicing using multiple parameters could be of interest, but is not yet supported by the library.

Figure 4.2 illustrates a physical space parameterized by a oriented time value. All time slices are collected in one bundle.

4.2.2 Fiber Slice

A Slice is a collection of geometrical objects called *Fiber Grid* objects. They are identified by a name. Thus, a slice provides a map of a string to a grid object.

Again, the () or [] operator is utilized to extract or create a grid object:

```
RefPtr<Grid> myGrid = mySlice("myName");
```

The slice provides an `GridIterator` that can be used to iterate over all or a subset of grid objects of a time slice. To utilize an iterator a `GridIterator`
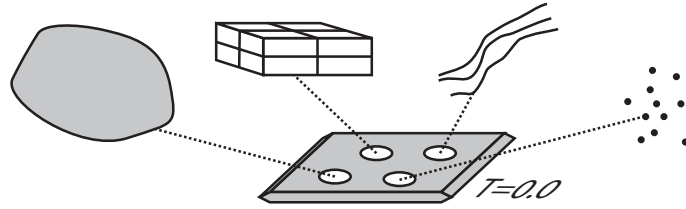
Figure 4.3: A *Fiber Slice* at $T = 0.0$ hosting four *Fiber Grid*s of different kind. Each grid is identified by a unique name.

has to be derived, its `apply` function has to be implemented and then iteration is invoked by, for example:

```
mySlice.iterate(myGridIterator);
```

Figure 4.3 illustrates the *Fiber Slice* container.

4.2.3   Fiber Grid

A *Fiber Grid* object represents geometrical object, a discretized manifold. All data describing such an object form a grid object. Typically, it will contain vertex positions, vertex connectivity and data sets defined, for example, at vertex positions, on edges, on faces or on cell volumes.

Figure 4.3 shows four different kinds of geometric objects: a general manifold, a $3D$ uniform grid, a collection of lines and a particle cloud.

To schematically distinguish between such different object types the *Fiber Grid* provides a map to `Skeleton`s describing the topology of one grid object (the next layer in the hierarchy).

The grid object is equipped with certain most frequently needed topology objects, such as, `Vertices`, `Connectivity`, `Edges` and `Faces`. Additional skeletons can be added.

Though, one can access the topology object via the grid object this is usually not necessary and is hidden to the end user of the library. In fact *Fiber Grid* objects are used to access and retrieve data fields directly by their names, for example:

```
RefPtr<Field> VectorField = myGrid("Velocity");
```

Convenience functions are provided the get and create Cartesian Representations or to directly extract Cartesian vertices of the grid:

```
RefPtr< Field > VertexField = myGrid.getCartesianPositions();
```
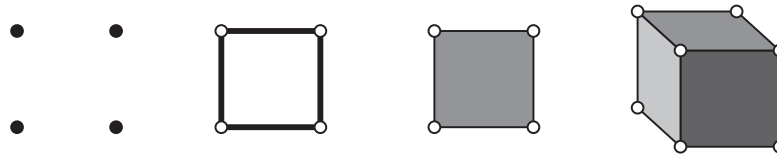
Figure 4.4: Dimensionality of some example topological objects. From left to right: Vertex: 0, Edge: 1, Quad-Face: 2, Cubic-Cell: 3.

4.2.4 Fiber Topology

The object describing the topology of the data is called a `Skeleton` in the *Fiber Bundle* library. It holds the *neighborhood* information and it is characterized by three integer numbers: the **dimensionality**, the **index depth** and the **refinement level**.

The neighborhood information is in the most general case stored as a list of neighbors of one spatial elements to the others as a list of indices. In many cases the neighborhood can be expressed procedurally and no explicit data has to be stored, as, for example, neighbors of vertices of a uniform grid.

The *dimensionality* describes the dimensions that the data, stored in the data fields, is associated with. A dimensionality of 0 represents a vertex, 1 represents a line or an egde, 2 a surface or polygon, 3 a volume or cell and 4 a hyper-volume, and so forth. *Figure 4.4* illustrates the dimensionality up to 3.

If, for example, scalar values are stored on $3D$ cell volumes the scalar data field would be nested in a `Skeleton` of dimensionality 3. The data field describing the vertices of the cell corners would be stored in a `Skeleton` of dimensionality 0.

Index depth describes how many "dereferencing" or "lookup" operations are necessary to get "down" to a vertex of the geometrical object. An edge, for example, can be defined as a pair of two vertices (or pair of indices of vertices), resulting in an index depth on 1. A path can be defined based on these edges. Now, to get "down" to the vertex level one has to first lookup the edge and then lookup the vertex. Thus, a path, defined via edges has an index depth of 2. The following table illustrates several possible values for skeleton characteristics by example (taken from [6]):

Figure 4.5: Uniform grid having different refinement levels.

Topological entity	Index Depth	Dimensionality
Vertex	0	0
Edge	1	1
Face	1	2
3D-Cell	1	3
Collection of Vertices	1	0
Path of Edges	2	1
Surface of Faces	2	2
3D-Cell Complex	2	3
Set of Cell Complexes	3	3

Another property is also defines the skeleton: the **refinement level**. *Figure 4.5* illustrates an uniform grid of three different refinement levels. Index depth and dimensionality, of course, are not changing on different refinement levels.

The *Fiber Bundle* library provides a convenience function for the creation of vertex topologies, since they are needed for all grid objects, on the grid layer:

```
Skeleton mySkeleton = myGrid.makeVertices(Dims);
```

4.2.5   Fiber Representation

The next layer describes the coordinate representation of the data. It either describes the coordinate system that was used to represent some vertices or field data, or it specifies a *relative representation*, for example, when faces are represented as vertices (or vertex indices).

The representation holds a set of *Fiber Field* objects which finally store the actual numerical data. Fields are accessed by the `FieldID` which is basically a name. One special field, the "Positions" field, is always included in a representation. This field defines the locations stored in this representation. If other fields are stored as well, these represent the data on the "Positions".
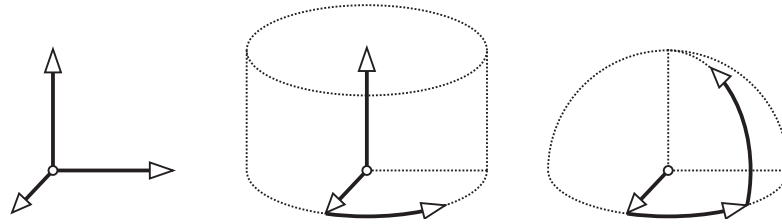
Figure 4.6: Examples of different coordinate systems that could be used to describe some $3D$ data. In terms of the *Fiber Bundle* library these are called `Chart`s. From left to right: Cartesian3D, Cylindrical3D, Spherical3D.

The *Fiber Bundle* library allows to specify "Positions" explicitly by specifying coordinates for each spatial element (e.g. for curvilinear grids), or procedurally (e.g. for uniform grids). A uniform grid can be fully described by the location of the origin, the number of vertices in each dimension and a distance between vertices in each dimension.

> According to the **fiber bundle** theory, the ***Fiber Grid*** objects together with the **"Positions"** fields represent the ***base space*** of the manifold. The **additional data fields** represent the ***fibers*** on that base space.

4.2.6   Fiber Field

A *Fiber Field* finally is the hierarchical layer storing the numerical data which is stored in data arrays. Thus, the field provides a map from an index to a data element.

All fields in one representation share the same index space which is based on the "Positions" field. Thus, if one representation holds the "Positions" field, a scalar field and a vector field, the same index is used to access the position, the scalar value at that position and the vector at that position used on the three different fields.

Having the actual numerical data organized in such a compact array form is very practical in visualization applications. Modern architecture of graphics hardware requires compact arrays for the description of geometry and texture data. *Fiber Field* arrays often can be loaded directly onto the graphics hardware without memory reorganization.
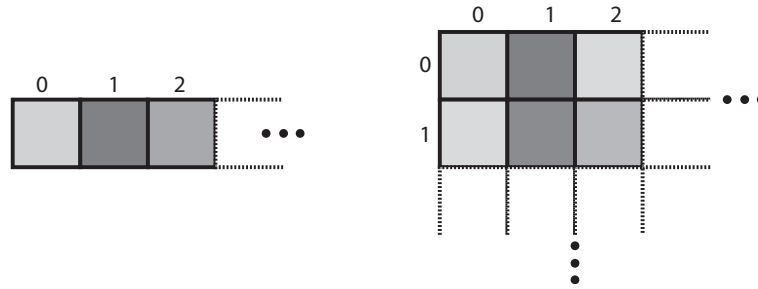
Figure 4.7: Final data storage in a *Fiber Field* as data array. One or more dimensional arrays can be used.

However, instead of hosting one data array a fields also can be fragmented. Such fragmentation allows, for example, to model multi-block data. Data arrays are contained in fragments instead of being stored in the *Fiber Field* directly. Fragments are then accessed by fragment IDs which basically are names.

The *Fiber Field* also provides a fragment iterator. When no fragments are present the call back operations executed with a field iterator will be applied to the data in the un-fragmented field. Thus, when implementing an iterator, the simpler case of an un-fragmented field is automatically covered.

To extract a data array of a field the `getData` function is called. The *Fiber Bundle* library uses some template base classes to handle arrays. They provide compile time range checking and a common interface for access. Access via the base class directly is possible but should be avoided since it is slow.

In my work I mostly use `MultiArrays`[3] of one or three dimensions to access field arrays. Here follow two short examples showing data array extraction from a *Fiber Field*. Some error checking is indicated:

Listing 4.1: Extracting a one dimensional `MultiArray` of cartesian positions of a *Fiber Field*.

```
1  RefPtr<Field> Positions = BaseGrid->getCartesianPositions();
2      if(!Positions) { /* some error handling */ }
3
4  RefPtr<MemBase> mBase = PositionsField->getData();
5
6  RefPtr<MemArray<1, point> >Vertices = mBase;
7      if(!Vertices)
8      {
```

---

[3]derived from `MemArray`

```
9        puts("Error:_MemArray<1,point>_is_incompatible!_Type_is:");
10           mBase.speak("some_text");
11        return;
12    }
13
14 MultiArray<1, point>&BaseCrds = *BaseVertices;
15
16 MultiIndex<1> mi;
17    mi=0;
18
19    point myPoint = BaseCrds[mi];
```

First the field is extracted from the underlying grid object by using a convenience function, *line 1*. Then data is retrieved into a array class `MemArray` of dimension 1 and type of `points`. When no field or no data can be retrieved the according reference pointer is null and can be used for error checking. When the `MemBase` is retrieved first it can be used to get information about the data object in case of an error, lne10. Finally, a reference of `MultiArray` is instantiated downcasting the dereferenced `MemArray`, *line 14*. The `MultiArray` can be accessed using a one dimensional `MultiIndex` efficiently.

Here follows an example that shows the extraction of a three dimensional vector array from a fragmented field by fragment name. Error checking is omitted:

Listing 4.2: Extracting a three dimensional `MultiArray` of vectors of one fragment of a *Fiber Field*.

```
1 RefPtr<Field>     VecField   = FieldSelection.getField();
2
3 RefPtr<FragmentID> FragID = new FragmentID( FragName );
4
5 RefPtr<MemArray<3,tvector>> VData =
6                        VecField->getCreator(FragID)->create();
7
8 MultiArray<3, tvector>&VCrds = *VData;
9
10 MultiIndex<3> mi;
11    mi=0,0,0;
12
13    tvector myVector = VCrds[mi];
```

Here, the field is retrieved from a `FieldSelector`, see *chapter 4.3*. The three dimensional `MemArray` of vectors is initialized by a creator of the fragment, *line 5*. Again, a `MultiArray` is used for the direct array access. The `MultiIndex` is also three dimensional in this case.

The next example illustrated how to utilize an iterator over fragments. The iteration over grids for example, follows the same concept:

Listing 4.3: Applying an iterator over fragments of a *Fiber Bundle Field*

```
struct   MyIterator : Field::Iterator
{
typedef DirectProductMemArray<point> ProcArray_t;
RefPtr<Field> myItField;

    /* store something */

    MyIterator::Myiterator( const RefPtr<Field>&myItFieldP )
    : myField(myItFieldP)
    {}

override bool apply( const RefPtr<FragmentID>&f,
          const MemCore::StrongPtr<Fiber::CreativeArrayBase>&DC )
    {
    RefPtr<ProcArray_t> PCrds;
        PCrds = Coords->getData(f);

        /* do something */
    }
};

MyItertor doSomething(myField);

myField->iterate(doSomething);
```

The `MyIterator` is applied on the fragments of `myField`. The apply function of the `Iterator` base class is implemented providing the current `FragmentID`. In the apply function the actual array is retrieved using the `Field`'s `getData` function with the `FragmentID` as a parameter, *line 16*. This time not a `MultiArray` is retrieved but a `DirectProductMemArray`, which is used to describe vertex data on uniform grids procedurally.[4]

4.3 Simplified Access via Selectors

The organization of the *Fiber Bundle* hierarchy basically provides mappings in one direction. It is straightforward to extract a field from a given grid. But the other way around would require a search in the bundle to find the according grid to a field.

---

[4]A uniform grid can be described by its origin, the axis distances between vertices and its size.

In the *Vish* visualization environment, see *chapter 5*, that heavily utilized the *Fiber Bundle* library sometimes fiber accesses were difficult. Selector classes were introduced for convenience that would store much more related information. Two selector classes are available: a `GridSelector` and a `FieldSelector`, which is derived from the former.

The `GridSelector` class holds the name of the selected grid and a handle to the bundle it is hosted in. It provides functions to extract grid objects that are closest to a given time and functions that extract grid objects next or previous to a given time. Having a `GridSelector` available the grid at a current time t can be extracted via:

```
RefPtr<Grid> myGrid = myGridSelector->findMostRecentGrid( t );
```

The next or previous grid can be extracted by:

```
RefPtr<Grid> myGrid= myGridSelector->findNext( t );
```

```
RefPtr<Grid> myGrid= myGridSelector->findPrev( t );
```

Since the `FieldSelector` is derived from `GridSelector` it provides the functionality described above and additionally allows to extract the selected field, the grid and the time slice carrying the field:

```
RefPtr<Field> myField = myFieldSelector->getField()
```

```
RefPtr<Grid> myGrid = myFieldSelector->getGrid()
```

```
RefPtr<Slice> mySlice = myFieldSelector->getSlice()
```

Another convenience function returns the "Positions" field according to the selected data field, if they are given in Cartesian representation:

```
RefPtr<Field> myPositions = getCartesianPositions();
```

More convenience functions are available and can be found in the documentation, see *chapter 5.1.2*.

The selector classes are now used as data handles (for example as module parameters) in the *Vish* environment and simplify the fiber data access tremendously.

4.4 Data Examples

Three examples of how data is laid out in the *Fiber Bundle* model are shown in this section. The examples describe most of the data I used in the thesis:

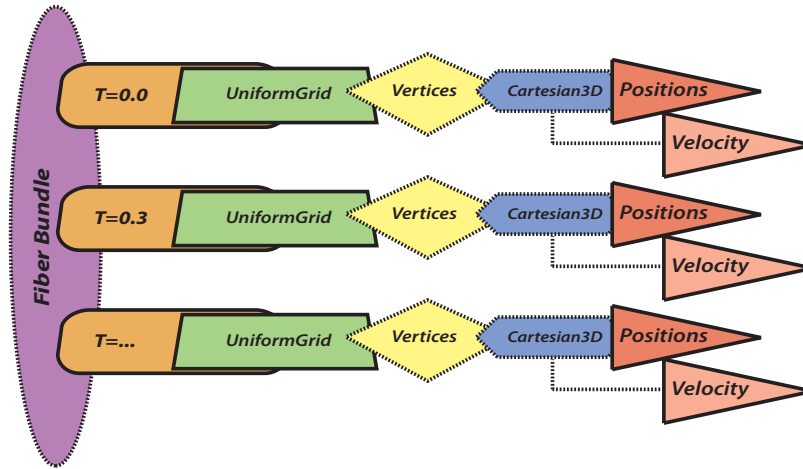- a uniform grid hosting vector, scalar or tensor field data

Figure 4.8: Structure of *Fiber Bundle* layers of the uniform grid hosting a vector field for velocity, as it is used to describe the Couette flow field in *chapter 7.1*. Here, the light red color of the "Velocity" data field indicates the fiber on the manifold. The two data fields (red) are contained in the `Cartesion3D` representation.

- a curvilinear multi-block grid hosting vector and scalar data

- a grid that describes lines with additional data stored on them

4.4.1 Uniform, procedural

Many data are provided on a uniform hexahedral procedural grid formulated in Cartesian coordinates. The $3D$ vector field of the Couette flow application and the metric tensor fields are created by `Analytic Creators` that sample an uniform grid based on some formula. Data is sampled on demand by the creator when it is requested. The diffusion tensor field used in the MRI application is read from numerical data, but is also hosted by a uniform procedural grid, see *chapter 7*.

Figure 4.8 shows the hierarchical layout of the vector field used in the Couette flow application, see *chapter 7.1*. It provides a vector field on a uniform grid. The original stationary flow is scaled over time, thus, several time slices are stored in the bundle. In this application all field data is computed on demand, stored and kept in the bundle, just before data is accessed the first time.

The uniform grid (the base space) itself does not change over time. The *Fiber Bundle* library will just reuse the same grid object for all time steps in
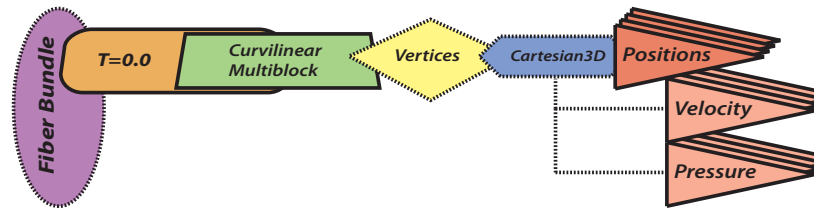
Figure 4.9: Structure of *Fiber Bundle* layers of the curvilinear multi-block grid hosting a vector field for velocity and a scalar field for pressure, as it is used to describe data for the stirtank in *chapter 7.2*. Again, the light red colors illustrate the fibers on the manifold sharing the index space of the "Position".

that case, though this is not visible in the data layout. The only component that is changing over time is the vector field describing velocity (the fiber).

The data layout itself does not differ between grids given procedurally or explicitly. The difference is located in the array data structure used to vertex "Positions" data. As described above a `DirectProductMemArray` is used to store procedural coordinates. The coordinate array of a procedural uniform grid can be extracted using the `getData()` function of a field, for example:

```
1 typedef DirectProductMemArray<point> ProcArray_t;
2 RefPtr<ProcArray_t> ProcCrds = myPositionField->getData();
```

4.4.2   Multiblock, Curvilinear

The data from the stir tank application, see *chapter 7.2*, is given in curvilinear multi-block data. Coordinates of the grid object are given explicitly and are separated over multiple blocks of coordinates. The coordinates are given on a hexahedral grid.

The *Fiber Bundle* data model captures the multi-block organization by storing its field as fragmented fields. Each fragment refers to one multi-block.

Figure 4.9 illustrated the hierarchical layout. The differences to the uniform grid example are, that data fields are stored using `MultiArray`s, for explicit coordinates, and that data fields are fragmented. The data set holds a vector field for velocity and a scalar field for pressure, stored as fibers on the grid object. Besides the index space now also the space of fragment IDs is shared among the fibers hosted on the vertex "Positions".

Fields must be accessed using fragment IDs or via fragment iterators as described above.
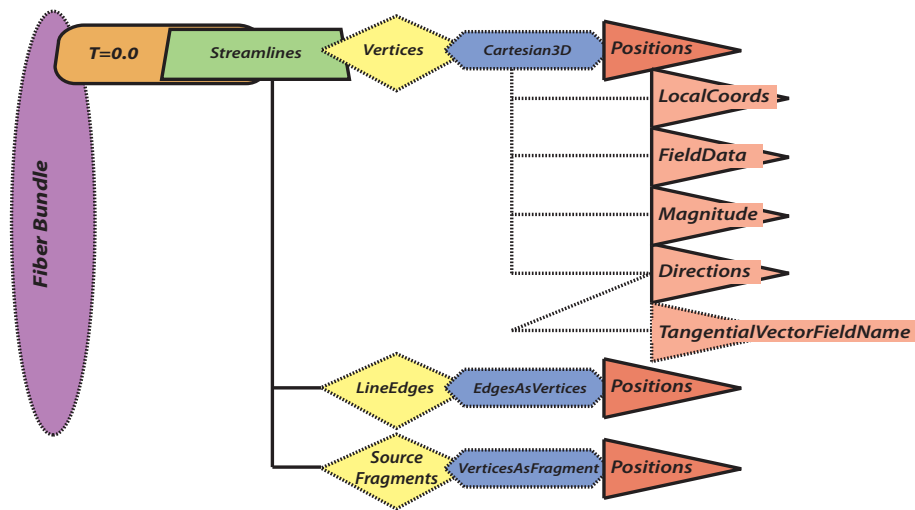
Figure 4.10: Structure of *Fiber Bundle* layers of the grid describing computed integral lines as they are used throughout all applications chapters. The light red colors illustrate the fibers on the manifold sharing the index space of the "Position". Here, data fields are stored on the vertices of the lines. The edges of the lines are stored in its own topology using a relative representation on the `Vertices`, thus, using vertex indices to describe edges. An additional topology `SourceFragments` stores multi-block IDs relatively to the vertices. The fiber field `TangentialVectorFieldName` is an alias to `Directions` used as convention for rendering modules in *Vish*.

4.4.3   Lines

The lines stemming from the integration, see *chapter 6*, are also stored using the *Fiber Bundle* data model.

This example shows the application of relative representations. First, the lines store information "what make them lines". Vertex positions are stored in the topology `Vertices`. Now, the `LineEdge` topology (index depth 1, dimensionality 1) defines edges by using the relative representation `EdgesAsVertices`. Thus, vertex indices are used to define edges. Tangential direction vectors are computed during integration and are stored in the data field `Directions`. To provide compatibility to line grids in the *Vish* environment an alias `TangentialVectorFieldName` is provided, as well.

Next, data related to the data field that is being integrated is stored. The data field value at the vertex position is stored in the fiber `FieldData`. In case of streamlines this is the velocity vector and in case of geodesics this is the metric tensor. A norm of this object is stored in the scalar field

`Magnitude`.

Finally, data that will speed up sampling of any other field on the line grid is stored, i.e. interpolation weights. This information includes the local index coordinates of a vertex, see *chapter 5.4*, and the IDs of the multi-block, a vertex is contained in. Local coordinates are stored as a fiber in the vertex topology and fragments (or better fragment IDs) are stored on a different topology (index depth 1 and dimensionality 3) using a relative representation on the vertices.

Example code illustrating data storage using the *Fiber Bundle* library is given in *chapter 6.2*.

Chapter 5

Vish - The Vis(h)ualization Environment

Vish is the visualization application of choice for this thesis. It it a very modular, flexible and fast framework especially build for scientific visualization created mainly by Werner Benger. It provides the programmer with a lot of functionality addressing problems especially faced when doing scientific visualization, such as, handling large data sets and memory management. It provides an automatically generated GUI, 3D rendering with camera and time navigation, hierarchical caching, mathematical functionality and a number of standard modules for analyzing datasets.

It is meant to enable the developer to focus on visualization algorithms and not spending his and her time on basic tasks like GUI programming or reading and writing data files. This comes, of course, with the cost of learning and understanding some concepts of the visualization environment. It follows a systematic approach using strong theoretical concepts in the background. This helps to create overall more reusable code. These concepts are:

- Topology

- Differential Geometry

- Geometric Algebra

This chapter should help to overcome this hurdle and will give a quick start. Since most of this information is available already on some web documentation or tutorial examples it is kept short and will give some references to further reading. While basics are described here shortly, more in-depth examples will be explained by describing parts of the source code written for computing the geodesics in the following in *chapter 6*. Also, see [11] for more descriptions of concepts behind *Vish*.
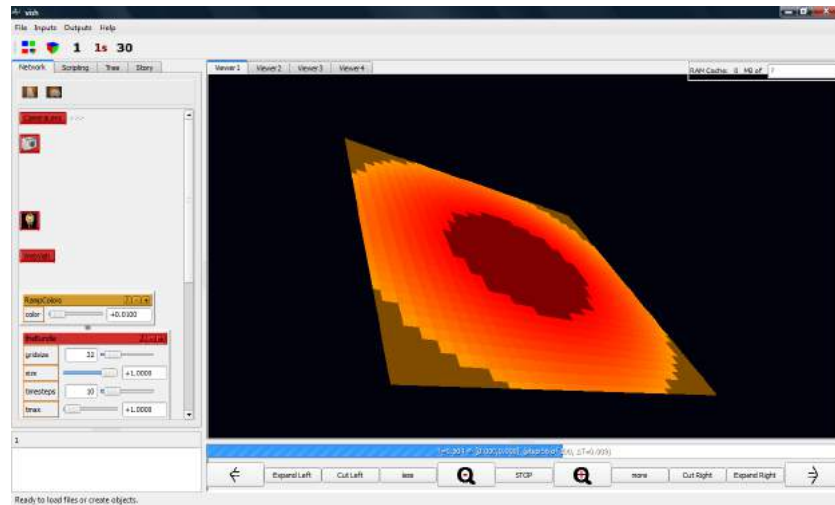
Figure 5.1: The GUI of *Vish*. It consists of a 3D graphics view port(right), a time line(bottom right) and an area where parameters can be controlled(left), the network frame.

5.1 Development Quick Start

5.1.1 Availability and Installation

The source code of *Vish* is available under the *light++* license, see [7], and is free for personal and academic use via SVN[1] from `http://vish.origo.ethz.ch`.

A windows executable to automatically install a windows build of the application and tools needed for development can also be found at [21]. Here, several information and links are collected, including: mailing list access, the *Vish* and the *Fiber-Vish* API[2] created with doxygen[3], a small programming tutorial and a gallery.

The installer, however, does not include the *Vish* source, but automatically downloads tools and libraries needed for development. Thus, a SVN checkout is still required afterwards.

For Linux64 bit a self extracting binary is available. For development also a SVN checkout is required. For usage and development under Mac OS more work has to be done manually.

---

[1](**S**ub)**V**ersion **C**ontrol

[2]**A**pplication **P**rogramming **I**nterface

[3]A tool for creating documentation by parsing and analyzing the source code directly, see [57]

Dependencies on external libraries are kept as minimal as possible. As of now, the following external libraries are required:

- QT, a cross platform GUI library, see [45]

- GNU Bison and Flex, a GNU parser generator and lexer, from [30]

- GNU FreeType, a GNU font library

- HDF5, a hierarchical file format, see [33]

Earlier the HDF5 file format library had to be downloaded manually but it was added to the *Vish* SVN repository, lately.

To get access to the SVN one must join the vish group and, thus, must be added to the group to get full permissions.

Installation instructions can be found at origao, see [64], under `development`. When using windows it is best to install using the installer provided at [21] and to install all libraries or programs into the standard directory that is suggested by each installer. Otherwise it might get necessary to modify some `Makefiles` during compilation.

5.1.2 Source Code Organization and Naming Conventions

Vish follows an own naming convention through directory, file and class names. The *Vish* kernel is called the `ocean` which represents the environment inhabiting classified beings.

The `ocean` has interfaces to four big code areas which are connected to the kernel only:

- GUI

- Script

- Graphics

- Data I/O

The preferred components are the Qt library for the GUI, a simple script interpreter, OpenGL for Graphics and HDF5 for Data I/O. They can be exchanged by using the provided interfaces.

The more high level the code in the kernel gets the higher level is the classification of the being living in the `ocean`. Starting from the `$VISH/ocean/plankton` library and becoming more complex via e.g.

`$VISH/ocean/shrimp` to `$VISH/ocean/eagle`, which for example provides mathematical, algebraic classes. Typically a being consists of several API source files. If there are concrete *Vish modules, section 5.2*, implemented based on the contained source files the directory will contain a directory called `egg` hosting these implementations.

A complex being is the `$VISH/fish/fiber` library. This is the *Fiber Bundle* library as described in *chapter 4*, which is independent. All source code dependent on the `$VISH/ocean` and the `$VISH/fish/fiber` library are gathered in the `$VISH/fish/lakeview` directory. Here, almost all *Vish Modules* can be found.

When developing a new *Vish Module* without extending the *Vish* kernel this most likely the directory where the source code will be placed.

Both, the `$VISH/ocean` and the `$VISH/fish` library use the `$VISH/memcore` library which provides basis classes for memory management, such as, template classes for reference pointers. Native pointers are avoided in source code and strong and weak reference pointers are preferred, *section 3.3*. This reduces programming errors due to allocation and freeing of memory.

The *Vish* coding style usually intends that code is separated in header (`*.hpp`) and implementation(`*.cpp`) files. Usually every class is placed in an own file which has the same name as the class. For example, an class implementation called `RandomPointDistribution` is placed in the file or files `RandomPointsDistribution.cpp` and/or `RandomPointDistribution.hpp`.

Classes or source files are kept short. In the best case, files will have less than 300 lines of code. Up to 800 is considered as acceptable but having more lines of code should give the programmer a hint for refactoring and clearer organization the code. Maybe some functionality can be gathered and made better reusable in another header file.

Experimental test or source code should be placed in `$VISH/modules/examples` where some examples of implementations of different kinds can be found.

While working into the software environments I created a quick start tutorial which starts from very a simple example. More complex source code examples demonstrate how to use OpenGL and the *Fiber Bundle* library. The tutorial can be found in the `$VISH/tutorial` directory.

All documentation is located in `$VISH/doc` but has to be created manually. There are different parts of the documentation that have to be created each. For creation of the *Vish* documentation invoke `make doc` in the `$VISH/` parent directory. For creation of the *Fiber Vish (Fish)* documentation invoke `make doc` in the `$VISH/fish` directory. `Make doc` in the `$VISH/tutorial` directory creates the documentation for the tutorial. These documentation can

also be found online at [21].

The directory `$VISH/data` is used to place f5 data files, see [49], or vis script files, *section 5.2.4*

5.1.3   Make Files and Compilation

The *Vish* project uses a make file system. The basis make file configurations for different platforms, like Linux 64, Windows 32, etc. are located in the `$VISH/make` directory. Internally the `uname` command is used to differentiate between platforms and the returned string of the command is concatenated to the `arch-` prefix.

Each directory containing source files contains a file `Makefile` specifying the following:
The name of the dynamically linkable object is defined by, for example, `VISH=fishcephalus`. During compilation a file `fishcephalus.dll` or `fishcephalus.so` is created and placed in the `$VISH/bin/$arch` directory containing the compiled code of the source of this directory.
Script files can be copied to the `$VISH/data` directory by specifying `VIS=` followed by a list of `*.vis` files separated by a whitespace.
All libraries that are necessary for linking have to be specified by `LIBS=` followed by strings '-l' concatenated with the name of the library separated by a whitespace.
If `OBJS=` is specified the following list of `.cpp` files are used for compilation. If not specified all `.cpp` files in the directory are compiled.
The `Makefile` ends with `include $(VPATH)../GNUmakefile.rules`. For a more detailed description of the make file system, see the documentation located at `$VISH/make/doc`. A typical `Makefile` is shown in *listing 5.1.*

Listing 5.1: Typical *Vish* `Makefile`. The example shows a simplified version of the `Makefile` found in the `$VISH/fish/lakeview/cephalus` directory

```
1  VISH=fishcephalus
2
3  VIS=GridSubtractor.vis  GridAdder.vis  GridConvolver.vis  ...
4
5  LIBS=-lbundle -lfiberbaseop -lfiberop -lshrimp -lplankton
6  $(FIBERLIBS) -lmemcore -lm -lfishbone  $(OCEAN)
7
8  include  $(VPATH)../GNUmakefile.rules
```

Compilation of the source is done using a `gcc`[4] compiler in the according system by invoking `make` in the parent *Vish* directory `$VISH/`. Parts of the

---

[4]Version 3.4.2 or newer.

Vish environment can be compiled by invoking make in the appropriate sub-directory or by providing the directory name after `make -C`.

To compile in debug mode use `make Make_CFG=Debug` and to compile optimized use `make MAKE_CFG=Optimize`.

`Make libclean` deletes all precompiled object files located in the `$VISH/lib` directory and `make clean` does a complete cleanup in the current directory.

5.2   Scene Network

To solve an entire visualization task *Vish* breaks this task into small pieces of work. An entity processing a small piece of task is called a **module**. Modules are connected by a directed graph or **network**. The graph can also allows cycles. Information is passed in two steps through the graph. A controls flow decides what has to be updated and the data flow transports the information.

It is a common approach to break tasks into such building blocks in graphic applications. Atomic tasks can be combined in many different ways to solve complex tasks making this a very flexible approach.

Famous examples of graphics applications that successfully use a network scheme are *Autodesk Maya* (a professional 3D application used in movie productions, see [3]), *Eyon Fusion* (a nonlinear compositing application used in movie productions, see [27]), *Avizo*, the former *Amira* (a scientific visualization toolkit, see [62]) and *OpenDX* (a scientific visualization toolkit, see [46]).

However, there are different schemes of how data is flowing through the network or how recomputing of modules is controlled. *Vish* uses a data pull scheme. Thus, the end or sink nodes of the network, which typically are modules responsible for rendering, notify the connected upstream modules that they need to provide up to date data. This request is forwarded to the next upstream connected modules and finally all involved modules are updated or recomputed from source to sink.

So, sink modules pull from the sources and updates are only done where they are really needed. *Maya*, for example, uses a similar technique, by first flagging necessary updates starting from the sink modules, see [55]. In contrast, *Avizo* uses a push scheme where data sources push the data through to the sink nodes.

The following section explains how the existing *Vish* network capabilities can be extended by introducing new modules and new data types for the data transport. It also introduces data access using the *Fiber Bundle* library and describes how rendering is done. Finally, the script language to store networks in text files is presented.
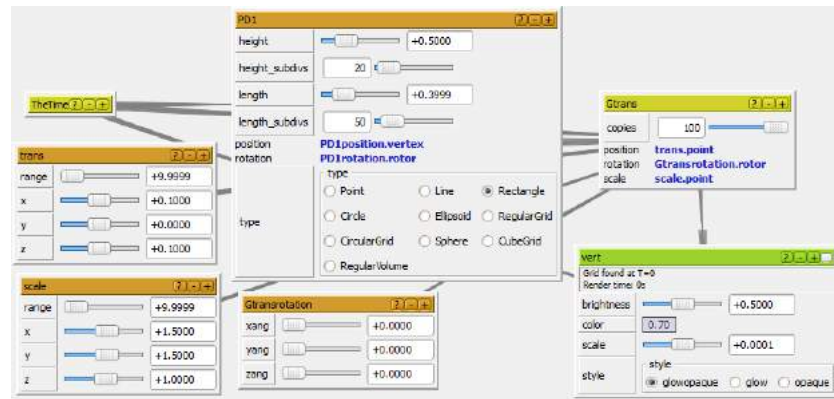
Figure 5.2: Example of a module network shown in the network frame of *Vish*'s GUI. Modules can be created by the user via a context menu showing a hierarchical list of all modules which appears when clicking on an empty space in the network frame, see *figure 5.1*. When doing a right click on an existing module a context menu appears containing modules that have compatible input parameters to the output parameter of the right clicked module. A created module then is automatically connected.

5.2.1 Modules

Modules are the building block of the *Vish* network. They are derived from the base class `VObject` and have to overload the function `bool update(VRequest&R, double precision)`. The module usually is equipped with input and output parameters. When a module is requested to update[5] it executes the `update` function and typically recomputes the output parameters dependent on the input parameters.

An input parameter can be added by declaring the template class `TypedSlot< ParameterType >` in the derived `VObject`. In the constructor of the class the parameter is initialized and also properties for a parameter can be set.

In the same way, an output can be added by declaring the template class `VOutput< ParameterType >`. Initialization and configuration also is done in the constructor.

Examples of supported parameter types are `int`, `double`, `Enum`, `Eagle::PhysicalSpace::point`, `Eagle::PhysicalSpace::tvector`, `Eagle::PhysicalSpace::metric33`, `Eagle::rgba_float_t`. Custom types can be added, as described in *section 5.2.2*.

---

[5]An internal age parameter is used for that purpose.

To ensure thread save behavior it is forbidden to store data in the module by using a typical data type in the module class. Data is always dependent on a context. So, also when accessing input and output parameters this is always done via the context. The context is provided in the `update` function via the `VRequest&R` object.

Listing 5.2 shows how to implement a simple *Vish* module. It has two input and one output parameter of type `int`. In the update function the output is computed dependent on the input doing a multiplication. So, the module just multiplies two integer numbers and outputs its result.

Listing 5.2: Source code example of a simple *Vish* module. The `MultiPlyInt` example class can also be found in the tutorial section at `/tutorial/basic/MultiplyInt.cpp`

```cpp
#include <ocean/plankton/VCreator.hpp>
#define override

using namespace Wizt;

class MultiplyInt : public VObject
{
    TypedSlot<int> IntInParam;
    TypedSlot<int> Multiplier;
    VOutput<int>   IntOutParam;

public:
    MultiplyInt(const string & name, int p,
                const RefPtr< VCreationPreferences > & VP)
    : VObject( name, p, VP )
    , IntInParam(   this , "intinput", 0 )
    , Multiplier(   this , "multiplier", 2   )
    , IntOutParam( self(), "intoutput", 1  )
    {
        Multiplier.setProperty("max", 8);
    }

    override bool update( VRequest & R, double precision )
    {
    int x = 0;
    int m = 0;

        IntInParam << R >> x;
        Multiplier << R >> m;

        x *= m;

        IntOutParam << R << x;

```

Figure 5.3: Example of a module shown on different expert levels. When the module is displayed in the GUI parameters with a high parameter value are hidden. The expert level that should be shown can be increased by clicking the + button in the header of the module. This is used to hide parameters from the user that are not needed frequently. The expert level increases from left to right.

```
35          return true;
36      }
37  };
38
39  static Ref< VCreator<MultiplyInt, AcceptList<int> > >
40          VMultiplyIntCreator( "Tutorial/MultiplyInt",
41                               ObjectQuality::EXPERIMENTAL);
42
43  VISH_DEFAULT_INIT
```

The macro in *line 2* is used to label function if they are overloading a pure virtual function. This makes code better readable and understandable and is used throughout the most parts of source code in the *Vish* environment. Thus, the `update` function in *line 23* is flagged since it is an overriding function of `VObject`.

The module `MultiplyInt` is derived from `VObject` making it a *Vish* network module. The module is equipped with one output and two input integer parameters, *line 8* to *line 10*.

The parameters of the constructor, *line 13*, are just handled to the super class, see *line 15*. The `TypedSlot< >` parameters are initialized by using the constructor syntax with the following parameters: `this`, a string specifying the name of the parameter in the GUI and script, the initial value and optionally a so called *expert level*.

The body of the constructor is used to set properties of parameters. Here, a maximum value for the `Multiplier` parameter is set to `8`, see *line 21*.

Finally, the `update` function is overridden. To get the values of the input into local types the context `R` has to be used. *Line 29* and *line 30* shows how this is done using the `<<` operator to read from the context in combination with the `>>` operator to write to the local type. Now, the local types can be used in computations, as in *line 32*.

The value of a local type is written to the output parameter in a similar fashion by using the `>>` operator via the context `R`.

The module itself is ready for use now but a `static VCreator` object has to be provided for *Vish*. It defines what types can be connected to the module and specifies a name and place in the GUI context menu for the module creation. The first template parameter defines the class that is created, the second the a list of compatible input parameter types. The `AcceptList< >` template can be nested when more than one type is required. The name of the `VCreator` class is arbitrary but the parameter string is important and specifies the name of the module in the creation context menu. It is also the string used to create a module instance in the `*.vis` script language, *section 5.2.4*. The second parameter sets a category used to filter modules in the GUI menu. Possible values are `ObjectQuality: :RECOMMENDED`, `:MATURE`, `:DEMO`, `:BETA`, `:EXPERIMENTAL`, `:TEMPORARY` and `:OUTDATED`.

If necessary, data can be stored or cached in the module from one `update` call to another by using a provided `state` object. To add such a state class a nested state class has to be defined in the module class and a creator function has to be overridden, see *listing 5.3*.

Listing 5.3: Adding a state class to remember data from previous `update` calls.

```
1  class MultiplyInt : public VObject
2                    , public StatusIndicator
3  {
4  /* ... */
5      struct   MyState : State
6      {
7          int remember_value;
8      };
9
10     override RefPtr<State> newState() const
11     {
12         return new MyState();
13     }
14 /* ... */
15     override bool update( VRequest & R, double precision )
16     {
17 /* ... */
18          RefPtr<MyState>   state = myState(R);
```

```
19
20          state −>remember_value = x ;
21
22          setStatusInfo (R,  "Computation_done ! " ) ;
23 /*  . . .  */
24      }
25 } ;
```

Now, `MyState` can be used to store data. The simplified example makes no real sense here. It just shows the correct procedure to add and access data via a state object. Also, other possibilities for caching data are provided, *section 5.3*.

A number of super classes for multiple derivation are provided for `VObjects`. Features such as printing text messages in the GUI can be added by deriving from the `StatusIndicator` class, see *listing 5.3*. Now, the functions `setStatusError` and `setStatusInfo` can be used in the `update` function for printing strings, see *line 22*.

5.2.2   Data Transport and Access

Creating new Parameter Types

New valid parameter types for the `TypedSlot< >` template can be defined. Firstly, the new class type has to be defined and secondly new specialization of the template class `VValueTrait< >` has to be provided of the new custom type. The `VValuetrait< >` contains two functions that are used to convert the custom type to and from string representations. *Listing 5.4* shows a simple example of enabling a type called `myNewType`.

The `VValuetrait< >` is a Type Trait and allows to equip any foreign types with the required functions, *section 3.1*.

Listing 5.4: Example of adding a custom parameter type.

```
1 struct myNewType
2 {
3          myDataClass data ;
4 } ;
5
6 namespace Wizt
7 {
8 template <>
9 class VValueTrait< myNewType >
10 {
11 public :
12     static   bool setValueFromText ( myNewType & i ,
13                                     const string & s)
```

```
14     {
15         return false ;
16     }
17
18     static   string Text(const   myNewType & e )
19     {
20     std :: string s ("");
21     std :: stringstream tmp ;
22
23         tmp << e . data ;
24         s = tmp . str ();
25     }
26             return s ;
27     }
28 };
29 }//namespace  Wizt
```

It is important to implement the `Text` function because this string representation is used in the *Vish* network to decide whether parameter values have changed or not. So this is important for achieving the correct update behavior of connected modules. Also, the string functions are used for loading and saving `*.vis` scripts, *section 5.2.4*.

Fiber Bundle Data Access

Vish also provides ready to use parameter types and classes for accessing and connecting data represented in the *Fiber Bundle* data model.

The module can be equipped with functionality to operate on a *Fiber Grid* and/or *Fiber Field* by deriving from `Fish<Grid>` and/or `FishField`. In that case, `TypedSlot<Grid>` and/or `TypedSlot<Field>` are inherited and neither must be added to the class nor must be initialized in the constructor of the module.

When equipping a module with such a *Fiber Field* or *Fiber Grid* it has to be derived also from `Fish<Slice>` first.

Listing 5.5: Example of equipping a module with input slots for one *Fiber Grid* and one *Fiber Field*. Note that all classes derived from the `Fish<>` interface are initialized by one line of code.

```
1 class DataCrunch :  virtual public VObject
2                   , virtual public Fish< Slice >
3                   , virtual public Fish< Grid >
4                   , virtual public Fish< Field >
5                   , public StatusIndicator
6 {
7 /*  ...  */
```

```
 8      public:
 9        DataCrunch(const string & name, int p,
10                     const RefPtr< VCreationPreferences > & VP)
11        : VObject( name, p, VP )
12        , Fish<VObject>(this)
13 /* ... */
14
15    override bool update( VRequest&R, double precision )
16    {
17 /* Example to get a Grid inherited from Fish<Grid> */
18    GridSelector GS1;
19        MyGrid  << R >> GS1;
20
21        Fiber::Bundle::GridInfo_t Grid1=findMostRecentGrid(GS1,R);
22
23    RefPtr<Grid> BaseGrid = Grid1;
24        if (!BaseGrid)
25        {
26            puts("DataCrunch::update() ERR, No parent grid found");
27            return setStatusError(R, "No parent grid found");
28        };
29
30 /* Example to get a Field inherited from Fish<Field> */
31    FieldSelector FS;
32            MyField << R >> FS;
33
34    std::string FieldName = FS.getFieldName();
35    RefPtr<Representation> vrep;
36    RefPtr<Field> WorkField;
37
38        vrep = SomeGrid->getCartesianRepresentation();
39        WorkField = (*vrep)(FieldName);
40        if(!WorkField)
41        {
42            puts("DataCrunch::update(): ERROR, No field found");
43            return setStatusError(R, "No field found");
44        }
45 /* ... */
46    }
47 };
```

Additional *Fiber Field* or *Fiber Grid* input parameters can be added by using `TypedSlot<Field>` or `TypedSlot<Grid>`. Similarly, output parameters can be added by `VOutput<Field>` or `VOutput<Grid>`. The initialization in the constructor then has to be done explicitly, for example:

```
, mySecondInputGrid(this, "another grid", GridSelector())
, mySecondInputField(this, "another field", FieldSelector())
, myOutputGrid(self(), "outgrid", GridSelector())
```

, myOutputField(self(), "outfield", FieldSelector())

Inside the `update` function *Fiber Grid*s and *Fieber Field*s can be accessed via the `Selector` classes, *section 4.3*.

Line 18 to *line 28* in *listing 5.5* shows how the get a reference pointer of a *Fiber Grid* object. First, a `GridSelector` is created and fed by the input `TypedSlot<Grid> MyGrid`. The selector is used to get the grid of the most recent `Fiber Slice` by using the `findMostRecentGrid` function providing the actual context. After assigning to the reference pointer of the *Fiber Grid* it is checked if it is a valid pointer and the `update` function returns in case of an error.

Line 31 to *line 44* in *listing 5.5* shows the extraction of a *Fiber Field* reference pointer. Similar to the grid process it uses a `FieldSelector` class. Note, that `FieldSelector` is derived from `GridSelector` and can also be used to extract a handle to the *Fiber Grid* which is hosting the data field, *section 4.3*(*line 39*). Here, a field using a Cartesian representation with the name extracted from the field input parameter is extracted and again tested for validity.

To output a *Fiber Grid* or *Fiber Field* an according selector class has to be created and fed into the `VOutput<>` slot. Here is a short example that shows how to output a grid:

```
1 GridSelector myGS( grid\_name, BPtr );
2 myGrid << R << myGS;
```

How *Fiber Grid*s and *Fiber Field*s are prepared and stored to the *Fiber Bundle* is demonstrated in the code excerpts of *chapter 6*.

5.2.3 Rendering Modules

Rendering modules are similar to basic modules. Additionally they provide a function for rendering an object. The rendering function is called by the renderer each time an update of the view is requested.

The class `VRenderObject` is derived from `VObject` and thus inherits all functionality as described in *section 5.2.1*. It provides a pure virtual function to be overridden for rendering. An OpenGL context and state is provided as well.

The `render` function of a `VRenderObject` is called from within the `environment_render`(inherited from `VEnvironmentRenderObject`) function which restores the OpenGL state afterwards.

It is important for the 3D scene and camera navigation to have an bounding information of the object to be rendered available.

VRenderObjects provides three functions for manipulating the bounding box of the object:

```
1 void resetBBox(const RefPtr<ValuePool>&VP) const
2 void embrace(const RefPtr<ValuePool>&VP, const point_t&crd) const
3 void closeBBox(const RefPtr<ValuePool>&VP) const
```

Before preparing the OpenGL data structures the bounding box is reset. When preparing the vertex data the **embrace** function can be used to grow and fit the bounding box. Finally, it has to be closed with **closeBBox**.

As the render function is called very often, like several times a second, it is a critic function and should be really fast. If object are not changing they most likely should be cached and not be created again at each **render** call.

The idea is to prepare the basic data in the **update** function. In the **render** function rendering objects loadable to the graphics card are created and cached to the graphics card and finally called for drawing.

Geometric Algebra

Scientific visualization and also computer graphics in general make use of lots of vector algebra. Usually linear algebra is used for that purpose. And also often just arrays of floating point numbers are used to represent vectors.

However, there are different kinds of vectors which are not compatible in calculations or have to be transformed differently. *Vish* differentiates between several kinds of vector types, like: **point**, **tvector**, **bivector**, **trivector**, etc. This is implemented using template classes, type definitions and operator overloading based on the one dimensional array template class **FixedArray<Type, Size>**.

For example, multiplying two vectors of type **point** is not allowed, whereas subtracting two **points** will yield a tangential vector **tvector**. This approach makes the source code more similar to writing mathematical notation and makes it more error prove as well. I believe that using higher level objects with included semantics for computation will avoid unnecessary errors.

Furthermore, this is also motivated by the theory of geometric algebra which provides a natural extension to arbitrary dimensions in vector calculus (which is, for example, not the case for the cross product in linear algebra[6]), see [10], also added to *appendix C*. Herein, examples are given that show the benefit of using geometric algebra: simpler calculation and shorter source code. For a basic introduction to geometric algebra see [61]. Further reading for application in physic related problems see [23].

---

[6]Which is well defined in $3D$ only.

OpenGL

OpenGL is a library for rendering 3D graphics in real time accelerated by modern graphics hardware. Furthermore, it is a widely used standard and available on multiple platforms like, Windows, Linux and Mac OS, see [32].

In contrast, DirectX which is often used for real time 3D graphics especially in game development is neither a standard nor does it support as many platforms.

Thus, *Vish* supports real time 3D graphics using OpenGL. A complete guide on programming OpenGL can be found in [51] and [50]. There are several possibilities how geometries can be created that have their own advantages and disadvantages. These are described in the following *section*.

Using OpenGL in a Rendering Module

OpenGL code is directly used in the draw function for rendering. The geometry of an object may be defined by geometric primitives using the `glBegin( GLenum mode )` and `glEnd()`. Hereby, the `GLenum` specifies the type of the primitive. Examples for possible types are points, lines, triangles, quads, quadstrips, etc. Necessary information for a primitive is provided inside the begin-end block using OpenGL calls for vertex positions, vertex colors, texture coordinates and similar. The OpenGL calls can be mixed with C++ code as well. *Listing 5.6* shows an example that creates a helix using a `GL_QUAD_STRIP`.

Listing 5.6: Example of creating 3D object geometry by using `glBegin( GLenum mode )` and `glEnd()`. The example is an excerpt of the tutorial example found in `/tutorial/opengl/Simple3DObject/Simple3DObject.cpp`

```
glBegin( GL_QUAD_STRIP );
    for (int i=0;i<N;i++)
    {
    double   phi0 = i*dphi,
             phi1 = (i+1/complexity)*dphi;

    point    A, B;
        A = r*cos(phi0), r*sin(phi0), phi0*zs;
        B = r*cos(phi0), r*sin(phi0), phi1*zs;

    bivector N;
        N = cos(phi0), sin(phi0), 0;

        embrace( Context, A );
        embrace( Context, B );

        glNormal( N );
```
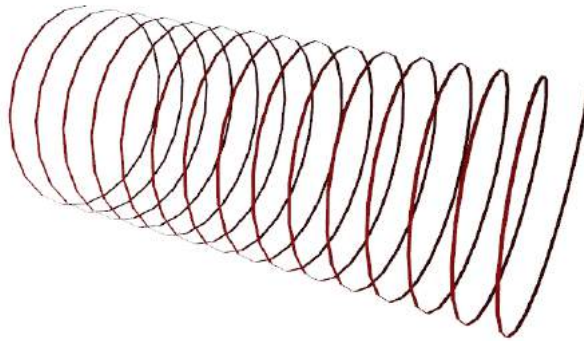
Figure 5.4: Helix rendered by the OpenGL code shown in *listing 5.6*.

```
18          glVertex ( A );
19          glVertex ( B );
20      }
21  glEnd ();
```

Here, a strip of quad primitives is created by stepping forward in the loop and adding two vertices with the same surface normal. The `glNormal` function encapsulates the OpenGL function `glNormal3f` for convenience and the `glVertex` accordingly. Note, the usage of the `VRenderModule` member function `embrace` to enlarge the objects bounding box on the fly, see *listing 5.6 line 14* and *line 15*.

This method can be very convenient for certain types of geometry but usually is very slow. While still using the same OpenGL calls this method can be accelerated drastically by compiling the calls into a so called display list. A pre compiled list of subsequent OpenGL commands will be executed much faster. To compile a display list in *Vish* use the macro `glCompile( )` which basically encapsulates the OpenGL command `glNewList()`, as shown in *listing 5.7*.

Listing 5.7: Compiling a display list in *Vish*

```
1  RefPtr<DisplayList> DL;
2      glCompile ( *DL )
3      {
4          glBegin ( /* ... */ );
5  /* ... */
6          glEnd ();
7      }
```

Of course, performance will only be gained when not recompiled with every call of the `VRenderObject`'s `code` render function. Thus, the display

list should be remembered and just be called and only be recompiled when geometry changes.

Vish provides a technique to cache a display list object using a so called *rendering cache, section 5.3. Listing 5.8* shows an example on how to use enable this caching functionality. The `[]` operators are used to create and the operators are used to access the *rendering cache.*

Listing 5.8: Code excerpt describing the caching of a display list in a *Vish intercube*. Taken from:
/tutorial/opengl/DisplayListObject/DisplayListObject.cpp

```
1  double   complexity = 0.5;
2     Complexity << Context >> complexity;
3
4  RefPtr<MyState>        state = myState(Context);
5  RefPtr<ValueSet>       RenderParameterSpace = new ValueSet();
6  RefPtr<DisplayList> DL;
7     try
8     {
9        DL = Context( *state )( this )( RenderParameterSpace );
10    }
11    catch (...)
12    {}
13
14    if ( DL && state->complexity == complexity )
15    {
16       if ( DL->call() )
17       return;
18    }
19
20    DL = Context[ *state ][ this ][ RenderParameterSpace ];
21    if ( !DL ) throw "No␣Display␣List!?";
22       state->complexity = complexity;
23
24    resetBBox( Context );
25    glCompile( *DL )
26    { /* ... */ }
```

If there is no valid display list available the access in *line 9* will fail, a new list object will be created in *line 20* and compiled using the `glCompile` in *line 25*.

If a valid display list was retrieved dependent on the state, the module and a rendering parameter space the custom parameter `complexity` is also checked for being up-to-date additionally in *line 14*. Finally, the display list is called and the geometry drawn in *line 16*.

Unfortunately, the above described techniques to create and draw OpenGL geometries will not be supported in OpenGL versions newer than

3.0. So, for implementing quick tests or some modules for prototyping this might still be an option but not for long term features in *Vish*.

Display lists are superseded by so called vertex buffer objects (*VBO*s). This technique is similar to enabling textures in OpenGL. Buffers or data arrays have to be created which are then bind. Data is filled into the bind buffers. A bind buffer can then is drawn by a single OpenGL command, such as `glDrawArrays` or `glDrawElements`. Of course, data buffers must satisfy certain layouts required by the drawing commands.

Vish provides interfaces for these OpenGL functions, as shown in *listing 5.9*. The example source code created *VBO*s for the same helix geometry as in *listing 5.6*.

Listing 5.9: Code excerpt describing the creation of vertex buffer objects. Taken from:
`/tutorial/opengl/VertexBufferObject/VertexBufferObject.cpp`

```cpp
RefPtr<VBO> myVBO;
RefPtr<VertexArray> Points  = new TypedVertexArray<point >();
RefPtr<NormalArray> Normals = new TypedNormalArray<bivector >()
    {
    std::vector<point>    PointData;
    std::vector<bivector> NormalData;

        PointData.resize( 2*N );
        NormalData.resize( 2*N );

        for ( int i = 0; i < N; i++ )
        {
/* ... compute points A, B and bivector N as above ... */

            PointData[ 2*i   ] = A;
            PointData[ 2*i+1] = B;
            NormalData[2*i   ] = N;
            NormalData[2*i+1] = N;
        }
        Points ->load( PointData );
        Normals->load( NormalData );
    }
    myVBO->append( Points );
    myVBO->append( Normals );

    myVBO->setRenderer( new DrawArrays(GL::QUAD_STRIP, 2*N) );
```

First, a *VBO* object has to be created. Then, data arrays are prepared, see *line 2* and *line 3*. Next, `std::vectors` are used to store the computed

geometry data in RAM[7]. When the data in RAM is complete it is loaded into the data arrays and, thus, loaded in the memory of the graphics hardware, *line 21* and *line 22*. The data arrays have to be appended to *Vish*'s VBO object and finally the renderer has to be configured with the correct information about the type and size of data, *line 26*.

Prepared *VBO* objects can be cached in the same way as a display list. *Listing 5.10* shows the details.

Listing 5.10: Code excerpt describing the caching of vertex buffer objects. Taken from:
`/tutorial/opengl/VertexBufferObject/VertexBufferObject.cpp`

```cpp
1  RefPtr<MyState> state = myState(Context);
2  RefPtr<ValueSet> RenderParamSpace = new ValueSet();
3  RefPtr<VBO> myVBO;
4      try
5      {
6          myVBO = Context(*state)(this)
7                         ( RenderParamSpace )( VERTEXBUFFER() );
8      }
9      catch (...){}
10     if (myVBO && !myVBO->empty() && state->complexity==complexity)
11     {
12         if (myVBO->call() )
13         return;
14     }
15
16     myVBO = Context[*state][this]
17                     [ RenderParamSpace ]( VERTEXBUFFER() );
18
19     myVBO->clear();
20
21     state->complexity = complexity;
22
23  /* ... fill VBO as described above ... */
24
25     myVBO->call()
```

The only difference is the additional test if the *VBO* is empty in *line 10*. Otherwise it is totally equivalent to *listing 5.8*.

Shaders can also be used for rendering. *Vish* provides a class called `Program` for that purpose. The fragment, geometry or vertex shader code is then stored in a `static const char` string.

Listing 5.11 shows how a shader program is created, *line 4*, how it is validated for correct syntax, *line 7*, and how it is activated for use, *line 12*.

---
[7]Random Access Memory

Unexpected exceptions are caught in *line 14*. `Uniform` variables of the shader code can be accessed using the `setUniFormValue*` member function of the `Program` class.

Listing 5.11: Code excerpt describing how to use `Program` with a OpenGL shader.

```
1  RefPtr<Program>&MyProgram = FR.MyProgram;
2      try
3      {
4          MyProgram = new Program( colormap_vertex_shader,
5                                   colormap_fragment_shader);
6
7          if (!MyProgram->isValid() )
8          {
9              puts("MyRenderModule:_invalid_program!");
10             return;
11         }
12         MyProgram->use();
13     }
14     catch(const Program::Error&E)
15     {
16         E.print("Compiled_program_not_usable");
17         assert(0);
18     }
19
20     MyProgram->setUniformValuei( "volumedata", 0);
21     MyProgram->setUniformValuei( "colormap", 1);
22
23     {
24     double val = 1.0;
25         Sharpness << Context >> val;
26         MyProgram->setUniformValuef("Sharpness", std::exp(5*val));
27     }
28     {
29     int N = 1;
30         NumberOfPeaks << Context >> N;
31         MyProgram->setUniformValuei( "NumberOfPeaks", N);
32     }
```

For more information of shader programming in general, see [50] and for more examples in *Vish* see the tutorial section of *Vish*.

5.2.4 Vish Scripts

A simple script language is used to define networks in *Vish*. At the current state the script languages enables to create modules, to set parameter values,

to connect parameters and to open `*.f5` data files. It is a keyword free language and supports *UTF8*.

A script file can be loaded by using the `File->Open` of the *Vish* GUI or by giving the script as the first parameter when starting *Vish* from the command shell, for example from the `/data` directory:

```
../bin/vish Analytic.vis
```

A new **module** can be **created** by using the name string of its `VCreator`. This string also is used to create the context menu for module creation in the GUI, *section 5.2*. A module found in `Utility/Point3D` can be created using the same string followed by a name for the instance the module. For example:

```
Utility/Point3D myPoint
```

Parameter values can be **set** by using the name of the module instance followed by the name of the parameter to be set separated by a dot. An equality followed by the value will set the parameter value. If the value is of wrong type or if the value is out of range of the `min` and `max` properties of the parameter the script line will be ignored. For example:

```
myPoint.x=1.5
```

Setting a parameter can be done within the context of a certain view port window. A parameter can have different values in different view ports. To specify the view port use `{}` after the parameter name. For example:

```
myPoint.x{Viewer2}=1.5
```

A **parameter** can be **connected** to the an other parameter. Most frequently it is connected to an output of another module. In that case it is sufficient to specify the name of the module instance and the appropriate output parameter is found automatically. The `=>` operator is used for parameter connection. A connection can also be done to an explicit parameter. For example:

```
vertices.grid=>geodesics
```

To **load** an **F5 file** via script the `@` symbol is used followed by the string of the directory and filename enclosed by `""`. For example:

```
@"../data/mydata.f5"
```

The `#` symbol is used for **comments**. All characters after will be ignored until the next newline.

Enclosing the name of a module instance by `<>` forces an **update** of the module:

```
<myPointViewer1>
```

Prefixing a ! in front of a name of a module instance will *touch* the module and flags it for a **required update**:

```
!myPoint
```

The GUI frame showing the module can by **minimized** hiding all parameter GUI controls. Then only the name of the module is shown in the network, see left most figure of *figure 5.3*:

```
Network.iconify(myPoint)
```

Listing 5.12 shows a typical *.vis script using most of the script features described above.

Listing 5.12: Typical vis script defining a *Vish* network. The example shows the content of **/fish/lakeview/cephalus/UnigridIsosurface.vis**

```
 1  CreateFiber/AnalyticScalarfield  AnalyticScalarField
 2  #AnalyticScalarField.spacetime=>MyGrid.spacetime
 3  #!AnalyticScalarField.a
 4  <AnalyticScalarField{Viewer1}>
 5
 6  Colormaps/Colorramp  FieldColors
 7  Display/OrthoSlice  MyFieldSlice
 8  MyFieldSlice.colormap=>FieldColors
 9
10  MyFieldSlice.scalarfield=>AnalyticScalarField
11  MyGridBox.spacetime=>AnalyticScalarField
12
13  Network.iconify(AnalyticScalarField)
14  Network.iconify(MyFieldSlice)
15  Network.iconify(MyGridBox)
16
17  Display/BoundingVolume  BVol
18  BVol.source=>MyFieldSlice
19
20  Compute/IsoSurface  IsoSurface
21  IsoSurface.scalarfield=>AnalyticScalarField
22  IsoSurface.isolevel=0.5
23  #<IsoSurface>
24
25  Display/SurfaceView  IsoView
26  IsoView.grid=>IsoSurface
27  #<IsoView{Viewer1}>
```

5.3 Caching

A good caching approach in the visualization network[8] is identified as key requirement for interactive exploration of huge datasets. Caching avoids unnecessary re-computations, speeds up computation and rendering, leading to interactive frame rates that would not be possible otherwise.

Vish allows caching on three levels. Firstly, caching in a **network module**. Secondly, data can be cached at the any **data sources**, especially on *Fiber Bundle* data sources and on data flow nodes itself. Finally, OpenGL *display lists* and *VBO*s can be cached directly on the **graphics hardware**, *section 5.2.3*. All these caching techniques are used during the implementation of the modules needed for visualizing geodesics.

Data can be cached or stored in the *network module* by using a `State` object, *section 5.2.1*. This procedure is suitable for small sets of data, like remembering a certain attribute value from a last call of `update`, to check if it had been changed or similar. Data storage is kept outside the module.

The description below is following the content of [12], also included in *appendix C*.

Data can be cached in **data sources**. The data source itself has to be a so called *InterCube* object, as described in [11]. A template class, the so called *OperatorCache* is used to cache some computational results associated to the *InterCube*. Any class can inherit the *InterCube* properties be being derived from the `MemCore` class `InterCube`. A data set like, for example, a vector of doubles can then be stored by initiating the template *OperatorCache*:

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

"Now given an InterCube object provided to a computational routine, one may retrieve an OperatorCache object that may be stored there. If not, we would need to create one anyway:" [12]

```
1  void VizNode::compute(InterCube &MyData)
2  {
3      OC_t*OC = OC_t::retrieve( MyData, this );
4      if( !OC ) OC = new OC_t();
5  }
```

Data is retrieved from the *InterCube* dependent on the data type and the visualization module. Data again is stored outside the local memory.

Needed re-computation, for example when certain parameters get changed, can be decided by utilizing the `unchanged()` member function of the *OperatorCache* template. The function takes a `ValueSet` that is tested for changes.

---

[8]or visualization cascade, in contrary to visualization pipeline

Listing 5.13: Code fragment showing how to use a `ValueSet` of `TypedSlots` with a `OperatorCache`.

```
RefPtr< ValueSet > Changeables = new ValueSet();

    *Changeables <<= IsoValue(Context);

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

After a new `ValueSet` is created in *line 1* the `TypedSlot<double>` `IsoValue` is added to the set using the `<<=` operator in *line 3*. The *OperatorCache* is then queried with the `ValueSet` wheter something has to be done or not.

Data can be cached in **Fiber Bundle data sources** inside a *Fiber **Bundle*** if it is stored as a *Fiber Grid* object, *chapter 4*. In that case the *Fiber Grid* is identified by a `double` time and a `string` name for identification in the *Fiber Bundle*. When additional values should influence the identification they just can be concatenated to the name string.

Listing 5.14: Example of caching a *Fiber Grid* in a *Fiber Bundle*. Based on the example described in [12].

```
Grid VizNode::compute(Bundle&B, double time,
                      string Gridname, string Fieldname,
                      double Isolevel)
{
string   IsosurfaceName = Gridname + Fieldname + Isolevel;

RefPtr<Grid> IsoSurface = B[ time ][ IsosurfaceName ];

    if (!IsoSurface)
    {
    RefPtr<Grid> DataVolume = B[ time ][ Gridname ];
        IsoSurface = Compute( DataVolume, Fieldname, Isolevel);

        B[ time ][ IsosurfaceName ] =  IsoSurface;
    }
    return IsoSurface;
}
```

Listing 5.14 shows a source code example where a *Fiber Grid* object is cached in the *Fiber Bundle*. The caching is dependent on the actual time, the *Fiber Grid* and the *Fiber Field* names of the underlying data source

and a parameter `IsoLevel`. The *Fiber Grid* object `IsoSurface` is computed and stored in the *Fiber Bundle, line 11* to *line 14*. This is an expensive operation since the underlying data sets may have to be loaded from disk and the computation of the `IsoSurface` has to be done. In a second call of the `VizNode::compute` function this operations are skipped because it can successfully retrieve the `IsoSurface` from the *Fiber Bundle* B in *line 7*.

5.4   Data Field Interpolation and Finding Local Coordinates

While the *Fiber Bundle* library provides a clear and unified structure to store all kinds of scientific data, working with the library itself still needs special some handling for different types, such as, checking for fragmented grids, checking for refinements levels and similar.

When developing visualization modules some of the required source code is needed frequently in a similar fashion. There is still room for implementing convenience functions and classes that ease accessing *Fiber Bundle* data structures.

In context of this work, for instance, the local index coordinates of a grid object by a given world coordinate point had to be computed. The local index coordinate is then used for data field interpolation. This is a common approach, see, for example, [18].

Figure 5.5 shows two 2 dimensional grids and a point described in local and in world coordinates. Computing local coordinates in an uniform regular grid is straightforward by doing a linear interpolation of the local coordinates of the cell that contains the point. But, when computing a local point in a curvilinear multi-block grid things become more difficult, especially when trying algorithms must be fast.

The modules that needed local index computation should work with all possible grid data structures. The calculation needs different handling dependent on the underlying data. Possible grids are uniform, multi-block, curvilinear or AMR grids.

I introduced a class called `LocalPointFinder`. It takes a *Fiber Slice*, a *Fiber Grid* and the *Fiber Field* holding the coordinate values as parameter and provides a `get` function that returns a local coordinate and the fragment ID of the fragment that contains the point in case of multi-block data by a given world coordinate value:

```
bool get( const point position, pair<point, string>&data );
```

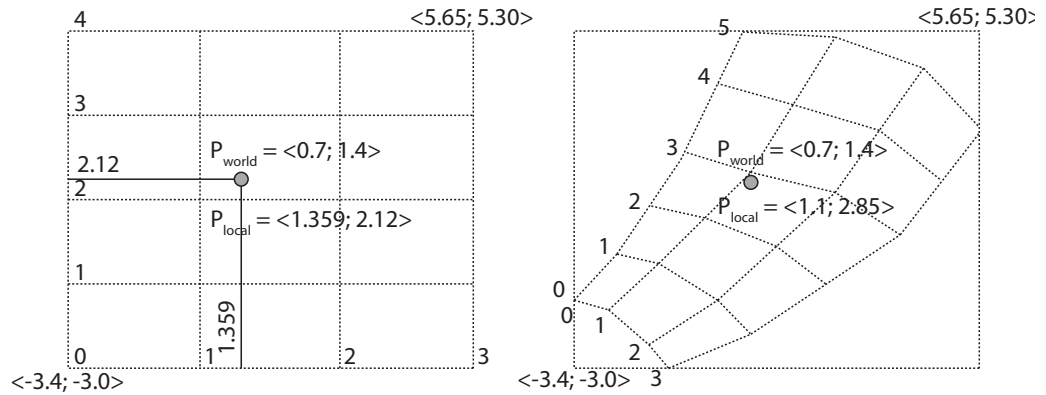Now, a developer of a module can conveniently get the local coordinates

Figure 5.5: A point is represented in world coordinates and in local index coordinates. The left side shows the simple case where local coordinates can be computed by linear interpolation. The right side shows local coordinates in an curvilinear grid. The surrounding rectangular illustrates the bounding box of the grid.

as shown in *listing 5.15* without caring about the underlying grid structure. After the creation of the `LocalPointFinder` class in *line 8* its `get` function is called, see *line 20*, where the local coordinates of the point `myPoint` are retrieved into the `pair<> localPoint`.

Listing 5.15: Getting local coordinates of a grid object. Here, a `FieldSelector` is used to extract all the grid information required for the local point search.

```
1  FieldSelector     FieldSelection;
2     MyField << Context >> FieldSelection;
3
4  /* ... check if data is available in the FieldSelection ... */
5
6  Info<Slice> currentSlice = FieldSelection.FieldSource;
7
8  RefPtr<LocalFromWorldPoint>
9  PointFinder = new LocalFromWorldPoint(
10                 *currentSlice.getSlice(),
11                 FieldSelection.getGrid(),
12                 FieldSelection.getGridname(),
13                 FieldSelection.getGrid()->getCartesianPositions()
14                 );
15
16 pair<Eagle::PhysicalSpace::point, std::string> localPoint;
```

```
17
18  Eagle::PhysicalSpace::point myPoint(0.2,  0.1,  4.4);
19
20  bool test = PointFinder.get( myPoint, localPoint );
```

The following sections describes more details of the `LocalFromWorldPoint` class in depth. The class itself was the result of several rewritings and optimizations and finally became a part of the core of the *Fiber Bundle* library, found in `/fish/fiber/baseop`.

5.4.1   UniGrid

The simplest case for computing a local coordinate applies to single-block uniform regular grids. Utilizing a fragment iterator as described in *chapter 4* will automatically capture support for single and multi-block grids. Inside the iterator it has to be checked if the vertex data is given procedurally.

To completely describe a uniform regular grid it is sufficient to know its world position bounds and the number of cells in each direction, *chapter 4*.

Getting this data is shown in *listing 5.16*. *Line 8* checks if a procedural `ProcArray_t` could be retrieved. If successful this describes a uniform regular grid.

Listing 5.16: Example of checking for a simple uniform field and doing a simple linear interpolation to compute local index coordinates of a given space point.

```
1  RefPtr<Field> myField
2  point position;
3
4  /* ... */
5
6     RefPtr<ProcArray_t> PCrd = coords->getData();
7
8     if( PCrds )
9     {
10    point DomainStart = PCrd->first();
11    point DomainEnd =   PCrd->last();
12    tvector DomainDiagonal = DomainEnd - DomainStart;
13
14    MultiIndex<3> FSize = PCrd->Size();
15    point max_float_index;
16       max_float_index = FSize[0]-1, FSize[1]-1, FSize[2]-1;
17
18    float_index = point( tvector( component_wise(component_wise(
19                 position - DomainStart, DomainDiagonal,
20                 Eagle::Operator<'/'>() ),
```

21 | $\qquad$ max_float_index , Eagle :: Operator<'*'>())));

The following code shows how to do the interpolation to compute local coordinates. The dimensions in all axis directions are stored in the `DomainDiagonal`. The maximal index length of the 3D data array is computed by $(sizeX-1, sizeY-1, sizeZ-1)$, see *line 18*. Now, the interpolation is done by computing $x_{local} = \frac{x_{pos}-x_{start}}{x_{interval}}(maxindex_x)$ in all dimensions simultaneously using the `component_wise` template. This allows to do mathematical operations component wise on a `FixedArray<>` and furthermore enables the use of *SIMD* instructions on the processor, if possible.

5.4.2 Multiblock

When dealing with a multi-block dataset at first the block that contains the world point has to be found. The fragments itself may be uniform or curvilinear.

A simple and quick to implement solution would be to do an iteration over all fragments and test if a fragment's bounding box contains the world point. However, when having thousands of block more sophisticated methods can speed up the search drastically.

In case of fragmented uniform data just one fragment would be found as a result of a testing bounding boxes. When dealing with curvilinear grids more than one fragment can be found that contains the world point, because boundinx boxes overlap, see *figure 5.7*.

I chose a binary tree search for finding candidate multi-block. A *KDTree* is utilizzed. Distances to the center of candidate fragments are computed and returned into a *STL* container.

I implemented an N dimensional *KDTree* based on John Tsiombikas C implementation (`http://code.google.com/p/kdtree/`). It was almost completely rewritten using C++ template programming.

The tree stores any data elements at a N dimensional position. When all data was inserted into the tree a range query can be done by specifying a N dimensional position and a distance. All data elements inside this distance are then returned.

Figure 5.6 shows the geometrical structure of a *KDTree*. Data positions are inserted into the tree. A new data position is stored left of right of the parent position. Thus, the inserted data position splits the space in two "child" regions. On each level of the tree nodes the direction of the splitting is switched. In $2D$ the space is split by alternating lines in x and y direction. In 3D space is split by alternating planes or bivectors with normals in x, y and z direction.
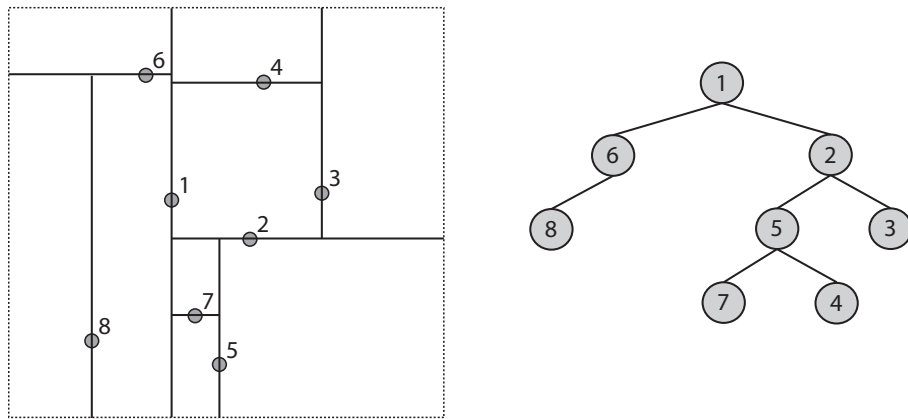
Figure 5.6: *Left:* Geometrical illustration of a KDTree. Points are added into the tree in order of their numeration. Each insertion splits the space in two regions: left and right or bottom and top. If a point is added it is inserted as left or right child node dependent on the position of the parent. On each tree level the direction of the splitting axis is alternated and the remaining subspace is split. *Right:* Tree structure of objects. Each node can have a left and a right child. The tree corresponds to the positions on the *left*. Bottom nodes are inserted left and top nodes right.

Listing 5.17 shows the structure of the `KDTree` class. The template parameters are `int N` for the dimension of the used space or the dimension of the locations where data is stored and `class T` for an arbitrary data type. It is derived from `MemCore::ReferenceBase` to enable reference counting for `KDTree` pointers.

The tree itself is build from tree nodes. Every node has a left and a right child and stores the data `T data` at the `N` dimensional position `pos`. Additionally the direction of the splitting has to be remembered for each node. The tree node is implemented as an inner `struct Node` of `KDTree`, see *line 4*.

When new data is inserted into the tree, only one coordinate of the *N*-dimensional position is checked whether it is located left or right of the current node. The current splitting coordinate is stored in the `dir` member of the node. If a leaf node is reached the position and its associated data is added as a left or right child.

The `Node`'s `insert` function is defined recursively, *line 22*. The `KDTree`'s `insert` function just handles the right start case and starting insertion at its `root Node`, *line 39*.

Listing 5.17: Class header of the KDTree class.

```
template<int N, class T>
class KDTree : public MemCore::ReferenceBase<KDTree<N,T> >
{
    struct Node
    {
    FixedArray<double, N> pos;
    int                     dir;
    T                       data;
    Node*           left;
    Node*           right;

    Node( const FixedArray<double, N>&posP,
          const int dirP, const T dataP)
    : pos(posP), dir(dirP), data(dataP), left(0), right(0) {}

    ~Node();

    template<class Functor>
    void callFromFar( const FixedArray<double, N>&queryPos,
                      Functor&func);

    void insert_rec( const FixedArray<double, N>&posP,
                     const T&data );

    template <class Container>
```

```
26      void find_nearest( const FixedArray<double,N>&posP,
27                          double range, Container&list );
28
29    };
30    Node* root;
31
32 public:
33    KDTree()
34    :MemCore::ReferenceBase<KDTree<N,T> >(this)
35    ,root(0) ,min_range(0.0) {}
36
37    ~KDTree();
38
39    void insert( const FixedArray<double, N>&pos,
40                 const T&data );
41
42    template<class Container>
43    bool nearest_range( const FixedArray<double, N>&pos,
44                        const double range,
45                        Container&res );
46
47    template<class Functor>
48    void callFromFar( const FixedArray<double, N>&queryPos,
49                      Functor&func );
```

The `nearest_range` function utilizes the split space information to speed up the recursive range check. Many unlikely candidates can be excluded from the range test. A *N*-dimension position, a range value and a container have to be specified as parameters.

The function uses a template call-back to fill an container defined as a type trait, *section 3.1*. Support for any container can be added by template specialization of `KDTreeResult` as shown in *listing 5.18*. Data insertion is encapsulated by the `insert` function of the type trait, see *line 13*.

The container itself is passed as a reference to the `nearest_range` function. This example shows how a *STL* multimap can now be used to return results of a tree query. The data elements of the tree and the squared distance are returned as a multimap. Data insertion automatically sorts the elements by its distances.

Any other data container can be utilized by adding the according trait. The `KDTree` is thus independent of the container type and highly reusable, as suggested in [31].

Listing 5.18: Adding a container type that can be filled with results from a `KDTreff` range query.

```
1 template<class Container>
2 struct KDTreeResult;
```

```
3
4  template<class T>
5  struct  KDTreeResult<std::multimap<double,T> >
6  {
7      std::multimap<double,T> & result;
8
9      KDTreeResult( std::multimap<double,T>&r )
10     :result(r)
11     {}
12
13     void insert( const double dist_sq , const T&data )
14     {
15         result.insert( std::pair<double,T>( dist_sq , data ) );
16     }
17 };
```

Listing 5.19 shows the usage of the KDtree. A four dimensional tree is created. Data is filled into the tree. Then a multi-map container is prepared in *line 12* and used for the range query at `querypos` with range 0.5 in *line 16*.

Listing 5.19: Using the `KDTree`.

```
1  RefPtr<KDTree<4, int> >tree = new KDTree<4, int >();
2
3  FixedArray<double, 4> querypos , pos1 , pos2 ;
4      pos1 = 0.0 ,   0.0 ,   0.0 ,   0.0;
5      pos2 = 0.5 ,   0.0 ,   0.0 ,   0.1;
6
7      tree ->insert(pos1 , 1);
8      tree ->insert(pos2 , 2);
9
10     /* ... insert much more int data ... */
11
12 stl::multimap<double, int>res_map;
13
14         querypos = 0.1 ,  0.1 ,  0.1 ,  0.1;
15
16     tree ->nearest_range( querypos , 0.5 , res_map )
```

Such a `KDTree` is used in the `LocalFromWorldPoint` class to chose candidate fragments. In case of a fragmented vertex field a `KDTree` is created in the constructor of `LocalFromWorldPoint` by iterating over all fragments, computing the center of each fragment and storing the fragment index at its center positions into a $3D$ `KDTree`.

Since the `KDTree` only has to be computed once for a fragmented grid object it is cached using an *InterCube* object to the coordinate field. To enable this feature an additional wrapper template class has to be provided

that extends the `KDTree` by an additional interface, see *listing 5.20*.

Listing 5.20: Allow a `KDTree` to be stored in an *Vish InterCube* by deriving from `Memcore::Interface<>`

```
template<int N, class T>
struct KDInterface : public MemCore::Interface <KDTree<N, T> >
{
    MemCore::RefPtr<KDTree<N, T> > Tree;

    KDInterface(const MemCore::RefPtr<KDTree<N, T> > TreeP)
    : Tree(TreeP)
    {}
};
```

In the constructor of `LocalFromWorldPoint` the KDTree is cached to the vertex field `RefPtr<Field> coords`:

```
RefPtr<KDTree<3, int> > MyTree;

RefPtr<KDInterface <3, int> >
TreeInt = coords->findInterface( typeid(KDTree<3, int>) );

    if (!TreeInt)
    {
        MyTree = new KDTree<3, int >();
        TreeInt = new KDInterface<3, int >( MyTree );
        coords->addInterface( TreeInt );
    }
    else
    {
        MyTree = TreeInt->Tree;
    }
```

Besides storing the centers and the fragment numbers, also the size or bounding radius of the largest fragment is stored. This maximal bounding radius is used for the tree range queries. This ensures that no fragment candidates are missed. *Figure 5.7* illustrates an example in 2D showing fragments, midpoints, the query point and the query range.

Having found these candidates a more accurate test has to be done to find the correct fragment and fragment cell, that contains the world point.

5.4.3   Curvilinear

Finding a the containing cell in a arbitrary curvilinear hexahedral cell in a curvilinear grid turned out to be a non trivial problem.
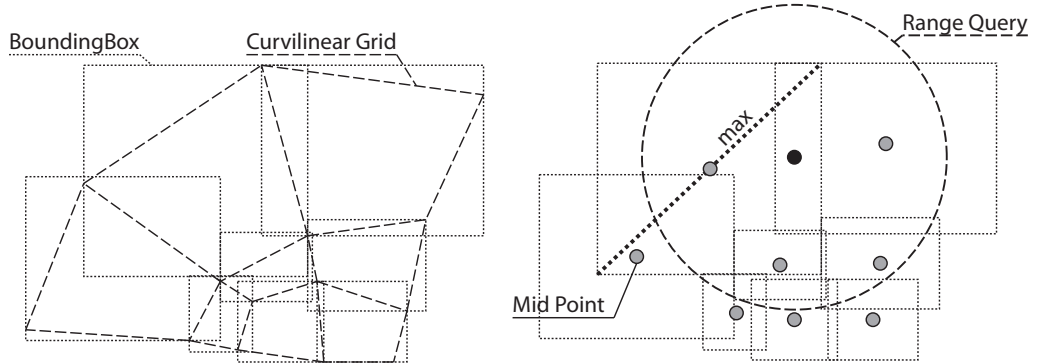
Figure 5.7: KDtree used for a range query on mid points of bounding boxes of curvilinear multi-blocks. Left: Bounding boxes of curvilinear blocks. Right: Mid points of bounding boxes. The maximum diagonal of all bounding boxes is used as the diameter of the range check at an arbitrary position (black dot).

Local Coordinates in one Hexahedral Cell

First I tried to find a map to transform a point from world coordinates to local coordinates when it is already known that the point is located inside a cell. I reduced to a two dimensional case for investigation. In that case a general four sided polygon have to be mapped to a normalized square of size 1.0, see *figure 5.9*.

Though, on the first glance, it looked similar to barycentric coordinates in a triangle, the additional geometry constrains turned out to lead to a non conform mapping, in contrast to the linear mapping in case of a triangle. Besides just mapping the corner points also all four boundary lines must be continuously constrained to the borders.

I finally found a suitable transformation method applicable to general $3D$ hexahedral cells in [54]:

A world point $\mathbf{p}(u, v, w)$, with u, v and w being local coordinates in the curvilinear cell can be computed using the following trilinear interpolation of the world coordinates of the eight corner points $\mathbf{p_0}$, $\mathbf{p_1}$ ... $\mathbf{p_7}$:

$$
\begin{aligned}
\mathbf{p}(u, v, w) = \ & (1-u)(1-v)(1-w) \cdot \mathbf{p_0} + u(1-v)(1-w) \cdot \mathbf{p_1} + \\
& (1-u)v(1-w) \cdot \mathbf{p_2} + (1-u)(1-v)w \cdot \mathbf{p_3} + \\
& uv(1-w) \cdot \mathbf{p_4} + u(1-v)w \cdot \mathbf{p_5} + \\
& (1-u)vw \cdot \mathbf{p_6} + uvw \cdot \mathbf{p_7}
\end{aligned}
\tag{5.1}
$$

Given a point $\mathbf{p}^*(x, y, z)$ in world coordinates the local coordinates u, v, w are found by using an iteration method. First, a point $\mathbf{p}(u, v, w)$ is guessed
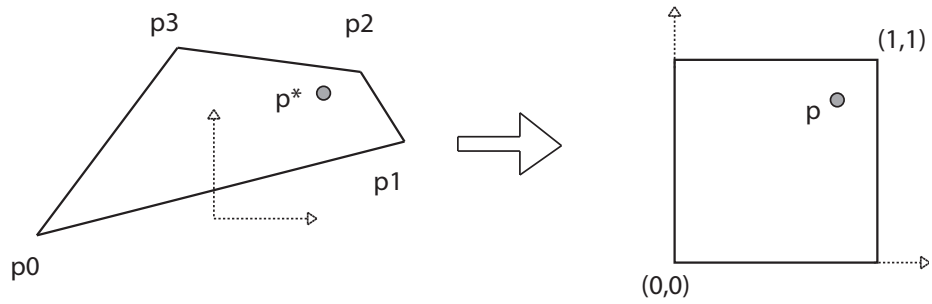
Figure 5.8: Mapping a general four sided polygon to a normalized quadratic turned out to be a non conform mapping.
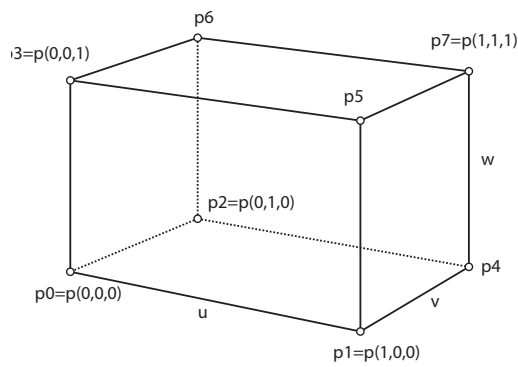

Figure 5.9: Three dimensional hexahedral cell.

inside the cell assuming certain local coordinates. I choose $u = v = w = 0.5$ the point in the center of the cell. The Taylor series expansion of *equation (5.1)* yields:

$$\mathbf{p}(u + \delta u, v + \delta v, w + \delta w) = \tag{5.2}$$

$$= \mathbf{p}(u, v, w) + \frac{\partial \mathbf{p}}{\partial u}\delta u + \frac{\partial \mathbf{p}}{\partial v}\delta v + \frac{\partial \mathbf{p}}{\partial w}\delta w +$$

$$+ \; O(\delta u^2) + O(\delta v^2) + O(\delta w^2)$$

with:

$$\frac{\partial \mathbf{p}}{\partial u} = (1 - v)(1 - w)(\mathbf{p_1} - \mathbf{p_0}) + v(1 - w)(\mathbf{p_4} - \mathbf{p_2}) + \tag{5.3}$$

$$+ (1 - v)w(\mathbf{p_5} - \mathbf{p_3}) + vw(\mathbf{p_7} - \mathbf{p_6})$$

$$\frac{\partial \mathbf{p}}{\partial v} = (1 - u)(1 - w)(\mathbf{p_2} - \mathbf{p_0}) + u(1 - w)(\mathbf{p_4} - \mathbf{p_1}) + \tag{5.4}$$

$$+ (1 - u)w(\mathbf{p_6} - \mathbf{p_3}) + uw(\mathbf{p_7} - \mathbf{p_5})$$

$$\frac{\partial \mathbf{p}}{\partial w} = (1 - u)(1 - v)(\mathbf{p_3} - \mathbf{p_0}) + u(1 - v)(\mathbf{p_5} - \mathbf{p_1}) + \tag{5.5}$$

$$+ (1 - u)v(\mathbf{p_6} - \mathbf{p_2}) + uv(\mathbf{p_7} - \mathbf{p_4})$$

$$\tag{5.6}$$

The desired increments δu, δv and δw can be computed by solving the following linear 3×3 equation system:

$$\frac{\partial \mathbf{p}}{\partial u}\delta u + \frac{\partial \mathbf{p}}{\partial v}\delta v + \frac{\partial \mathbf{p}}{\partial w}\delta w = \mathbf{p}^* - \mathbf{p} \tag{5.7}$$

Using the computed increments of the equation system a new point $\mathbf{p}$ is computed using *equation (5.1)*. Using the new point new derivatives are computed and the linear system is solved, again. This is repeated until $||\mathbf{p}^* - \mathbf{p}||$ falls under a certain threshold.

In the implemented algorithm a threshold of approximately $1/100^{th}$ of the cells size is used which is computed once in the constructor of `UniGridMapper`, see below.

Typically finding the local point requires one to two iteration steps to drop under the threshold. If the cell is distorted heavily the number of iterations increases.

This algorithm is also used to test if a point is contained in a certain cell. In that case the result must be inside $(0, 0, 0)$ and $(1, 1, 1)$.

The algorithm is used for two purposes. Firstly, to check if a point is located inside a cell, and secondly for the computation of the local hexahedral grid coordinates, which is already computed during the first test.
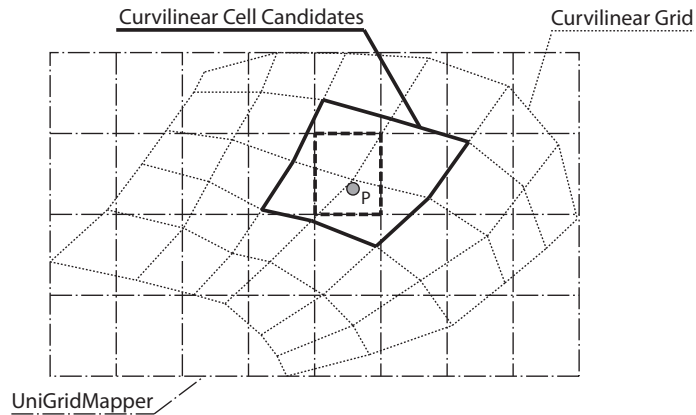
Figure 5.10: A curvilinear grid assigned to an uniform grid. The Uni-GridMapper provides a mapping from one uniform grid cell to curvilinear cells that touch it.

Finding Candidates in the Grid

When applying the test to a grid of curvilinear cells this would have to be done for all cells until the containing cells is found.

To speed up the search within a curvilinear grid I introduced a new data structure that reduces the number of cells to a few candidate cells for the testing, analogously to the KDTree for multi-blocks.

The idea was to create a uniform grid that matches the size of the bounding box of the curvilinear grid and then provide a map from uniform cells to curvilinear cells. Curvilinear cells touching a uniform grid cell are assigned to the uniform cell. *Figure 5.10* illustrates the method. The example illustrates the mapping of one uniform grid cell (fat dashed line) to four curvilinear cells (fat line). All other uniform grid cells provide a similar mapping.

I called the introduced class UniGridMapper. It provides the mappings for one curvilinear block. The UniGridMapper stores a list of indices of curvilinear cells that is accessed by the index of the uniform grid cell.

These lists are generated in the constructor of the class, providing the full mapping afterwards. While doing the iteration over all curvilinear cells also the bounding box of the UniGridMapper is computed and the average edge length of all curvilinear cells. The average edge length, or better, its $1/100^{th}$, is used for stopping the newton iteration as described earlier. The signature of the constructor is showed in *listing 5.21*.

Listing 5.21: Some interface details of class UniGridMapper

```
1  typedef MemArray<3, point> CoordsArray_t;
2
3  UniGridMapper( const RefPtr<CoordsArray_t>&_CurviCoords,
4                 const double res_scale = 1.0,
5                 const double prec_scale = 1.0 );
6
7  /* ... */
8
9  unsigned
10 localCellCoordinatesFromCurviGrid( const point & p, point & uvw,
11                                    double grid_epsilon = 0.0);
12 /* ... */
```

A `MemArray` holding the coordinates of the curvilinear block must be provided. Optional parameters control the resolution of the uniform grid, *line 4*, and the precision of the reached newton iteration, *line 5*. *Figure 5.11* shows different resolutions of the uniform grid.

The `UnigridMapper` class also implements the iteration method for the local coordinate computation, by providing the function `localCellCoordinatesFromCurviGrid`, see *line 10*.

When calling the function the list of curvilinear cells mapped to the according uniform grid cell are tested with the iteration method. Here, a problem arose at the borders of the curvilinear cells. Due to numerical inaccuracies it could happen, that a point was located slightly outside of, for example, four neighbored curvilinear cells.

To avoid to introduce epsilon tests I introduced a distance measure from the computed point to the closest borders of the cell. If none of the candidate cell contains a local point directly the "best" point, closest to a cell boundary is returned.

Listing 5.22 shows the neighbor test in *line 17*. The `DistPoint_t` **struct** is used to store possible points with distance and curvilinear cell index in a container, *line 11* and *line 32*. The city block (Manhattan) distance is computed in the **for** loop in *line 22*.

Listing 5.22: Handling local coordinates that are not found correctly at the borders of curvilinear cells

```
1  struct DistPoint_t
2  {
3      double dist;
4      point uvw;
5      MultiIndex<3> self;
6  };
7
8  /* ... */
9
```

```
10  MultiIndex<3> targetCell;
11  std::vector< DistPoint_t > EpsilonPreventer;
12
13  /* ... compute local coordinates into point uvw ...*/
14
15     localPointToIndex(uvw, targetCell);
16
17     if( fabs(targetCell[0] - curviCell[0]) <= 1.0 &&
18         fabs(targetCell[1] - curviCell[1]) <= 1.0 &&
19         fabs(targetCell[2] - curviCell[2]) <= 1.0 )
20     {
21         distPoint.dist = 0.0;
22
23         for(unsigned i = 0; i < 3; i++ )
24         {
25             diffPoint[i]     = uvw[i] - targetCell[i];
26             distPoint.dist  += ( diffPoint[i] < 1-diffPoint[i] ) ?
27                                 diffPoint[i] : 1-diffPoint[i] ;
28         }
29         distPoint.uvw = uvw;
30         distPoint.self = curviCell;
31
32         EpsilonPreventer.push_back( distPoint   );
33     }
```

In case all tests of the candidate cells fail, the local coordinate of a neighbor with the smallest city block distance is returned by `localCellCoordinatesFromCurviGrid` in `point&uvw`. In such a case the returned local point was chosen from the `EspilonPreventer`.

Since the initialization of the `UniGridMapper` is an expensive operation the `UniGridMappers` are cached into the *Fiber Grid* object, where they are stored as a fragmented field according to the fragmentation of the grid.

They are stored in a skeleton with dimensionality three and index depth two as a relative representation into the *Fiber Grid* of the vector field, see *chapter 4*.

So, when one block is queried for a local coordinate again the `Unigridmapper` is extracted from the *Fiber Bundle*.

Summarizing the Steps

Following pseudo code summarized the process when a local coordinate from a world point using the `LocalFromWorldPoint`'s `get` function is requested on a curvilinear multi-block grid:

```
if(no KDTree is found for the coordinate positions field)
{
```
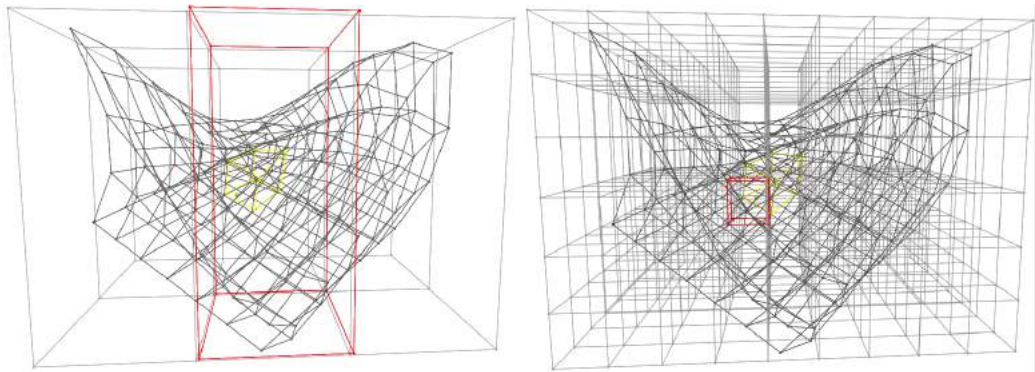
Figure 5.11: Graphical output of a rendering module for visually debugging the method used to find a local coordinate. A curvilinear grid (dark grey) is surrounded by a `UniGridMapper` (light grey). The world point used for the search is shown colored red. If a local point is found, within the threshold, it is recomputed to world coordinates by linear interpolation and drawn in green. The containing uniform cell is marked in red and the curvilinear cell is marked in yellow. The pictures show different resolutions chosen in the uniform grid: very low resolution on the left and higher resolution on the right. The module can be found in `/fish/lakeview/eye/retina/FindLocalCoordinates.cpp` and the illustrated scene can be started with the according `.vis` script located in the same directory

```
     *Compute KDTree by inserting fragment IDs at center positions
       of multi−blocks.
     *cache KDTree as Intercube to the coordinate positions.
}
else
    *Get KDTree from cache.

*Do a range tree query at the world position and retrieve
 candidate fragments.

for( each fragment candidate )
{
    if(no UniGrimapper is found in the bundle for the fragment)
       *Create UniGridmapper and store it in the bundle.
    else
       *Get Unigridmapper from bundle.

    *Get candidate curvilinear cells from the UniGridmapper.

    for( each candidate cell )
    {
       *Use iteration to find local coordinate.
       *Return a valid local coordinate.
    }
}
```

5.5   Basic Visualization Modules

During the development of the visualization modules and infrastructure needed for geodesics I implemented some useful basic features along the way. Here, five tools are presented that enhance any data visualization in *Vish*. Two tools display coordinates, one shows the resolution of a uniform grid, one labels colors of a color map with data values and one draws outlines of a multi-block grid.

5.5.1   Coordinate Grid

Displaying a coordinate plane is a very basic and helpful tool when analyzing 3D data. The implemented module is a simple rendering module with no required input connections. Coordinates are displayed as lines on a plane. A line is drawn at every multiple of 10 and every 10th line is labeled by its coordinate value. The main parameters of the module control the alignment, the size, the refinement, the offset, the font size and font position.
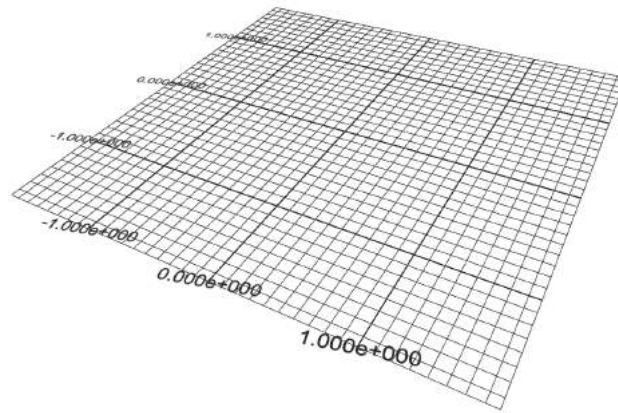
Figure 5.12: Coordinate Grid. A simple rendering module displaying basis coordinate planes and coordinate values.

The coordinate plane can be aligned in the xy, xz, and yz plane. The size can be fixed or be adjusted dynamically dependent on the camera distance. If the size is controlled dynamically the size on screen will be constant when zooming the camera. The refinement of the coordinate plane is always adjusted dynamically. But, one can choose between three levels of refinement: normal, coarse and fine (`finercoarser`).

The coordinate plane can be shifted in direction of its normal by a the `offset` parameter. Finally, the font size and the position can be controlled. The distance between the border of the grid object and the font can be set and a roll parameter defines the rotation around the coordinate plane normal. The coordinate values are displayed in engineering convention using the exponential notation.

5.5.2   Coordinate Grid Box

Another tool for analyzing 3D data and viewing its dimensions is the coordinate grid box. It is a rendering module that visualizes a connected bounding box. It also draws "rulers" on three main axis at multiples of 10 of the coordinates and labels the rulers with coordinate values.

The important attributes are used to control the ruler refinement, ruler size and the size, position and rotation of the coordinate labels.
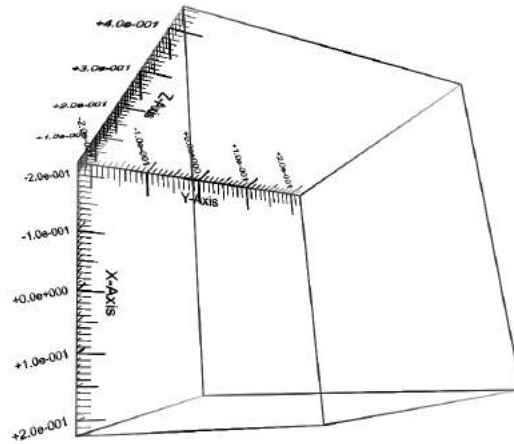
Figure 5.13: Coordinate Grid Box. A rendering module that displays the bounds of a bounding box and labels coordinate axes with numbers.

5.5.3   Uniform Grid Lines

The `GridLines` visualizes the grid resolution of a $3D$ uniform grid. Either the xy, xz or yz plane is shown. Therefore, a *Fiber Grid* must be connected to the module. Some basic parameters such as color, colorscale and an offset value are provided. The offset changes the position of the coordinate plane along its normal direction.

Figure 5.14 illustrates the resolution of a uniform grid by showing a bounding box and three coordinate planes drawn by three `GridLine` rendering modules. Here, the xy plane has no offset, whereas the two other planes have an offset of 0.5 moving the planes to the center of the grid.

5.5.4   Color Map Legend

When color maps are used to display some scalar fields then it is important to know and see the mapping between colors and scalar values. *Vish* had no such tool. It is common practice to show a color map using a colored bar and labeling numbers in the screen space[9], for example at the lower right corner of the $3D$ view.

First, I planned to create one single module to display color bar and number values. But then I recognized that it would be much more flexible when separating into two distinct modules, one for the bar and one for the labels. The labeling module would then be 'docked' to the color bar module.
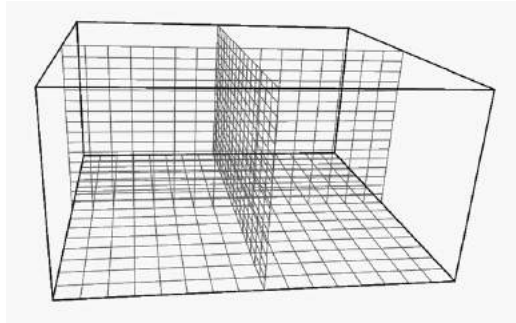
---

[9]camera image plane

Figure 5.14: Uniform Grid Lines. A rendering module that displays the grid resolution of a uniform grid. Here, three modules are used for drawing three axis oriented planes illustrating the resolution.
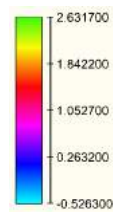

Figure 5.15: Color Legend rendered by two distinct modules. One for the color bar and one for the labels.

That way the labeling module would be easily replaceable be other modules, like a module that could display a histogram.

The basic idea was to build atomic components for HUD[10] nodes that can be connected and combined in different ways.

To specify a position in screen space a new attribute type was introduced: `Point2D`. Coordinates in screen space are normalized and run from 0.0 to 1.0 in x and y direction. A HUD component takes a 2D position as input and outputs a 2D position as output, that another docking components then can take as an input.

The attributes to control the color bar module `ColorLegend` are the position, the height, the width and some attributes that control its appearance. The most important input attribute is the `TypedSlot<VColormap>` where the color map to be visualized is connected.

The module for number labeling is called `RangeHUD`. It has an input attribute for position and some attributes for configuration, like subdivision, which controls the number of displayed values and an attribute for font size. The main input attribute is a `TypedSlot<Range>` that defines the number interval. To synchronize the height of the labeling ruler with the height of the `ColorLegend` an additional input slot `height` is connected.

Figure 5.15 shows the graphical result of a `ColorLegend` connected by position and height to a `RangeHUD`. *Figure 5.16* shows the necessary *Vish* network. Here, the connections described above are illustrated. A color map module is connected to the `ColorLegend` and a range module is connected to the `RangeHUD`. A `Point2D` module is connected to the `ColorLegend` module controling the overall position of the two HUD modules. `RangeHUD` is connected to `ColorLegend` by position and height, as stated above.

5.5.5   Multiblock Outlines

During development of algorithms dealing with curvilinear multi-block datasets it became necessary, especially for visual debugging, to display the boundaries of all or some selected multi-blocks.

Thus, I added a rendering module that iterates over the *Fiber Grid* fragments of a dataset and extracts and renders the outlines of a multi-block dataset. Attributes are provided to control the transparency and color of the outlines. Later, a filter was added, such that only those multi-blocks are drawn that had a `UniGridMapper` object associated, *section 5.4. Figure 5.17* shows an example of 2088 curvilinear multi-blocks.
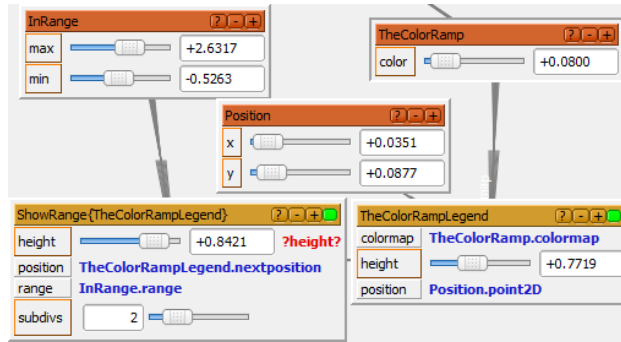
---

[10]head up display

Figure 5.16: Example network showing connections between modules needed to display a color legend.
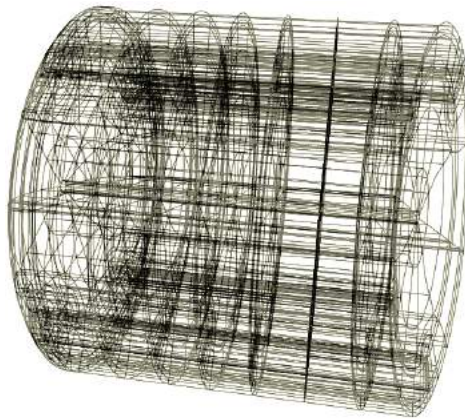

Figure 5.17: Example of displaying 2088 multi-blocks by using the `Display/Blocks` module. The rendering module extracts the outline for each multi block and renders all outlines using illuminates lines.

Chapter 6

Computation and Visualization

The first approach for computing and visualizing integral lines that was implemented was a short all-in-one *Vish* module. It received a procedural vector field as input. Streamlines were computed using a simple Euler integration, as described in *section 6.2*. Initial seed points were located on a fax in the xy-plane and could be moved around. Also the line rendering was done in the same module.

It took me just four days to implement this 400 lines of code *Vish* module and I learned a lot about the software environments and the requirements for integral line visualization. Now it can be found in the tutorial section of *Vish*: `SimpleStreamline.cpp`.

I recognized that the flexibility of the basic approach can be increased by separating the whole process into small tasks. The visualization process should be split into three basic modules:

- Define seeding geometry, or initial conditions.

- Do the integration.

- Render integration lines.

Thus, for all of these module types different modules could be implemented and assembled like building blocks according to the requirements of the visualization.

This chapter first describes modules that were developed for defining the seeding geometry, then introduces the implemented computation modules and finally presents a rendering module for integration lines.

6.1 Defining Initial Conditions for Integral Lines

6.1.1 Initial Positions, Seed Points

The first point distribution module I implemented created some simple geometric point distributions like points on a line or points on a circle and it used pure *Vish* attribute connections to transport a `std::vector<point>` to the now separated computation module.

For data transport, a new attribute type was introduced to be used in the network by providing `VValueTraits` as described in *section 5.2* accordingly. *Listing 6.1* shows the source code for the attribute introduction.

Listing 6.1: Emitter Points Attribute

```
namespace IntegralLines
{
struct EmitterPoints
{
        std::vector<point> Vertices;
};
} //namespace IntegralLines

namespace Wizt
{
template <>
class VValueTrait< ::IntegralLines::EmitterPoints >
{
public:

    static bool setValueFromText(::IntegralLines::EmitterPoints&i,
                                 const string & s)
    {
        return false;
    }

    static string Text(const ::IntegralLines::EmitterPoints&e)
    {
    std::string s("");
    std::stringstream tmp;

        if( e.Vertices.size() > 0 )
        {
            tmp << e.Vertices.size() << e.Vertices[0]
                << e.Vertices[1]      << e.Vertices[2];
            s = tmp.str();
        }
```

```
33            return s;
34       }
35  };
36  }//namespace Wizt
```

First, the new type used for an attribute connection is encapsulated in a new class: `struct EmitterPoints`. It contains a standard `std::vector` of 3D space points. Now, a new specialization of the `VValueTraits` for the new type is required. It contains two important functions to convert the data into text and vice versa. The `setValueFromText` function of the `VValueTrait< ::IntegralLines::EmitterPoints >` class is not needed here.

What is important is the `Text` function. Here, data is converted to text. The text string is used in the *Vish* network to trigger updates from connected modules. The created string of `EmitterPoints` contains the number of points and just the coordinates of the first point. Whenever the number of the points or the coordinates of the first point change, the module connected to an output of type `EmitterPoints` will execute its `update` function.

The next step was to use the *Fiber* data model to store the created points since they have all characteristics of a *Fiber Grid* object. So, such an object was stored into a *Fiber Bundle*. For data access to further modules a handle to the *Fiber Grid* was provided.

Later a second module was added that creates randomly distributed 3D space points inside a defined 3D volume.

Geometric Point Distributions

The geometric point distribution module creates a *Fiber Grid* of simple geometric shape. Namely: point, line, rectangle, circle, ellipsoid, uniform rectangular grid, circular grid, sphere and cube.

The shape can by controlled by two scaling parameters that stretch the shape in one of two axes and by two parameters controlling the subdivision in each axis direction. *Figure 6.1* shows all possible shape types with a certain set of parameters.

The position and the rotation of the shapes are controlled by input parameters into the module and are connected to a *Vish Point3D* and *Vish Rotor* module.

An example network of the *Vish* network that just draws some points from *figure 6.1* is shown in *figure 6.2*.

To transform the created points two handles were added. One for translation, a 3D space point defining the origin of the shape and one for rotation, a rotor.
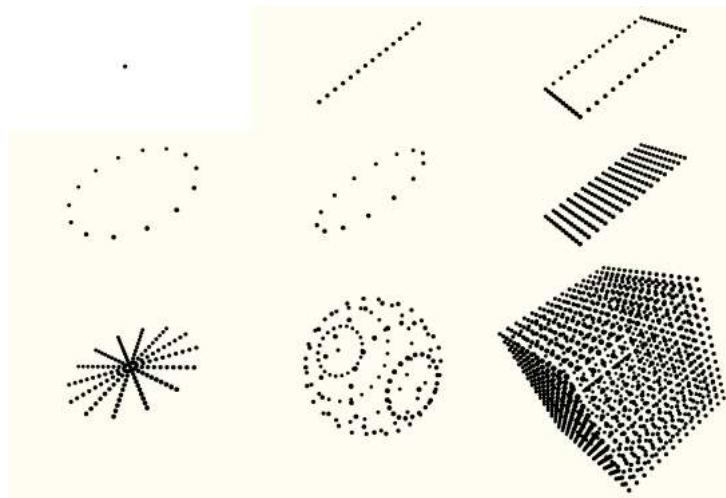
Figure 6.1: Simple geometric shapes created with the point distribution module. Paramters: *height* = 0.5, *length* = 1.0, *height_subdivs* = 5, *length_subdivs* = 10. Types from left to right and top to bottom: point, line, rectangle, circle, ellipsis, rectangular grid, circular grid, sphere and cube.

Listing 6.2 shows some simplified source code of the module. The `class pointsShapes` is derived from `VObject` making it a *Vish* module. It gets the input control parameters via `TypedSlots` and provides a `GridSelector` of a *Fiber Grid* as output via `VOutput`.

Listing 6.2: Geometric Point Distribution

```
class      PointsShapes : public virtual VObject
{
public:
    TypedSlot<double> Length,
                      Height;
    TypedSlot<int>    LengthSubDivs,
                      HeightSubDivs;
    TypedSlot<point>  Pos;
    TypedSlot<rotor>  Rotation;
    TypedSlot<Enum>   EnumType;

    VOutput<Grid>        myGrid;

    PointsShapes( const string & name, int p,
                  const RefPtr< VCreationPreferences > & vp )
    : VObject( name, p, vp )
    , Length( this, "length", 1.0 )
/* ... some more TypedSlot initialisations ... */
```

```
19
20      , EnumType( this , "type", Enum("Point", "Line", "Rectangle",
21                             "Circle", "Ellispoid", "RegularGrid",
22                             "CircularGrid", "Sphere", "CubeGrid"))
23      , myGrid( self (), "emitterpoints", GridSelector() )
24      {
25          LengthSubDivs.setProperty( "max", 100 );
26  /* ... set some more slot properties ... */
27
28          if( attachUniqueObject( Pos ) < 0 )
29          {
30              AttachErrorCode AC = attachNewObject(Pos,
31                                                  name + "position");
32              if ( AC<0 )
33                  printf("position: %s\n", AttachErrorCodeMessage(AC));
34          }
35          Pos->AllowAutoConnection = true;
36  /* ... similar code for automatic rotor module creation ... */
37      }
38
39      ~PointsShapes() {}
40
41      override bool update( VRequest&R, double precision ) { ... }
42  };
```

The constructor of `PointsShapes` is used to a constrain data ranges of several input parameters, see *line 25*. Also, standard modules that should be connected to input slots can be created automatically. For example, the standard 3D space point creation module can be automatically connected to the `TypedSlot<point>` Pos, see *line 28* to *line 35*.

If the user in *Vish* creates a new geometric point distribution module the modules for controlling the origin and rotation are created and connected automatically to the point distribution module.

All computation and creation is done inside the `update` function. Here, all slot parameters are read first and dependent on them, a `vector` or points is computed. Finally, these points are stored in a *Fiber Grid*. *Listing 6.3* shows how this is done.

Listing 6.3: Geometric Point Distribution

```
1  bool PointsShapes :: update(VRequest&R, double precision)
2  {
3  /* ... input read ... */
4  /* ... points computed into vector<point> StartPoints ... */
5
6      BundlePtr BP = new Bundle();
7      Slice & S    = BP[ 0.0 ];
8
```

```
 9    string grid_name = Name() + "_PointDistrib";
10    Grid & PointGrid = S.newGrid( grid_name );
11
12    Skeleton & PointVertices  = PointGrid.makeVertices(3);
13    RefPtr<Chart> myCartesian = PointGrid.makeChart(
14                              typeid( CartesianChart3D ) );
15
16    Representation&CartesianVertices = PointVertices[myCartesian];
17    RefPtr<Field> Coords       = CartesianVertices[FIBER_POSITIONS];
18
19    typedef MemArray<1, point> CrdsA_t;
20    RefPtr<CrdsA_t> CoordsData = new CoordsArray_t(
21                              StartPoints.size() );
22
23    MultiArray<1,point > & Crds1 = *CoordsData;
24    {
25        for( index_t i=0; i < StartPoints.size(); i++ )
26        {
27            MultiIndex<1> n( i );
28            Crds1[ n ] =  StartPoints[i];
29        }
30    }
31    Coords->setPersistentData( CoordsData );
32
33    GridSelector GS( grid_name, BP );
34    myGrid << R << GS;
35 }
```

During the development the idea arose to use a *Fiber Grid* object handle instead of the *Point3D* attribute for defining the translation. A minor change in the point distribution module made it possible to create the shapes at all the 3D points of the input grid. So every point of the *Fiber Grid* was an origin to the created shape. Thus, complicated point distributions could be created in a recursive fashion by combining more point distribution modules one after the other.

For example, the first point distribution module would create a line of points and output the *Fiber Grid*. A second point distribution module would take this as an input grid and around each point of the line would create a small circle of points.

Later this recursive creation functionality was identified to be a **pure Fiber Grid operation** and was thus extracted into an own module only operating on *Fiber Grid* objects. The operation was called **Grid Convolution** and is described in detail in *section 6.1.1*.
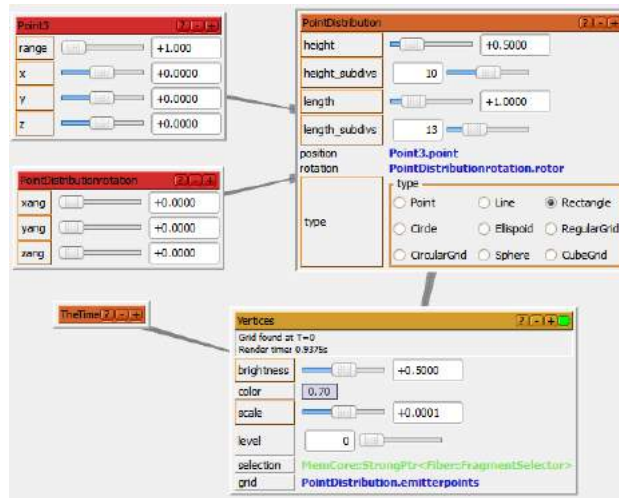
Figure 6.2: Vish network for drawing points of a geometric point distribution. The modules *PointDistributionrotation* and *Point3* create a point and a rotor which are input into the distribution module *PointDistribution*. The distribution module creates a *Fiber Grid* which is output into the module for drawing the vertices.

Random Point Distribution

The random point distribution creates seed points in a certain volume which are randomly placed in space. One parameter controls the number of points generated and one a random seed. There are two ways how the dimensions of the volume can be controlled.

Firstly, a radius parameter can be used that specifies the length of a cubic volume, see *figure 6.3*.

Secondly, a *Fiber Grid* object can be connected instead. The bounding box of this objects then defines the size of the volume. If the *Fiber Grid* nests a *Fiber Scalar Field* it is possible to filter the created random points by a certain scalar value range. In that case, the scalar value is evaluated at each random point and is checked against the given range. If the value is outside the range, the point is clipped.

Figure 6.4 shows some clipped random points by clipping against the pressure value, revealing some structure of a mixing ventilator in the center of the data set.

During the use of the clipping functionality it was recognized that this feature could be taken further and used in other aspects as well. Making it possible to sample any given *Fiber Field* on an arbitrary other *Fiber Grid*
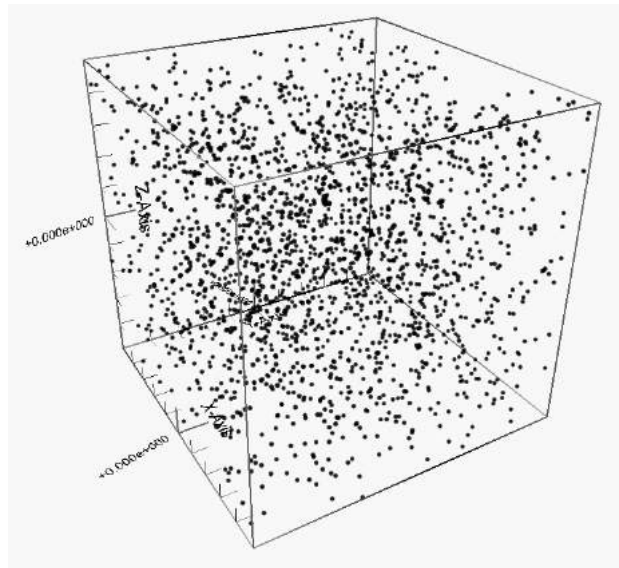
Figure 6.3: Random point distribution in a volume controlled by the radius parameter.

would open a range of new visualization possibilities in a very flexible way.

For example, one could compute an isosurface, which is handled as a *Fiber Grid* object, of a scalar field and sample a second scalar field onto the isosurface. Then, the values of the second scalar field could be visualized via a color map on the isosurface. So one could, for example, analyze a temperature on a zero pressure isosurface. First Steps implementing the a general field on grid evaluator module were started but still have to be completed.

Listing 6.4 shows the class header of the random point distribution. It is a *Vish Module* by deriving from `VObject`. The class gets a *Fiber Grid* handle and inherits functionality to extract all *Fiber Grid* information by being derived from `Fish<Fiber::Slice>`, `Fish<Fiber::Skeleton>` and `Fish<Fiber::Grid>`. To equip the module with an additional *Fiber Field* handle (for the clipping scalar field) a `TypedSlot<Fiber::Field>` is added as an attribute. Additionally the class is derived from `StatusIndicator` which enables some string output inside the GUI of the module. The output is a new *Fiber Grid* object which is stored in the *Fiber Bundle* of the input grid.

Listing 6.4: Random Point Distribution

1

```
 2  class RandomPointDistribution : public virtual VObject,
 3        virtual public Fish<Fiber::Slice>,
 4        virtual public Fish<Fiber::Skeleton>,
 5        virtual public Fish<Fiber::Grid>,
 6        public StatusIndicator
 7
 8  {
 9  public:
10      TypedSlot<double> Radius;
11      TypedSlot<Range>  CutRange;
12      TypedSlot<int>    Seed, NumberOfPoints;
13      TypedSlot<Enum>   CutOff;
14
15      TypedSlot<Field>  CutField;
16
17      VOutput<Grid>      myGrid;
18
19      RandomPointDistribution(const string&name, int p,
20                              const RefPtr<VCreationPreferences>&vp)
21      : VObject(name,p,vp )
22      , Fish<VObject>(this)
23      , StatusIndicator(this)
24      , Radius( this, "radius", 5.0, 1)
25      , CutRange(this, "range", Range(0,1) )
26      , Seed( this, "seed", 0, 1)
27      , NumberOfPoints(this, "nrpoints", 30)
28      , CutOff( this, "cutoff", Enum( "off", "on" ))
29      , CutField(this, "field" )
30      , myGrid(self(), "emitterpoints", GridSelector() )
31      {
32        Seed.setProperty("max", 100);
33        NumberOfPoints.setProperty("max", 20000);
34        NumberOfPoints.setProperty("min", 1);
35          }
36
37      ~RandomPointDistribution() {}
38
39      override bool update( VRequest&R, double precision ) { ... }
40  };
```

Grid Union, Convolution and Transformation

During the work on the seeding modules and this idea of grid convolution which could be done as an operation only on grid objects thoughts came up concerning other possible **operations** that could be done only on *Fiber Grid*s. Here, I present three operations.

The most basic idea was to create a module to **unify** or concatenate given
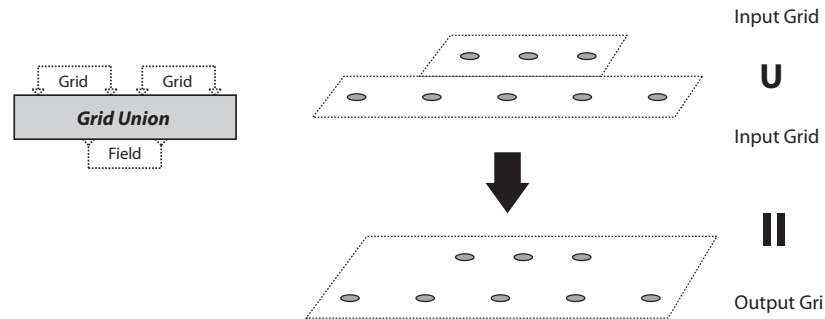
Figure 6.4: Random point distribution in a volume. A scalar field and a range was used to clip the points.

grids. A simple module would take two different *Fiber Grid* objects as input and output the union as one unified grid. *Figure 6.5* illustrates the operation.

Especially when creating seed points this is another tool that widens the range of shapes that can be generated, such as by combining the results of two geometric point distributions, like a circle and a line, in one grid object.

I extracted the **grid convolution** from the old geometric point distribution module and identified it to be a powerful tool for creating complex point shapes. The convolution module takes two grid objects as input and outputs a new grid. The output grid is the result of a copy and translation process. One of the input grids is copied and translated from its origin position $(0.0, 0.0, 0.0)$ to the position defined by the first point of the second grid. This process is then repeated for all points of the second grid. *Figure 6.6* illustrates the operation.

It was recognized that the convolution is symmetrical. It yielded the same result when switching the two input grids. This was not estimated but became clear after some analyses.

When applying the operation several times with different grid objects, complex shapes can be created. For example, to create circles of lines forming a circle again needs three grid objects and two convolution operations, see *figure 7.10*.

Figure 6.5: Unifying Grids results in a new combined Grid. Left: Schematic illustration of the *Vish* module. Input are two *Fiber Grid* objects and output is a unified grid. *Right:* Visualization of the union operation on two grid objects.

Figure 6.7 illustrates another example of multiple convolution. Here, it is used to create a rectangular shape from two lines which is then again convolved along a circle, taken from [1], also included in *appendix C*.

A nice extension would be to add a rotation alignment when doing the copy process. In this case the carrier grid would have to provide a vector for the direction and a second vector for defining the roll. The to-copy-grid would have to provide the two corresponding vectors, but would have to do so for the whole grid object. The z-axis and y-axis could be used intrinsically.

Having such a rotation alignment implemented one could, for example, convolve shapes along a line and keep the rotation such that the copied grid always stays orthogonal to the line direction, similar to an extrusion surface in frequently used in 3D modeling.

As a first approach the **transformations** for changing position and rotation of the shapes was done directly in the geometric point distribution module. This was extracted into a separate module, since then it can be reused on all possible *Fiber Grid*s. Modules for grid object creation like a point distribution now generate objects around the coordinate origin as its center. The grid object can then be translated by using the transformation module afterwards.

The transformation module takes a *Fiber Grid* object as input and a `tvector` for translation, a `tvector` for scaling and a `rotor` for rotation. Output, again, is a grid object.

Besides, the module can be used to create a copy of the original grid,
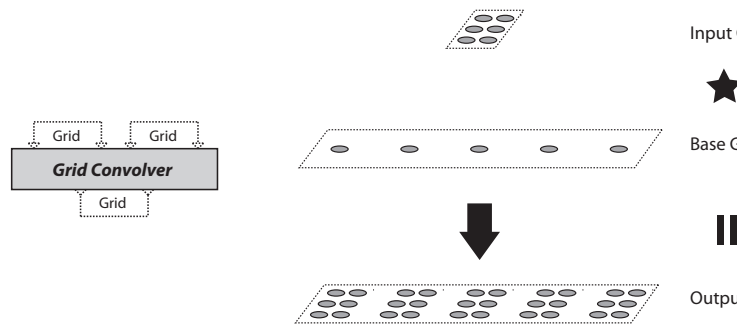
Figure 6.6: Grid convolution results in a new Grid. Left: Schematic illustration of the *Vish* module with two *Fiber Grid* inputs and one grid output. Right: Visualization of the convolution process.
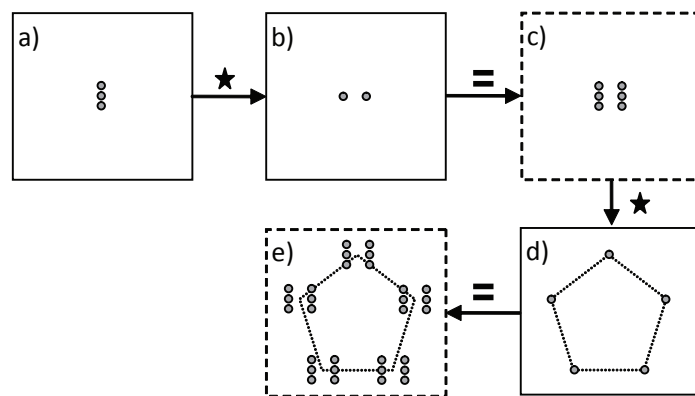

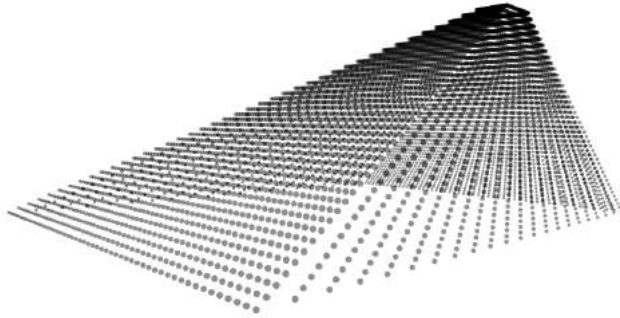Figure 6.7: Using two convolution operations to create seed geometry.

Figure 6.8: Seeding points created on a rectangular shape by a geometric point distribution module are transformed and copied several times by one grid transformation module. Re-applied scaling and positing creates a pyramid grid.

and furthermore it allows multiple copies with re-applied transformations. For example, when the grid object is scaled by 1.2 in x-direction and three multiple copies are generated, the copies have the scalings 1.2, 1.4 and 1.6. Similar rules apply to the other transformations, translation and rotation.

Thus, the module provides another basic tool to generate complex point distributions. *Figure 6.8* shows multiple copies of a rectangle with the repeated translation and scaling, forming now a whole pyramid.

6.1.2   Defining Initial Directions

To solve second-order differential equations, like the geodesic equation, it is necessary on top of this to specify an initial direction besides the initial position. Using the *Vish* environment it develops naturally to define initial directions as a *Fiber Field* of vectors on the *Fiber Grid* object used for defining the initial positions.

Grid Subtraction

A direction vector is simply computed by a vector subtraction. This led to the idea of defining the same operation on grid objects by subtracting all vertices of one grid from the vertices of the other grid. The computed *Fiber Field* is stored to the base grid, where the vectors are pointing from.

The following restriction applies to the module: The numbering of the vertices of the two grid objects must be compatible as well as the number of vertices must be the same. *Figure 6.9* illustrates the operation.
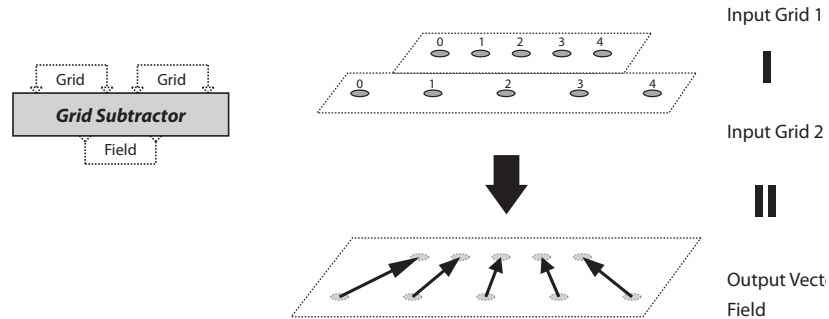
Figure 6.9: Subtracting grid objects results in a vector field. Left: Schematic illustration of the *Vish* module. Two *Fiber Grid* objects are input and one *Fiber Field* object is output. Right: Illustration of the subtraction. Consistent numbering of vertices in both grids is assumed as indicated by the number labels.

The module provides two parameters. One to enable the normalization of the vectors and one for scaling the vectors by a scalar factor. Scaling is done after normalization. So when `normalize` is `yes` and `scale` is 2.5, all vectors have a length of 2.5.

All of the presented modules were utilized to create the different kinds of seeding geometries and vector fields for the visualizations shown in *chapter 7*. For some seeding geometries up to five of the basic seeding modules were combined.

For example, the tube-like seeding of *figure 7.40* was created using two geometric point distributions (a line and a circle) that were combined by a grid convolution, then copied and offset by a grid transformation and finally these two grid were subtracted to yield the vector field for seeding the geodesic computation module.

6.2   Computation

The following section describes the computation and my implementation of streamlines and geodesics. The code structure was re-factored several times, finally yielding the geodesic module. Here, I describe the evolution process as it illustrates important key features of the current solution.

As mentioned in *section 6.1.2*, the whole task of the visualization is split into modules. Seeding points and directions are defined using *Fiber Grid*

and *Fiber Field* objects. The first computation module I implemented was a streamline module. The input was a vector field describing the velocity of a fluid and the seeding point.

The `update` function first did the data access on the *Fiber Bundle* input objects. Then, a main loop solved the differential equation stepping along each streamline after another. It used the `get` function of `LocalFromWorldPoint`, see `chp:interpol`. Finally, a `Fiber Grid` object of lines was output.

I recognized the advantage when adding more data to the line grid as *Fiber Fields*. This information was used for rendering and coloring the streamlines in the rendering module. The interpolated vectors and the magnitude[1] were added as fields to the line grid object. Later, all necessary information that would be needed to interpolate any other data field on the lines was added as well. Local coordinates and fragment IDs were now also stored as fields on the stream line grid.

Before implementing the module for geodesic computation the streamline module was analyzed and several code parts were identified that were also applicable to the geodesic module. Also other types of integral lines could reuse this code. Thus, I implemented a template base class for general integration lines. The parameters to control the computation are:

- Input field

- Input emitter grid (and field)

- Line length

- Step size

- Type of solving the differential equation

- Flags for adding additional Data

I implemented the template class `IntegralLines` over the two template parameters `FieldType` and `LineType`. The `FieldType` specifies the type of the data field that is used to compute the integral lines. In case of a streamline it is a `tvector` for a vector field and in case of a stationary geodesic it is `metric33` for the tensor field. The `LineType` is necessary since different integral lines may be computed on the same `FieldType`. For example, a path-line would have the same `FieldType` as a streamline but a different `LineType`. The `LineType` is a *Type Trait* as described in *section 3.1*.
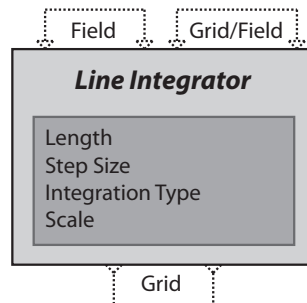
---

[1]Eukildean norm

Figure 6.10: Illustration of a line integration module showing input connections at the top and the output connection at the bottom. The top left input field represents the connection to the field data and the right field/grid connects to the seeding data.

Listing 6.5: The definition of the template class `IntegrationLines`. The module is the basis for different kinds of integral lines in *Vish*.

```
1  template<typename FieldType, typename LineType>
2  class IntegralLines : public virtual VObject,
3                        public virtual Fish<Field>,
4                        public StatusIndicator
5  {
6  /* ... */
7      TypedSlot<Grid>    StartGrid;
8      TypedSlot<Field>   StartField;
9
10     TypedSlot<double> LineLength;
11     TypedSlot<double> StepSize;
12     TypedSlot<double> Scale;
13
14     TypedSlot<Enum>   IncludeData;
15     TypedSlot<Enum>   IncludeMagnitude;
16     TypedSlot<Enum>   UseDop853;
17
18     VOutput<Grid>      myIntegralLines;
19  /* ... */
```

The *Vish* module is equipped with an input field handle, *line 3*, and has input handles for the initial condition, *line 7* and *line 8*. If the `StartField` member is connected, the corresponding grid objects to define the positions is extracted for `StartField` as well. The `StartGrid` is overridden in that case. `LineLength` and `StepSize` control the length and the interval step length of the integral line. `StepSize` is overridden when an adaptive solver is used for

integration. The enumerations in *line 13* and *line 14* flag if additional data should be stored as fields on the output grid. `UseDop853` switches between first order Euler and adaptive 8th order Runge Kutta integration.

It follows an overview of the tasks done in the `update` function:

1. Get input field information

2. Get emitter grid or emitter field information

3. Create grid output name and check if re-computation is necessary

4. Create automatic seed points if no emitter grid is available

5. *Do the integration*

6. Bake a new Grid object that carries the actual lines

All these tasks are the same for streamlines and geodesics. Besides operating on different types of data, the only task that has to be specialized for each `LineType` is *task 6*, the actual integration.

Task 1 and 2:
The handles to the *Fiber Grid* and *Fiber Field* objects are extracted by using `GridSelector` and `FieldSelector` as described in *section 5.2.2*

Task 3:
Here it is checked if an output grid is already existent from an earlier computation. Since the line grid object is stored in the *Fiber Bundle* it can be tried to get such a grid from the bundle, see *line 4* in *listing 6.6*. If no valid grid is returned a new one has to be computed. Otherwise it is checked whether the input field, the line length, the step size, the connection to the input field and the emitter grid are older than the coordinate field of the output grid. If they all are older a `return` exits the `update` function of the module. If one of these objects is younger than the found grid at the output it has to be recomputed.

Listing 6.6: Source code that checks if the integration line grid at the output has to be recomputed because of changes of the input field, the seed point grid or module parameters.

```
1  string grid_name = FieldSelection.Gridname() + "_" + Name();
2
3  Info< Fiber::Slice > ActualSlice = FieldSelection.FieldSource;
4  RefPtr<Grid> OutGrid = (*ActualSlice.second)(grid_name);
5
6  if ( OutGrid )
7  {
```

```
 8    if (RefPtr<Field> OutCrds = OutGrid->getCartesianPositions())
 9    {
10       if (InputToIntegrateField  ->isOlderThan( *OutCrds ) &&
11           LineLength->age(Context).isOlderThan( *OutCrds ) &&
12           StepSize  ->age(Context).isOlderThan( *OutCrds ) &&
13           ConnectionAge()          .isOlderThan( *OutCrds ) )
14            {
15                   if (EmitterCoordinates)
16          {
17            if (EmitterCoordinates->isOlderThan( *OutCrds ) )
18               return setStatusInfo(Context,
19                    "Reusing previous computation");
20          }
21       else
22          return setStatusInfo(Context,
23                "Reusing previous computation (no emitter)");
24 } } }
```

To equip a class with the aging functionality one derives it from the `Memcore::Ageable` base class of the *Vish* environment. `Field` and `Grid` are derived from `Agable` and can be compared to other ageable objects, see *line 10* and *line 17*. The `TypedSlot` has an age as a member. Here, the `Context` on which parameters depend is important, see *line 11* and *line 12*.

Task 4:

If no emitter grid or emitter field is found a new emitter grid is created. The positions of the seed points are then created along a diagonal of the bounding box of the connected input data field.

Task 5:

The main loop uses a *Type Trait*[2] class for integration, see *listing 6.7 line 1*. The integrator has to be initialized by a handle to a container storing the computed lines, a handle to a data field, a `LocalPointFinder`, the step size for one integration step, a flag for the integration type, the number of steps and the time when the integration is started.

Listing 6.7: Main loop

```
1 LineIntegrator<FieldType,LineType>  Integrator(
2                                     Lines, FieldSelection,
3                                     LocalPointFinder,
4                                     step_size, Dop(),
5                                     line_length, time );
6 int step_counter = 0;
7
8    while( hasfinished == false && step_counter < line_length )
9    {
```

---

[2] *section 3.1*

```
10  /* ... */
11         hasfinished = MyLineIntegrator.advanceBreadthFirst();
12         step_counter++;
13      }
```

The `LineIntegrator` template can be specialized if necessary. For streamlines and geodesics this was not required and the general template definition was sufficient. Calling the `integrateBreadthFirst` function, *line 1*, advances all integral lines by one integration step. This is repeated until the `LineLength` is reached or all lines cannot be computed further, *line 8*.

```
1  template<class FieldType, typename LineType>
2  class LineIntegrator
3  {
4  /* ... */
5  AtomicIntegrator<FieldType, LineType> AI;
6  /* ... */
7
8  bool advanceBreadthFirst( )
9  {
10 unsigned end_counter = 0;
11
12     if(dop)
13        AI.initDop853( Lines.size() );
14
15     for(unsigned p = 0; p < Lines.size(); p++)
16     {
17 /* ... */
18     if( !dop )
19        test = AI.doEuler( *IntegrationLine, time );
20     else
21        test = AI.doDop853( *IntegrationLine, time, p );
22 /* ... */
23     }
24     return ( end_counter == Lines.size() ) ? true : false;
25 }
26 };
```

The `LineIntegrator` uses an instance of the `AtomIntegrator` to compute the integration steps, see *line 5*. The `AtomIntegrator` is a *Type Trait* as described in *section 3.1* and has to be specialized and implemented for each kind of integration line. The loop in *line 10* iterates over all integration lines of the Line container and advances each line by one step. If all lines cannot be computed further, for example because they have stepped outside the data field domain, `advanceBreadthFirst` returns `true`, otherwise `false`.

Task 6:
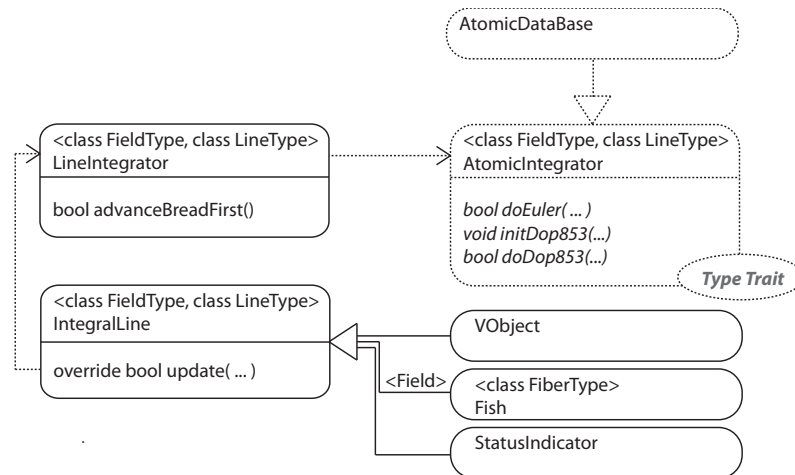Now the computed lines have to be stored as a *Fiber Grid* object into the

Figure 6.11: UML diagram of class relationships for integral lines. The *Vish* module `IntegralLines` uses a reference of `LineIntegrator` in the update function. The `LineIntegrator`, finally, uses a reference of the *Type Trait* `AtomicIntegrator` doing the integration. An `AtomicDataBase` is provided to equip the `AtomicIntegrator` with data structures needed for computation which are initialized by the `LineIntegrator`.

Fiber Bundle. Here, it is illustrated how the **vertices**, the **connectivity** and the **interpolated field data** of the lines are stored. For a more detailed description like storing other fields on the grid, *chapter 4.* First, a new *Fiber Grid* object is prepared to be added to the *Fiber Bundle* of the input data field for storing the lines:

```
RefPtr<Grid> IntegralLineGrid =
            FieldSelection.theSourceBundle->newGrid();
Grid&LineGrid = *IntegralLineGrid;
```

Creating the `IntegralLineGrid` using the `newGrid` function does not add the grid to the *Fiber Bundle*. The grid is added later, when the data has been baked completely. The pointer to the bundle is extracted from the `FieldSelector` of the input data field. Thus, the grid will be added into the bundle hosting the input data field.

Next, a new *Fiber Field* is created to store the vertex data of the integral lines:

```
RefPtr<Field> LinePositions = new Field();

RefPtr<MemArray<1, point> > Pts = new MemArray<1,point>(
                                    MIndex( nPoints ) );
```

```
5
6  RefPtr<Chunk<point> > data = Pts->myChunk();
7  std::vector<point>&LineCoordinates = data->std_vector();
8
9  index_t Pt = 0;
10
11 /* loop, store points in 1D Array LineCoordinates */
12    LineCoordinates[Pt] = (PointsPerLine[ver]).location;
13    Pt++;
14 /* ... */
15    LinePositions->setPersistentData( Pts );
```

After creating a 1D `MemArray` of the correct size, *line 1*, its data access using a `std::vector` is prepared (*line 6* and *line 7*). All vertices are then stored sequentially in the `MemArray`, *line 12*. The *Fiber Field*'s `setPersistantData` function stores the data into the `Field`, *line 15*. Using this function ensures that data is always kept in memory for caching. Here, room for improvement remains. A more sophisticated mechanism might be appropriate.

The created data field of the vertex is now added to the grid object `LineGrid`:

```
1  Skeleton&LineVertices       = LineGrid.makeVertices(3);
2  RefPtr<Chart>  myCartesian = LineGrid.makeChart(
3                              typeid(Fiber::CartesianChart3D) );
4  Representation&VerticesAsCartesian = LineVertices[myCartesian];
5    VerticesAsCartesian.setPositions( LinePositions );
```

Since the field describes the vertices of the grid a skeleton of vertices is created, *line 1*, the vertices being represented in Cartesian coordinates. Thus, a corresponding `Chart` object is generated and used to create a Coordinate representation. Finally, the *Fiber Field* `LinePositions` is added to the grid by calling `setPosition` on the representation passing the field. The `makeVertices` function is used for convenience substituting the creation of a skeleton of 0 dimensionality and 0 index depth, *chapter 4*.

Next, the connectivity of the vertices is stored:

```
1  typedef MemArray<1, std::vector<index_t> > EdgesArray_t;
2  Ref<EdgesArray_t> EdgesArray( NumberOfLines );
3  MultiArray<1, std::vector<index_t> >&Edges = *EdgesArray;
4  index_t vert_count;
5
6  /* loop over number of lines */
7    vert_count = 0
8    /* loop over vertices in one line */
9        Edges[ line ][ver] = vert_count;
10       vert_count++;
11
```

```
12  Skeleton&LineEdges                = LineGrid [ SkeletonID (1 ,1)  ];
13  Representation&EdgesAsVertices = LineEdges [ LineVertices  ];
14      EdgesAsVertices.setPositions ( new Field ( EdgesArray ) );
```

The connectivity is stored as a sequence of indices for each line running from
0 to the length of the line minus 1. The skeleton to describe an edge has a
dimensionality of 1, since an edge is a one dimensional object, and an index
depth of 1 since it requires one lookup to get to a vertex position, *line 12*.
The representation is on the line's vertices, *line 13*. Finally a new data field
holding the edge data is created and added to the grid in *line 14*.

This is an example of storing data on the vertices where data is of template
type `Fieldtype`:

```
1   typedef MemArray<1, FieldType> LinesDataArr_t;
2   Ref<LinesDataArr_t>  LinesData ( nPoints );
3   MultiArray<1,FieldType>&LineData = *LinesData ;
4   index_t vert_count = 0;
5
6   /* loop over vertices */
7       LineData [ MIndex ( vert_count ) ] = Line [ ver ] . data ;
8       vert_count++;
9
10      VerticesAsCartesian [ Fieldname ( Context ) ]−>
11                                  setPersistentData ( LinesData );
12      VerticesAsCartesian [ TangentialVectorFieldName ]−>
13                                  setPersistentData ( LinesData );
```

Data is stored in the same layout as the vertices. The same representation
`VerticesAsCartesian` is used specifying a name to identify the data field.
More than one name used. The data field is not created twice in the bundle,
just an alternative name ist stored.

Finally the baked *Fiber Grid* is inserted into the bundle at a certain *Fiber
Slice:*

```
1       currentSlice.getSlice()−>insert ( grid_name , IntegralLineGrid );
2   GridSelector GS( grid_name , FieldSelection.BundleSource () );
3       myIntegralLines << Context << GS
```

Also, the a handle to the line grid is output at the `VOutput` of the integration
line module.

Next is the description of computation and implementation of stream-
lines and geodesics. Other integral lines can also be based on the provided
`IntegralLines` template classes: Implementing pathlines in vector fields or
gradient lines in gradient or scalar fields is straightforward with a minimal
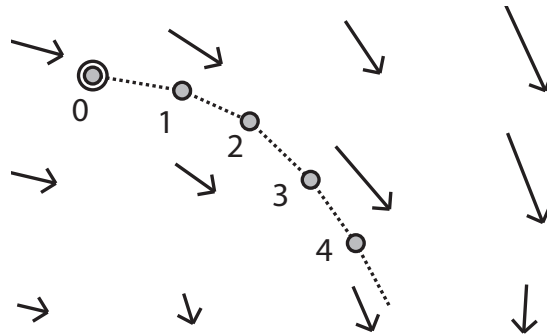effort in producing lines of code.

Figure 6.12: Computing a streamline in a vector field describing the velocity of a fluid. Starting at an initial position and following the direction of the vector field by a certain step size solves the differential equation defining the streamline curve.

6.2.1   Computing First Order Integration Lines

As described in *section 2.3* streamlines are a common tool to visualize flow in fluid dynamics. Streamlines are integral lines of first order, see *equation (2.75)*.

To solve the equation I first implemented a simple explicit Euler integration scheme. At the initial position the local coordinates of the vector field are computed. These are then used to compute the vector at that position by linear interpolation of the vector field.

According to *equation (2.75)*, the vector at that position is equal to the derivative of the streamline. A new point of the streamline can now be computed by stepping into the direction of the vector. The step size is controlled by the user. *Figure 6.12* illustrates the process. Computation starts at point 0 and continues following the vector field.

Implementation is done by specializing the `AtomicIntegrator` class provided by the integral architecture. A new type trait, the streamline, has to be introduced as well:

Listing 6.8: Layout of the streamline specialization of `AtomicIntegrator`.

```
struct  Streamline {};

template<>
class  AtomicIntegrator<tvector, Streamline >:public  AtomicDataBase
{
std :: vector<dop853<TangentialDifferentialEquationLinear >>LineInts
/*  ...  */

```

```
 9    bool doEuler( std::vector<IntegrationPoint<tvector> >&IntLine,
10                  const double&time );
11
12    bool initDop853( const int number_of_lines );
13
14    bool doDop853(std::vector<IntegrationPoint<tvector> >&IntLine,
15                  const double&time, const unsigned line_nr );
16 };
```

The `doEuler` function finally implements the integration in a few lines of source code. Here follows the main part:

Listing 6.9: Excerpt of `doEuler` for streamlines.

```
 1 bool test=LocalPointFinder−>get( IntLine[last_index].location,
 2                                  localPoint );
 3 /* ... */
 4    IntLine[ last_index ].setData( InterpolData, float_index,
 5                                   FragName );
 6
 7 Interpolate<3, tvector, LinearIpol<tvector> >
 8            myField( *ToIntArr, localPoint.first );
 9
10 tvector dir = myField.eval();
11
12    NewPoint = IntLine[ last_index ].location + step_size * dir;
13 /* ... */
```

First the local point coordinates are computed using the `get` function as described in *section 5.4*. Then the vector is computed using interpolation template classes provided by *Vish*. Finally the Euler step is done in *line 12*. The new point is then stored on the integration line, which is the basis for the computation of the next step. One line is represented by a vector of `IntegrationPoint`s, which is a simple template container class to collect all information computed at one point of the integration line, here: point world coordinates, the interpolated data (vector), the local coordinates, the id of the multi-block.

The Euler method is fast if high accuracy of the solution is not important. To achieve high accuracy one has to use a very small step size for integration.

"We have studied its convergence extensively in Section 1.7 and have seen that the global error behaves like Ch, where C is a constant depending on the problem and h is the maximal step size. If one wants a precision of, say, 6 decimals, one would thus need about a million steps, which is not very satisfactory. ..." [34]

Runge-Kutta (RK) methods are based on Euler steps. More than one Euler step is done and then combined into one RK step by several coefficients

[34]:

Let s be an integer (the "number of stages") and a_{21}, a_{31}, a_{32}, ..., a_{s1}, a_{s2}, ..., $a_{s,s-1}$, b_1, ..., b_s, c_2, ..., c_s be real coefficients. Then the method

$$k_1 = f(x_0, y_0) \tag{6.1}$$
$$k_2 = f(x_0 + c_2 h, y_0 + h a_{21} k_1)$$
$$k_3 = f(x_0 + c_3 h, y_0 + h(a_{31} k_1 + a_{32} k_2))$$
$$...$$
$$k_s = f(x_0 + c_s h, y_0 + h(a_{s1} k_1 + ... + a_{s,s-1} k_s - 1))$$
$$y_1 = y_0 + h \cdot (b_1 k_1 + ... + b_s k_s)$$

is called an s-stage explicit Runge-Kutta method (ERK) for (1.1).

Usually, the c_i satisfy the conditions ...

$$c_i = \sum_{j=1}^{i-1} aij. \tag{6.2}$$

Such a method converges much faster to the correct solution. When choosing the coefficient thoughtfully the error decreases exponentially:

$$\underbrace{||y(x_0 + h) - y_1||}_{local error} \; <= K h^{p+1}, \tag{6.3}$$

with p denoting the order of the RK scheme. A famous example for coefficients is "the" Runge-Kutta method of order 4. The coefficients are usually given in tabular form:

0				
c_2	a_{21}			
c_3	a_{31}	a_{32}		
$\vdots$	$\vdots$	$\vdots$		
c_s	a_{s1}	a_{s2}	...	$a_{s,s-1}$
	b_1	b_2	...	b_{s-1}	b_s

0				
$1/2$	$1/2$			
$1/2$	0	$1/2$		
1	0	0	1	
	$1/6$	$2/6$	$2/6$	$1/6$

This is a 4 stage RK method and thus needs 4 evaluations of the function to do one integration step which, in case of the streamlines, requires computing local coordinates and doing the interpolation 4 times. Since the local error bound decreases exponentially with the order, the step size can be increased and, thus, much fewer total steps are needed for integration compared to simple Euler integration.

Usually the analytically correct solution of a function is not known and local errors cannot be computed correctly. Nonetheless, a local error can be
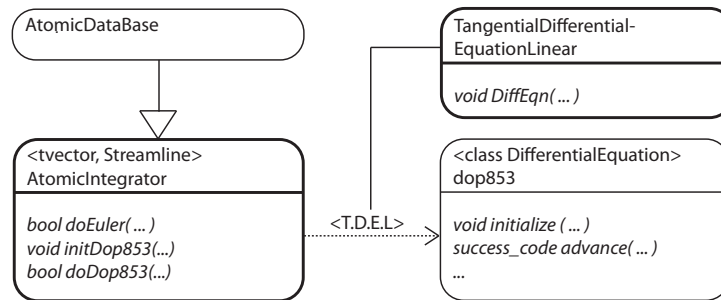
Figure 6.13: UML diagram of the streamline specialization of `Atomic-Integrator`. The integrator uses the Dop853 template class to solve the differential equation, which is defined by a function `DiffEqn` in a class used as template type of the solver.

estimated by comparing RK schemes of different orders. Such local error estimates can then be used to choose a step size adaptively, using as large steps as possible while guaranteeing a low error bound.

Vish provides a solver call Dop853 that was rewritten from the original FORTRAN algorithm found in [34]. It uses RK schemes of order 3 and 5 for error estimation and step size control. The actual step is finally of order 8, which is highly accurate. "The performance of this code, compared to methods of lower order, is impressive." [34]

Figure 6.13 shows the class relationships of the integration class and the `Dop853` solver. `Dop853` is a very generic solver suitable for systems of differential equations. Once set up with initial conditions the `advance` function is called to compute one RK853 step.

The differential equations are defined in a class that is then used as a template parameter of the solver class. Thus, the code is in-lined into the solver at compile time and can be optimized well, while still providing high code re-usability. Unfortunately, C++ does not allow to provide any kind of interface or "virtual function" for classes passed as template parameter.

For solving the streamlines, the differential equation was introduced by following class:

Listing 6.10: Streamline differential equation for the Dop853 solver.

```
1  struct     TangentialDifferentialEquationLinear
2  {
3  /* ... */
4  void DiffEqn(int nEquations, real s, const real *q, real *dq_ds)
5  {
6  Eagle::point3 Q;
```

```
7        for(int k=0; k<3; k++)
8            Q[k] = q[k];
9
10 /* get local coords at Q, compute interpolated tvector DataDir */
11
12        for(int i=0; i<3; i++)
13            dq_ds[i] = DataDir[i];
14 }
15 };
```

Where *line 13* represents the streamline differential equation *equation (2.75)*.

The `AtomicIntegrator` holds a `std::vector` of dop853 solvers, one for each streamline, see *listing 6.8*. The size of the vector is adapted in the `initDop853` function. The `doDop853` function now first initializes the according solver and then computes one step using the `advance` function of `dop853`:

Listing 6.11: Excerpt of `doDop853` for streamlines.

```
1 int last_index = IntLine.size()-1;
2 dop853<TangentialDifferentialEquationLinear>&LI=LineInts[line_nr];
3
4    if( last_index == 0 )
5        /* ... init solver LI */
6
7    switch( LI.advance() )
8    /* ... */
9    case Finished:
10    {
11    /* ... */
12        IntLine[ last_index ].setData( LI.DataDir, old_local,
13                                       LI.FragName );
14
15    point NewPoint( LI.f(0), LI.f(1), LI.f(2) );
16
17        IntLine.push_back( IntegrationPoint<tvector>() );
18        IntLine[ last_index + 1 ].setLocation( NewPoint );
19    /* ... */
20    }
```

Depending on the line number the corresponding equation is retrieved, *line 2* and advanced in *line 7*. Here, all streamline line related data is stored at the current point and the new point is pushed onto to the integration line.

After the algorithms have been implemented via the `AtomicIntegrator` an instance of the template has to be created and a `VCreator` has to be provided for *Vish*, as described in *section 5.2*:

Listing 6.12: Equipping *Vish* with a streamline module.

```
1  typedef  FieldLines :: IntegralLines<tvector , FieldLines :: Streamline>
2           FlowTassels3 ;
3
4  static  Ref<VCreator<FlowTassels3 ,  AcceptList<Fiber :: Field> > >
5  MyFlowCreator("Compute/Streamlines" , ObjectQuality ::EXPERIMENTAL);
```

Applications of the streamline algorithm can be found in *section 7.1* and *section 7.2*. *Section 7.1* includes a comparison between Euler and Dop853 integration.

I chose to equip the integration line modules with these two different integration methods to give the user the possibility to either compute very fast but inaccurate results, to get an impression of the solution, or to compute highly accurate results, when necessary.

6.2.2   Computing Second Order Integration Lines

With all the infrastructure available the next step was to implement geodesics. First, **spatial geodesics** were considered, thus, neglecting possible time components of the tensor. The metric tensor g is then written in a 3×3 matrix form.

In addition to the seed points, corresponding seed directions are necessary to solve the second order differential equation. Instead of a seeding grid a seeding field is connected to the integral lines *Vish* module, see *figure 6.10*. The seeding grid is extracted from the *Fiber Field* in that case.

Again, the integral line template base classes are used and only the template specialization of `AtomicIntegrator` have to be provided. As for the streamlines, the simple explicit Euler and the adaptive Dop853 are implemented.

To solve the geodesic equation, see *equation (2.53)*, first the Christoffel Symbols must be evaluated at the current position in the metric field given in Cartesian coordinates.

A convenience function to retrieve the Christoffel Symbols was introduced that uses a Christoffel Symbol class for computation. The convenience function was first implemented for use with the field type of a `metric33` only. Later it was rewritten such that the template parameter `T` is used as a field type.

Listing 6.13: Convenience function to compute Christoffel Symbols

```
1  template<class T>
2  bool getChristoffelXYZ( ... )
3  {
4  bool test = LocalPointFinder ->get( CurrentPoint , localPoint );
5  const Eagle :: point3 &lp = localPoint.first ;
```

```
6  /* ... */
7
8  T g, g_x, g_y, g_z;
9
10 Interpolate<3, T, CubicIpol<T> > myField( *ToIntArr0, lp );
11     g = myField.eval();
12
13 RefPtr<Fiber::Field>field=FieldSelection.getCartesianPositions();
14 RefPtr<Fiber::DirectProductMemArray<Eagle::point3,3> >
15     FData = field->getData();
16
17 Eagle::point3 delta = FData->Delta();
18
19 Interpolate<3, T, CubicIpol<T>, double,
20             NoDelimiter<T>, 0  > myField_x( *ToIntArr0, lp );
21     g_x = myField_x.eval();
22
23     g_x  /= delta[0];
24
25 /* ... similar for y and z ... */
26
27 Christoffels = new ChristoffelXYZ( g, g_x, g_y, g_z );
28
29 /* ... */
30 }
```

The `getChristoffel` function computes the local point using the `LocalPointFinder`, see *section 5.4*, which is then used to do a cubic interpolation of the metric tensor field, *line 10*. *Vish*'s interpolation classes also can deliver derivatives. *Line 19* illustrates how this is done. The last template parameter of `interpolate` specifies the axis direction of the derivation (x direction is index 0). The derivative is computed in index coordinates and thus has to be normalized by the interval length. Therefore, the interval distance is retrieved in *line 17*. A direct product array is utilized. By now, this technique is limited to uniform grids. For supporting other grid types the code between *line 13* and *line 26* has to be extended.

The `ChristoffelXYZ` class stores the metric tensor and its x, y and z derivatives and provides an operator to compute a component on demand:

Listing 6.14: Operator of `ChristoffelXYZ` to compute Christoffel Symbols on demand.

```
1  const double operator()( const unsigned mu,
2                           const unsigned nu,
3                           const unsigned lambda) const
4  {
5  double gamma = 0.0;
```

```
6
7     if (mu > T::Dims−1 || nu > T::Dims−1 || lambda > T::Dims−1)
8             return 0.0;
9
10    for (unsigned i = 0; i < T::Dims; i++)
11        gamma += g_inv(lambda, i) * ( g_diff[nu](mu,i) +
12                   g_diff[mu](nu,i) − g_diff[i](mu, nu) );
13
14    gamma /= 2;
15
16    return gamma;
17 }
```

Here, `g_diff` are pre-computed differences of the derivatives of the tensor and the tensor itself, *section 2.1.7* for exploring the structure of the Christoffel Symbols. The `g_inv` is the co-metric, the inverse of the metric. The inversion is done using an explicit formula, see [42]. The operator works for n-dimensional metrics. The dimension can be extracted from the template parameter `T`, see *line 10*.

Listing 6.15: Convenience function to compute $\ddot{q}$.

```
1  template<class T> typename
2  Eagle::Coordinates <typename T::Chart_t >::vector getQddot(
3     RefPtr< ChristoffelXYZ<T> >& Gamma,
4     typename Eagle::Coordinates<typename T::Chart_t >::vector&q_dot)
5  {
6  typename Eagle::Coordinates<typename T::Chart_t >::vector q_ddot;
7
8     q_ddot.set(0.0);
9
10    for( index_t lambda = 0; lambda < T::Dims; lambda++)
11    for( index_t mu = 0; mu < T::Dims; mu++)
12    for( index_t nu = 0; nu < T::Dims; nu++)
13        q_ddot[lambda] −= (*Gamma)( mu, nu, lambda ) *
14                          q_dot[mu] * q_dot[nu];
15
16    return q_ddot;
17 }
```

This function heavily uses template functionality. The type of the returned vector is extracted from the chart type of the `Coordinates` template, see *line 2* and *line 5*. In case of `T=metric33` this is a `Eagle::PhysicalSpace::vector` and in case of `T=metric44` this is a `Eagle::STA::vector`. The dimension is directly extracted from `T` and used in the loop control.

Having the $\ddot{q}$ available the geodesic equation can be solved, for example, using two Euler steps. One for computing the tangential vector $\dot{q}$ and one

for computing the next point x of the line. x_0 and $\dot{q}_0$ must be given as initial conditions:

$$
\begin{aligned}
\dot{q}_{i+1} &= \dot{q}_i + h\ddot{q}_i \\
x_{i+1} &= x_i + h\dot{q}_{i+1}
\end{aligned}
\tag{6.4}
$$

The Euler is implemented in the `doEuler` function of the `AtomicIntegrator` which is only partially template-specialized:

Listing 6.16: Layout of the partial geodesic specialization of `AtomicIntegrator`, compare *listing 6.8.*

```cpp
struct Geodesic{};

template<>
class AtomicIntegrator<T, Geodesic>:public AtomicDataBase
{
typedef Traum::dop853<GeodesicDifferentialEquation<T>>DiffSolver;
typedef typename T::Chart_t Chart_t;
typedef typename Coordinates<Chart_t>::point point;
typedef typename Coordinates<Chart_t>::vector tvector;

std::vector< DiffSolver > LineIntegrators;
/* ... */

    bool doEuler( std::vector<IntegrationPoint<T> >&IntLine,
                 const double&time );

    bool initDop853( const int number_of_lines );

    bool doDop853(std::vector<IntegrationPoint<T> >&IntLine,
                 const double&time, const unsigned line_nr );
};
```

The data field type is still a template parameter. Via the chart of the template type local typedefs define a `point` and `tvector`.

The `doEuler` function utilizes the convenience functions for the computation of the Christoffel symbols and $\ddot{q}$.

Listing 6.17: Excerpt of `doEuler` for geodesics.

```cpp
/* ... */
RefPtr<ChristoffelXYZ<T> > Gamma;
T g_curr;
pair<Eagle::PhysicalSpace::point, string>localPoint;

bool test = FieldLines::getChristoffelXYZ<T>( P_curr,
            LocalPointFinder, FieldSelection, step_size,
            Gamma, g_curr, localPoint );
```

```
 9  /* ... */
10  tvector Qdot;
11      if( last_index == 0 )
12          Qdot = getQdot<T>( IntegrationLine[last_index].direction,
13                             g_curr );
14      else
15          Qdot = IntegrationLine[last_index].direction;
16
17  tvector Qddot = getQddot<T>(Gamma, Qdot);
18
19  /* first Euler */
20  tvector Qdot_next = Qdot + step_size * Qddot * scale;
21
22      IntegrationLine[last_index].setData(g_curr, localPoint.first,
23                                          localPoint.second );
24
25  point P_new = IntegrationLine[last_index].location;
26  tvector tmp = step_size * Qdot_next;
27
28  /* second Euler */
29      P_new += tmp;
30
31  /* check if it is a valid step and a valid point*/
32
33  IntegrationLine.push_back( IntegrationPoint<T>() );
34  IntegrationLine[ last_index + 1 ].setLocation( P_new );
35  IntegrationLine[ last_index + 1 ].setDirection( Qdot_next );
```

First, the Christoffel Symbols are retrieved in *line 6* and used to compute $\ddot{q}$ in *line 17*. Two Euler steps are necessary to get the tangential direction and finally the new point on the integration line, according to *equation (6.4)*.

When doing the first Euler step an additional scale factor was introduced. Applying a scale here is equivalent to scaling the metric tensor field by a scalar factor, which was necessary for the MRI dataset presented in *section 7.5*.

Before storing the computed data in the `IntegrationLine` container it is checked if the integration step is a valid one. If the new point is located outside of the bounding box of the data field the line is stopped. Also, when certain criteria regarding the change in length and angle to the previous Euler step are contravened it is stopped, *section 7.3*.

For implementing the Dop853, again, a class describing the differential equation has to be made available. *Vish* provides a base class for differential equations of second order:

```
1  template<class T>
2  struct GeodesicDifferentialEquation :
3  public VirtualSecondOrderDiffEquation <double>
4  void Accel(real s, const real *x, const real *v, real *d2x_ds2)
```

```
5  {
6  point Q;
7  tvector q_dot;
8
9      for(int k=0; k < point::Dims; k++)
10     {
11         Q[k] = x[k];
12         q_dot[k]    = v[k];
13         DataDir[k] = v[k];
14     }
15 bool test = FieldLines::getChristoffelXYZ<T>( Q, LocalPointFinder ,
16             FieldSelection , step_size , Gamma,
17             CurrentMetric , localPoint );
18
19     if(! test) return;
20
21     tvector qdd = getQddot<T>( Gamma, q_dot );
22
23     for(int k=0; k < point::Dims; k++)
24         d2x_ds2[k] = qdd[k];
25 }
26 };
```

This is similar to *listing 6.10*, but here the second derivative is computed, see *line 24*. Again, the convenience functions are utilized.

The implementation of the `doDop853` function is almost identical to the one of the streamlines. The only difference appears in the initialization of the `doDop853` solver, since, also the tangential directions have to be set.

A `typedef` is used to initialize the `AtomicIntegrator` with T=metric33 and the `VCreator` is used to register the integration line module to *Vish*.

```
1  typedef FieldLines::IntegralLines<metric33, FieldLines::Geodesic>
2  GeodStationary3;
3
4  static Ref<VCreator<GeodStationary3 , AcceptList<Fiber::Field>>>
5  MyG33Creator("Compute/Geodesics33" , ObjectQuality::EXPERIMENTAL);
```

Figure 6.14 illustrates the class relationships of the classes regarding the geodesic computation, similiar to *figure 6.13*.

Spatial geodesics can be used to visualize $3D$ metric tensor field data. An application is stemming from medical imaging. Here, a diffusion tensor field can be visualized using spatial geodesics, *section 2.4* and *section 7.5*.

The implementation of **4D geodesics** is identical to the spatial geodesic implementation because the partial template specialization was utilized. One difference arises in the initialization. The initial points and directions are given in three space dimensions to the integration line module. The time
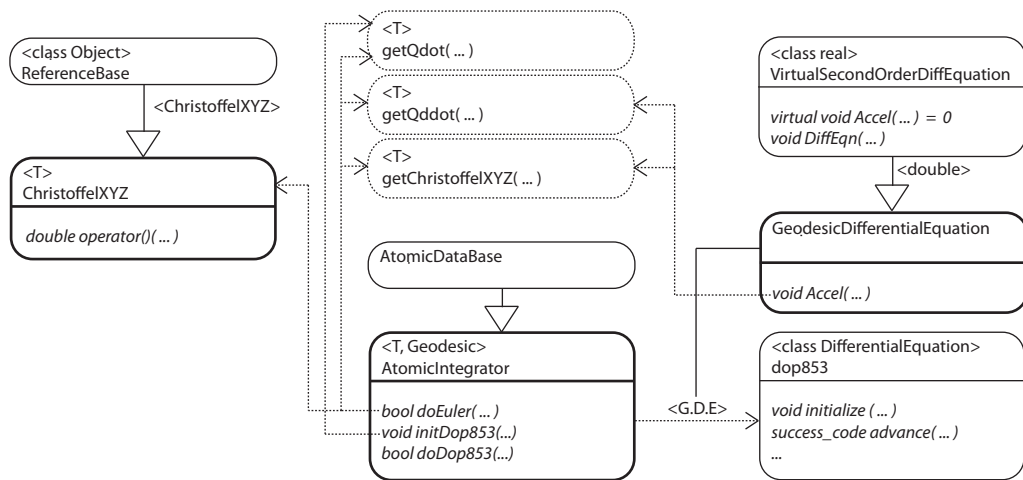
Figure 6.14: UML diagram of the geodesic specialization of `Atomic-Integrator`. The specialization is partial, thus making it applicable for `T=metric33` and `T=metric44` tensor fields. The integrator uses the Dop853 template class to solve the differential equation, which is defined by a function `Accel` in a class derived from `VirtualSecondOrderDiffEquation` used as template type of the solver. Christoffel symbols are encapsulated into a separate class and the convenience function `getChristoffelXYZ()` is provided for their computation. Another convenience function computes $\ddot{q}$: `getQddot()`.

component of the point is just set to zero but the time component of the initial direction needs proper treatment.

The 4th parameter of the initial direction has to be computed dependent on the metric tensor at its position to satisfy the geodesic requirement, see *equation (2.32)* in *section 2.1.7*.

$$g(\dot{q}, \dot{q}) = 0 \tag{6.5}$$

Written in *txyz* coordinates and using $v := \dot{q}$:

$$g_{\mu\nu} v^\mu v^\nu = 0 \tag{6.6}$$

After expansion:

$$
\begin{aligned}
g_{tt} v^t v^t &\quad + g_{tx} v^t v^x &\quad + g_{ty} v^t v^y &\quad + g_{tz} v^t v^z &\quad + \\
g_{xt} v^x v^t &\quad + g_{xx} v^x v^x &\quad + g_{xy} v^x v^y &\quad + g_{xz} v^x v^z &\quad + \\
g_{yt} v^y v^t &\quad + g_{yx} v^y v^x &\quad + g_{yy} v^y v^y &\quad + g_{yz} v^x v^z &\quad + \\
g_{zt} v^z v^t &\quad + g_{zx} v^z v^x &\quad + g_{zy} v^z v^y &\quad + g_{zz} v^z v^z &\quad = 0
\end{aligned} \tag{6.7}
$$

Collecting terms for v^t and using the symmetry of g ($g_{\mu\nu} = g_{\nu\mu}$) yields the quadratic equation

$$(v^t)^2 + v^t \frac{B}{A} + \frac{C}{A} = 0 \tag{6.8}$$

with

$$
\begin{aligned}
A &= g_{tt} \\
B &= 2(g_{tx} + g_{ty} + g_{tz}) \\
C &= (g_{xx} v^x v^x + g_{yy} v^y v^y + g_{zz} v^z v^z) + 2(g_{xy} v^x v^y + g_{xz} v^x v^z + g_{yz} v^x v^z).
\end{aligned} \tag{6.9}
$$

Finally, leading to the two possible solutions for v^t

$$v^t_{1,2} = -\frac{B}{2A} + -\sqrt{\left(\frac{B}{2A}\right)^2 - \frac{C}{A}} \tag{6.10}$$

with $v^t_1 < v^t_2$.

The implementation in the `AtomicIntegrator` of the geodesics always chooses the v^t_2 as valid initial condition.

This computation is encapsulated in the `getQdot()` function, see *listing 6.17, line 12*. It is only called for initialization. An additional feature was added to the function. If the initial direction is set to $(0, 0, 0)$ then the time coordinate will be set to 1.0 and the geodesics will represent the motion of particles initially being at rest instead of photons moving at the speed of light.

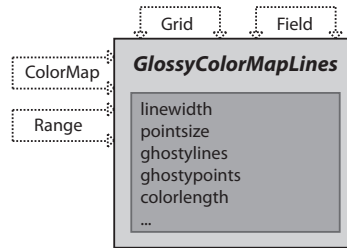Finally, the $4D$ geodesic module has to be registered to *Vish*:

Figure 6.15: Schematic illustration of the *Vish* line rendering module `GlossyColorMapLines`. Primary input is the *Fiber Grid* connection. A scalar field and colormap can be connected to texture the line.

```
1  typedef FieldLines::IntegralLines<metric44, FieldLines::Geodesic>
2  GeodStationary4;
3
4  static Ref<VCreator<GeodStationary4, AcceptList<Fiber::Field>>>
5  MyG44Creator("Compute/Geodesics44", ObjectQuality::EXPERIMENTAL);
```

The *4D* stationary geodesics are verified in *section 7.3* by visualizing a Schwarzschild metric, see *section 2.2*, and used to explore the Kerr metric in *section 7.4*.

6.3 Rendering Lines

For the rendering of lines a *Vish* rendering module was developed. The module is based on the *VBO* technique and makes use of the caching mechanism, as described in *section 5.2.3*.

A basic module for line rendering was already included in the *Vish* environment. I introduced an enhanced module that allows to texture the lines with any color-map driven by an arbitrary scalar *Fiber Field* stored on the line *Fiber Grid*.

The module primarily takes a grid object as input to get the geometry data, the vertices and connectivity, of the lines. Thickness of the rendered lines and size of vertex points can be controlled by the parameters `linewidth` and `pointsize`. Lines and points can be transparent (normal or additive).

If no scalar field is connected to `secondaryfield`, 1D texture coordinates are computed based on the length of the line needed for texturing by a color-map. If no color-map is connected a linear ramp between two user defined colors is created, starting from green and going to red.
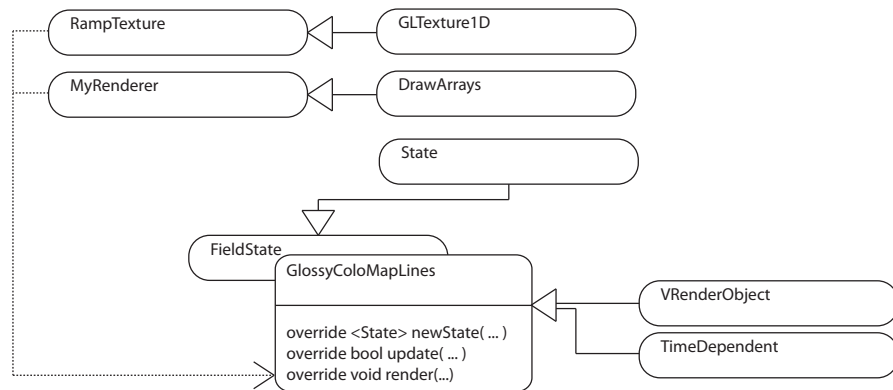
Figure 6.16: Uml diagram of the line rendering module. An inner state class is used that make data, extracted and computed in the `update` function, available in the `render` function. Rendering is done using a class for rendering data arrays (`MyRenderer`) utilizing *VBO*s, see *section 5.2.3*. A one dimensional color ramp is created if no color-map is providing a texture. *Vish* provies a `GLTexture1D` class for convenience. Textures and `VBO`s are cached as described in *section 5.3*.

To color the line by a scalar field and color-ramp, a scalar field and a color-ramp is connected to `secondaryfield` and `colormap`. An additional range object can be connected as well (`range`) to scale and clamp the scalar field range. A toggle between length based and scalar field coloring can be adjusted, too (`colorlength`). *Figure 6.15* illustrates the module connections and parameters.

To enhance the perception of $3D$ curvature, highlighting or shading of the lines was added based on *OpenGL* multi texturing. The highlighting texture is blended over the color texture additively (`glBlendFunc(GL_ONE, GL_ONE)`). The tangential vector of the line is used as a $3D$ texture coordinate which is then transformed into a $1D$ coordinate by using a texture transformation matrix. The matrix basically does a transformation dependent on the view direction, thus simulating a $cos(\phi)$ between the camera axis and the tangential line direction. The look-up in the pre-sampled $1D$ highlighting texture is added to the color of the line resulting in a shaded line. Lines become highlighted when the tangential vector becomes perpendicular to the camera view axis and diffuse when oriented parallel.

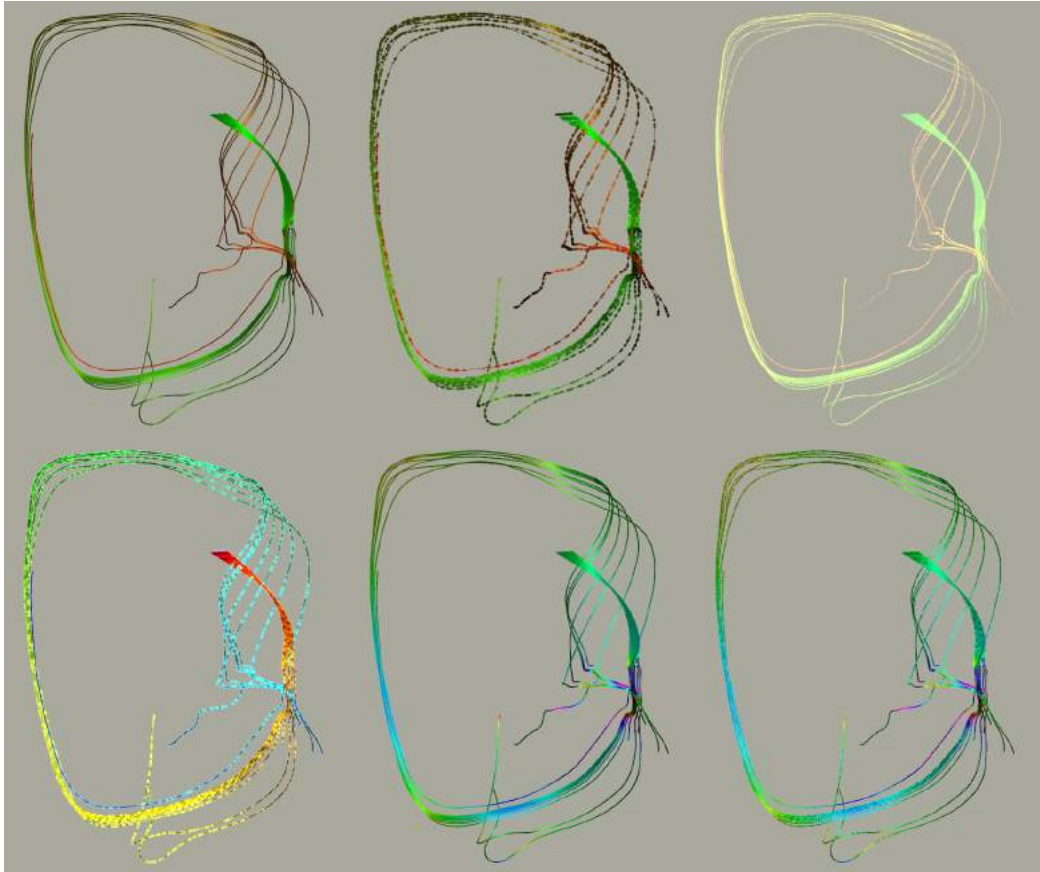*Figure 6.17* illustrates six different configurations of the line rendering module, drawing the same line geometry.

Figure 6.17: Same line geometry rendered with different settings in the line rendering module `GlossyColorMapLines`. *Top Left:* Standard parameters. Glossy lines of width 2. *Top Mid*: Rendering of points of the lines added. Narrower lines (width 1) and bigger points (size 3). *Top Right:* Additive lines and additive points of same width and size. *Bottom Left:* Color-map still driven by length for texturing and big additive points. *Bottom Mid:* Color-map driven by a scalar field, available on the line grid. *Bottom Right:* Like *bottom mid* but with small additive points.

Figure 6.16 shows the class dependencies of the rendering modules. The `update` function extracts the line geometry and computes curvature and the length parameter of the lines per vertex. Also, the scalar data field is extracted and normalized, so that it can be used as 1D texture coordinates for the color map. All this data is stored in the inner state object and only recomputed when the line geometry changes or other user input forces a recompute.

The render function gets all data it needs for instant rendering from the state object. The *OpenGL* calls for rendering the *VBO*s are encapsulated in a renderer class, which in this case was extended in `MyRenderer`. Textures and the *VBO* are cached in *Vish*s `GLCache`. *Listing 6.3* shows some excerpts on how this is done:

```cpp
void GlossColorMapLines::render(VRenderContext&Context) const
{
RefPtr<FieldState> S = getState(Context);
    if(!S) return;

/* ... */

RefPtr<ValueSet> RampValues = new ValueSet();
RefPtr<RampTexture> LineTexture2;

    try{
        LineTexture2 = Context(*S)(this)(RampValues)(TEXTURE());
    }
    catch(const GLError&){}

    if(!LineTexture2)
    {
    rgba_float_t c1, c2;
        ColStart << Context >> c1;
        ColEnd   << Context >> c2;

        LineTexture2 = new RampTexture(1, c1, c2);
        LineTexture2->enlighten();

        Context[*S][this][RampValues](TEXTURE()) = LineTexture2;
    }

/* ... and similar for glossy highlight texture .. */

/* ... get module render parameters and set OpenGL states ... */

GlossyTexture::Render LINERENDER(*LineTexture,
                                 Context.CameraSettings);

```

```
35  /* ... */
36
37  RefPtr<ValueSet> myCacheableValues = new ValueSet();
38  string    VBOKey = "";
39  RefPtr<VBO> myVBO;
40
41      try
42      {
43          myVBO=Context(*S->Vertices)(typeid(*this))
44                       (myCacheableValues)(VERTEXBUFFER(),VBOKey);
45      }
46      catch (...){}
47
48      if (!myVBO || S->reVBO)
49      {
50      RefPtr<MemBase> Pts = S->Vertices->create();
51      RefPtr<VertexArray>
52          VA = GL::FieldBuffer<VertexArray>::create( Pts, true );
53
54          myVBO=Context[*S->Vertices][typeid(*this)]
55                       [myCacheableValues](VERTEXBUFFER(), VBOKey);
56          myVBO->clear();
57          myVBO->append( VA );
58
59          if( S->TexLength && !ColorLen() ) //use length for coloring
60          {
61          const int TextureUnit = 1;
62          RefPtr<TexCoordArray>
63              TCA = GL::FieldBuffer<TypedTexCoordArray<double> >::
64                      create( TextureUnit, S->TexLengthScaled, false );
65
66          myVBO->append( TCA );
67          }
68
69          /* ... similar for color-map and glossy texture ...*/
70
71          RefPtr<MyRenderer::EdgesArray_t> EA = S->Edges->create();
72
73          myVBO->setRenderer( new MyRenderer(
74              ConvertVectorArray<unsigned int, index_t >::convert(EA),
75              GhostL(), GhostP(), point_size, width) );
76      }
77
78      myVBO->call();
79  }
```

First, the state object is retrieved. Then textures are prepared and stored in an `GLCache` associated to the `Context`. If the extraction of the texture from the cache fails, *line 12*, the ramp texture is created based on two color input

attributes of the module, *line 22*, and stored in the cache. It is identified by the state, the rendering module, a value set and a texture identifier, *line 25*. The additive texture used for highlighting (`GlossyTexture`), is stored in the cache similarly but using a different value set.

The texture matrix for the highlighting texture is computed by using the inner class `Render` of `GlossTexture`, providing the texture and the actual camera settings.

Next, the a vertex buffer object containing the vertex data is tried to be extracted from the cache identified by the vertex data pointer, the rendering module, a value set, a vertex buffer identifier and a vertex buffer key, *line 43*. If this fails a new *VBO* has to be created, what is done in *line 54*. The *VBO* is loaded with an `VertexArray` in a *OpenGL* compatible memory layout, in *line 57*. The `VertexArray` is directly created from the *Fiber Field* data array in *line 51*. This is a very efficient operation because of the array layout in the *Fiber Bundle* library.

Next, texture coordinate arrays are created and loaded into the *VBO* object, *line 59* to *line 67*. The connectivity of the lines is retrieved from the state, *line 71*, and the custom line renderer `MyRenderer` is set as a renderer for the *VBO* object passing the connectivity and some rendering parameters. Finally the *VBO* is drawn using the `call` function in *line 78*. Either it was retrieved from the cache or created newly.

Using this caching technique most rendering calls retrieve the data out of the cache. Parameter changes like point size or line width do not trigger a creation of a *VBO* object because it is not necessary. Also, *VBO*s are reused when a certain time step is revisited. When playing an animation in *Vish* only the first run will require the creation of *VBO* objects. In the second run of the same time interval these are all retrieved from the `GLCache`, with all data cached in the graphics card memory.

Instead of using the texture matrix and texture blending the shading of the lines could also be achieved using modern shading technology. In fact there are lots of other possibilities to further enhance the rendering of lines using *OpenGL* shaders.

Enhancements would include rendering transparent lines. Mapping a certain scalar data field to transparency could be a good tool to visualize certain characteristics of data. Rendering transparent lines would involve sorting the lines when drawing, which is a non trivial problem since it has to be done accurately enough and must be very efficient.

A nice extension would be to thicken the lines using a cylindric shape, like extruding a circle along the curve. It would be interesting to connect the thickness to some data. Scaling and squeezing could enhance visual perception of data. When applying shape extrusion one could also blend

between different shapes, like between a rectangular or star shape and a circle dependent on input data.

Chapter 7

Applications

This chapter demonstrates the use of the implemented *Vish* modules described above in different application domains.

The first application shows the computation of streamlines in an analytic vector field sampled on an uniform grid. Here, accuracy and time measurements verify the implemented differential equation solvers of streamlines.

The second application stems from an engineering application. It demonstrates the that the streamline algorithm was formulated independent on the underlying discretization grid structure and shows visualization features as texturing the lines and seeding using different kinds of *Fiber Grid* objects. A dataset containing vector and scalar field data describing the flow of a two fluids in a stir tank are visualized.

The third application shows the computation of geodesics in the analytical Schwarzschild metric sampled on a uniform grid. It is used to verify the computation algorithms and introduces visualization techniques for the coordinate-acceleration. The Schwarzschild metric then is explored in the xy-plane ($2D$).

The fourth application is an extension of the third and demonstrating computation geodesics in the analytic Kerr metric sampled on a uniform grid. It is explored in all three spatial dimensions and illustrates four dimensional coordinate-acceleration.

The fifth application utilizes spatial geodesics in a medical context. Diffusion tensor data is visualized stemming from a MRI scan of a brain. It demonstrates the high re-usability of the implemented algorithms.

Besides the illustrations showing the visualizations of the different applications a schematic *Vish* network is shown for each application illustrating the main modules in action, demonstrating flexibility and modularity of the *Vish* approach.

7.1 Visualizing Flow of CouetteFlow

The first application is the visualization of a vector field by streamlines of a an analytically described field.

Since the integral line modules only apply to numerically provided data the analytic vector field is sampled on a uniform grid in before. A three dimensional uniform grid is used with a resolution of $32 \times 32 \times 32$.

The vector field is given by the following equation, with $r^2 = x^2 + y^2$ and $v = 0$ if $r^2 < 1$ or $r^2 > 4$. The factors s and t are scaling factors. The field is sampled around the coordinate origin in the interval $(-2.1; -2.1; -2.1)$ to $(2.1; 2.1; 2.1)$.

$$v = \begin{pmatrix} -\frac{r^2-2}{r^2}s & y \\ \frac{r^2-2}{r^2}s & x \\ 5t & z \end{pmatrix} \tag{7.1}$$

The vector field is a modified Couette flow. In fluid dynamics the Couette flow describes the velocity of a viscous fluid between the gap of two opposed rotating cylinders. The vector field of the two dimensional Couette flow was extended by a linear variation in the z-axis. This also is scaled by a parameter dependent on time.

The analytic vector field is defined in a tube like shape, with z being the axis of the tube. The characteristics of the vector field in the $z = 0$ plane are illustrated on the left hand side of *figure 7.1*. The area where some velocity is defined can be clearly seen, with the radius r having a value between 1.0 and 2.0. The vector magnitude is changing with the radius, having a high value at the outer radius decreasing to zero and growing again by moving to the inner radius. The direction of the vector changes from counter clockwise orientation to clockwise orientation referring to the z-axis.

When looking at the $y = 0$ slice in *figure 7.1*, on the right hand side, the vector dependency on the z-coordinate is illustrated. Vectors are oriented in $z = 0$ plane when $z = 0$. If $z > 0$ the vectors are increasingly uplifted and if $z < 0$ they are pushed down, according to *equation (7.1)*.

The following figure tries to visualize these properties of the vector field by using just one seeding geometry.
Figure 7.2 is seeded by a point forming a uniform grid and slicing by an angle of 45 degree through the vector field volume. (Recognizing the properties is much easier when realtime 3D camera navigation is possible.) The figure shows the streamlines from the side and from the top. Magnitudes and orientations can be seen.

Figure 7.1 and *figure 7.2* are created using the same *Vish* module network, which is illustrated in *figure 7.3*. It consists of four main parts. Modules to
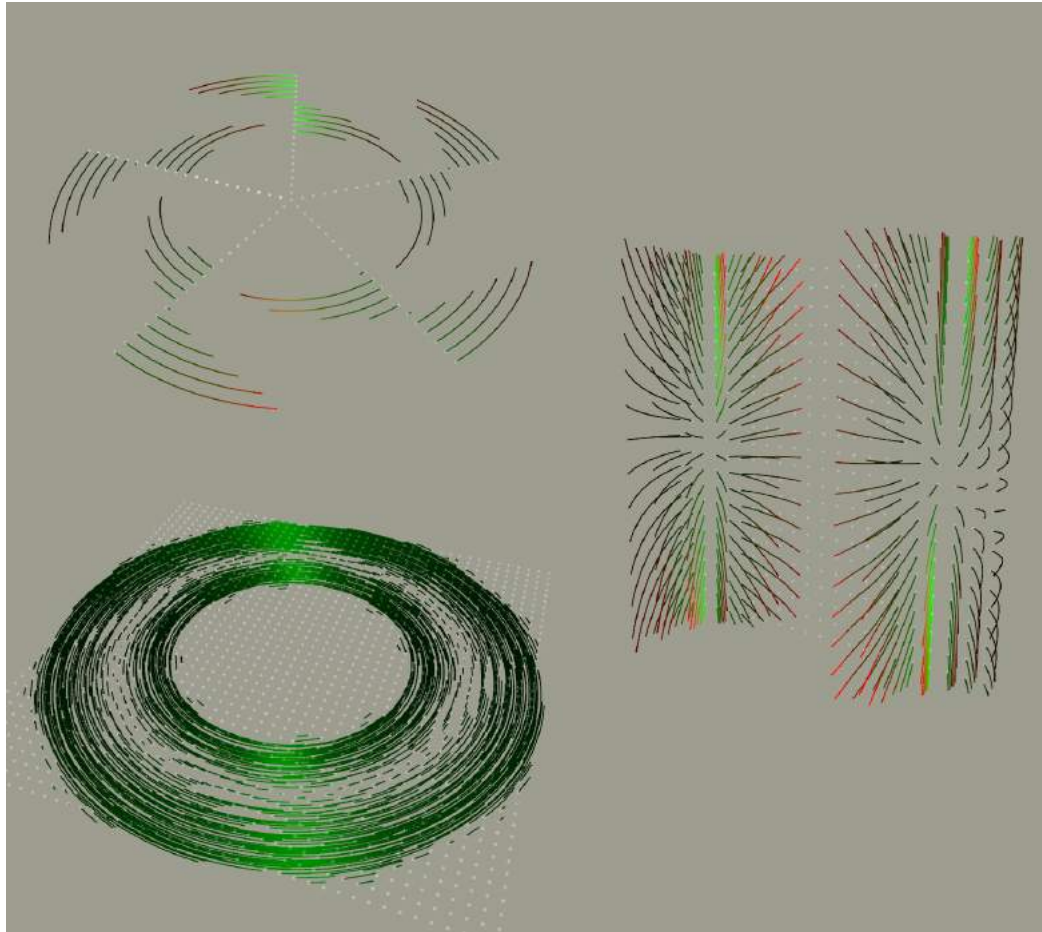
Figure 7.1: Couette flow visualized using streamlines. *Top Left:* Seed points on the $z = 0$ plane form a starry shape. The magnitude and direction of the vector field dependent of the radius can clearly be seen. *Bottom Right:* Seed point on the $z = 0$ plane forming a rectangular grid. The area where the velocity is defined and the magnitude of the vectors are illustrated. Small magnitudes occur where streamlines get very short, see the center ring of the disc. *Right:* Seed points on the $y = 0$ plane forming a uniform grid. The greater the distance to $z = 0$ the more the vectors become lifted up or pushed down.
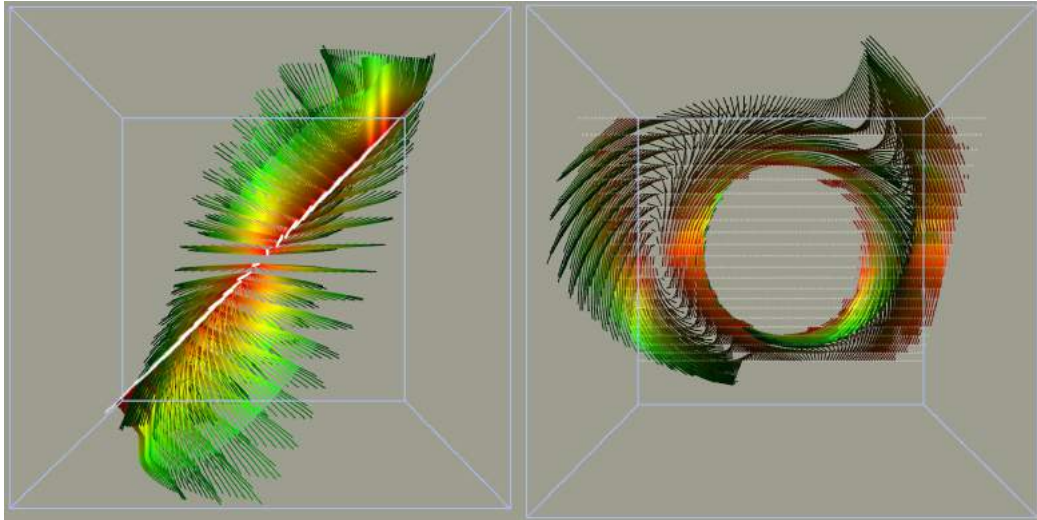
Figure 7.2: Couette flow visualized using streamlines seeded a regular grid rotated by 45 degree.

extract the vector field of the data bundle (cyan), a module to create a seeding grid (grey), the streamline integration module (green) and the line rendering module (yellow). To visualize a different data set the top node that delivers the *Fiber Bundle*, the `CouetteFlowDataCreator`, has to be exchanged.

The `GeomPointDistribution` may be replaced by another seeding module or even a whole subnetwork of nodes to create the seeding geometry. *Figure 7.12* shows a more complex network.

Figure 7.4 illustrates the contrary orientations and uplifted and pushed down vectors very clearly. But it does not show the definition space of the vector field. Here, four grid objects created by point distribution modules where combined into one seeding grid. The circular shapes are shifted a bit along the z-axis as that the streamlines escape the $z = 0$ plane.

The Couette flow vector field is a good test scenario to verify the implemented streamline algorithms. Especially, a test comparing accuracy and performance of the explicit Euler and the adaptive Runge Kutta of 8th order yielded expected results.

To compare the integration methods I seeded one streamline at the position $(1.0; 0.0; 0.0)$. Since the point is located in the $z = 0$ plane it should not escape this plane. Also the vector field flows perfectly circular around the center, the streamline should have the shape of a circle. First, I did an integration using the Dop853 and adjusted the line length, or number of integration steps, such that the streamline forms a circle. The endpoint
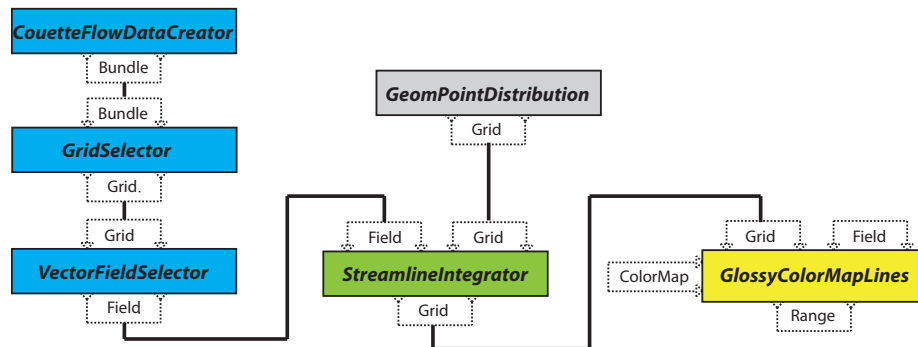
Figure 7.3: Schematic *Vish* network used to create the visualizations shown in *figure 7.1* and *figure 7.2*. The vector field data is extracted from the module that creates the bundle, by first selecting a grid object and then a vector field (cyan). The seeding geometry is created using a `GeometricPointDistribution` module (grey). The `StreamlineIntegrator` does the computation and stores a grid of lines back into the bundle hosting the vector field and provides a grid handle for the rendering module `GlossyColorMapLines`.

almost coincides with the start point, see top left of *figure 7.5*. There is no noticeable gab between the starting line (green) and the ending line (red).

Then, I switched to Euler integration and choose a step size such that the same number of steps would result in a similar circle as created by the Dop853 integration, see mid top of *figure 7.5*. Then I decreased the step size by half and then quarters until a visually similar circle was created. All information regarding step number step sizes and integration times are gathered in the following table[1]:

---

[1]Measurements were done in SVN Revision 1724 of *Vish* on a dell M1330 XPS latop with 4GB RAM, Intel Core2 Duo T8300 @ 2.4Ghz and NVidia Geforce 8400M GS graphics card. Compiled in Windows Vista 32 using gcc 3.4.2 (mingw special) in debug mode.
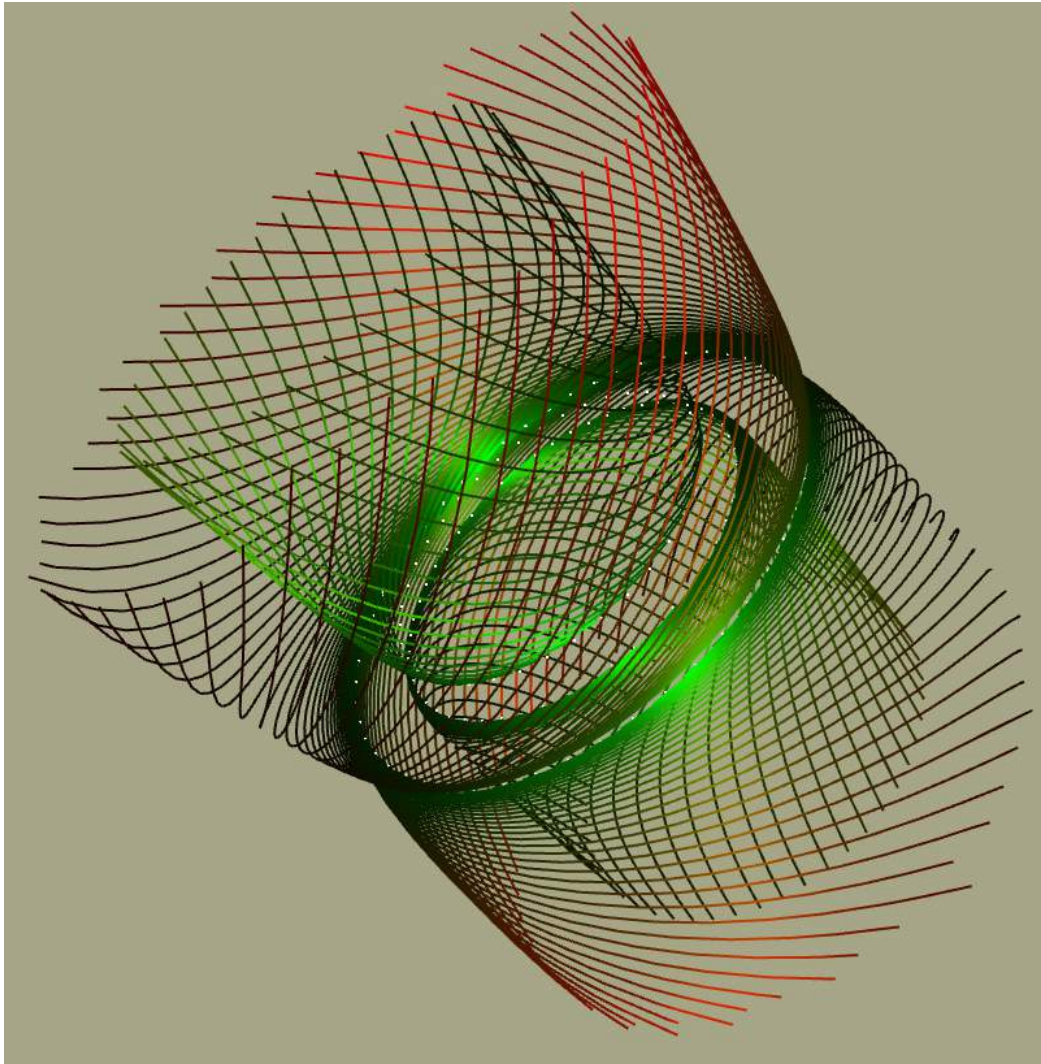
Figure 7.4: Couette flow visualized using streamlines seeded by four point distribution modules combined by three grid adder modules. The streamlines of the inner circle flow in different circular direction compare to the outer streamlines.

Integration	Steps	Stepsize	Overall Time [sec]	Time per Step [sec]
Dop853 1	154	-	0.406	0.0026
Euler 1	154	1.5	0.031	0.0002
Euler 2	92	0.75	0.015	0.0002
Euler 3	164	0.1875	0.031	0.0002
Euler 4	708	0.0469	0.14	0.0002
Euler 5	3090	0.01172	0.639	0.0002
Euler 6	12770	0.00293	2.278	0.0002
Dop853 2	474	-	1.139	0.0024
Euler 7	11350	0.01172	1.997	0.0002

When comparing the speed of *Euler 1* and *Dop853 1* the *Euler 1* outperforms the *Dop853 1* by a factor of 13 but with an awful accuracy, as illustrated. When decreasing the step size of Euler to get one good circle, see bottom right of *figure 7.5*, now, the *Dop853 1* outperforms the *Euler 6* by a factor of 5.6.

Another comparison was based on *Euler 5* and *Dop853 1*. They have a similar integration time. Based on the step size the length was now adjusted such that the streamline would flow around the axis for 4 times: *Dop853 2* and *Euler 7* in the table. The streamlines illustrated in *figure 7.6*. The bottom left figure demonstrates how the Dop853 streamline still stays accurately on the circle, while the Euler starts to spin outwards.

Thus, the Euler integration can be fast if accuracy is not important. The Dop853 still provides good speed at high accuracy, as expected, *section 6.2.1*.

Figure 7.5: Comparison of the integration methods Euler and Dop853. A correct integration should yield a circle, like, when using Dop853 (top left). An Euler integration with the same number of steps results in a spiral shape (top mid). Decreasing the step size of the Euler integration finally yields a circle but with a higher computation time than Dop853 (top right via bottom left to bottom right). Data according to the illustration is gather in the table.

E
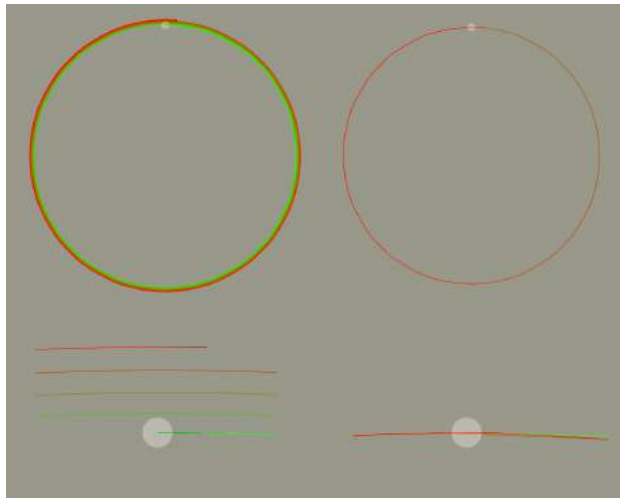
Figure 7.6: Comparison of *Euler 6*, left, and *Dop853 2*, right, which have a similar computation time. The streamline circles 4 times around the center. However, accuracy is very different. A zoom to the start point, bottom figures, illustrate how the *Euler 6* spins outwards whereas the *Dop853 2* stays accurately on the circle.

7.2 Visualizing Flow and Pressure in a Stirred Fluid Tank

An application stemming from the Mechanical Engineering Department of the Louisiana State University is the visualization of streamlines in a stirred tank. The dataset was provided by Sumanta Acharya and Somnath Roy.

They computed the mixing of two distinct fluids in a stirred tank, which is frequently used in chemical industries. The aim is the analysis of the mixing behavior. If the mixing properties of the tank could be improved this would be very interesting for chemical industries.

To capture the geometry of the stirred tank sophisticated methods were used, making visualization of the results difficult or impossible with available standard software, such as Tecplot [56] or EnSight [37]. 2088 curvilinear multi blocks were used to discretize the computational domain by over 3 million cells. Over 5000 time steps were produced during the simulation.

A rotating four bladed propeller induces the mixing in the cylindrical tank that has vertical baffles mounted along its walls. *Figure 7.7* illustrates the geometry of the tank.

As a result of the large eddy simulation with "Immersed Boundary Meth-
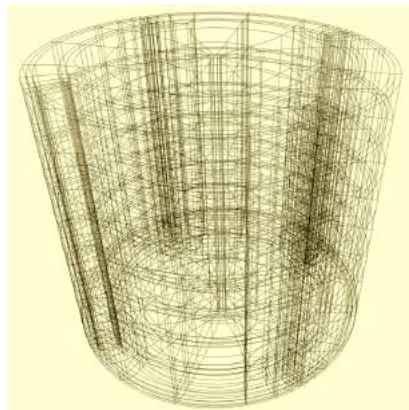
Figure 7.7: Stirred tank discretized by 2088 curvilinear blocks. The geometry of the four vertical baffles as well as the rotating axis and the curvature of the hemispherical bottom of the tank are modeled.

ods" the finite volume computation yielded a vector field describing the flow of the fluids and a scalar field for pressure. Data was converted to $F5^2$ format and could then be loaded via the *Fiber Bundle* library into *Vish*.

I used the data set to apply, test and develop my integration line concepts for streamline seeding, computation and visualization and, especially, to make the integration independent of the underlying grid data structure, see `FindLocalFromWorldPoint` in *section 5.4*. The same algorithm can be used for uniform and curvilinear grids. Other grid types are supported by extending the `FindLocalFromWorldPoint` class. As soon as, for example AMR support is added to the local point finder, the streamline module will also work in AMR data.

The results of the collaborative work were shown at the CCT booth on the Supercomputing conference in 2008, Austin USA, and finally yielded a communication paper, that I presented at WSCG 2009 [53], in Plzen Czech Republic, see [1] also included in *appendix C*. This formed a basis for further work mainly done by Bidur Bohara and led to a second publication (journal) at WSCG 2010, see [16], also included in *appendix C*.

The figures shown in this chapter illustrate different seeding strategies, compare the integration methods and demonstrate the benefit of using the *Fiber Bundle* model as a systematic data model. In combination with the module separation approach, utilized in the *Vish* network, visualization features were created that were not planned on purpose, but rather happened

---

[2]File mapping of the *Fiber Bundle* data model utilizing the HDF5 format, *chapter 4*.
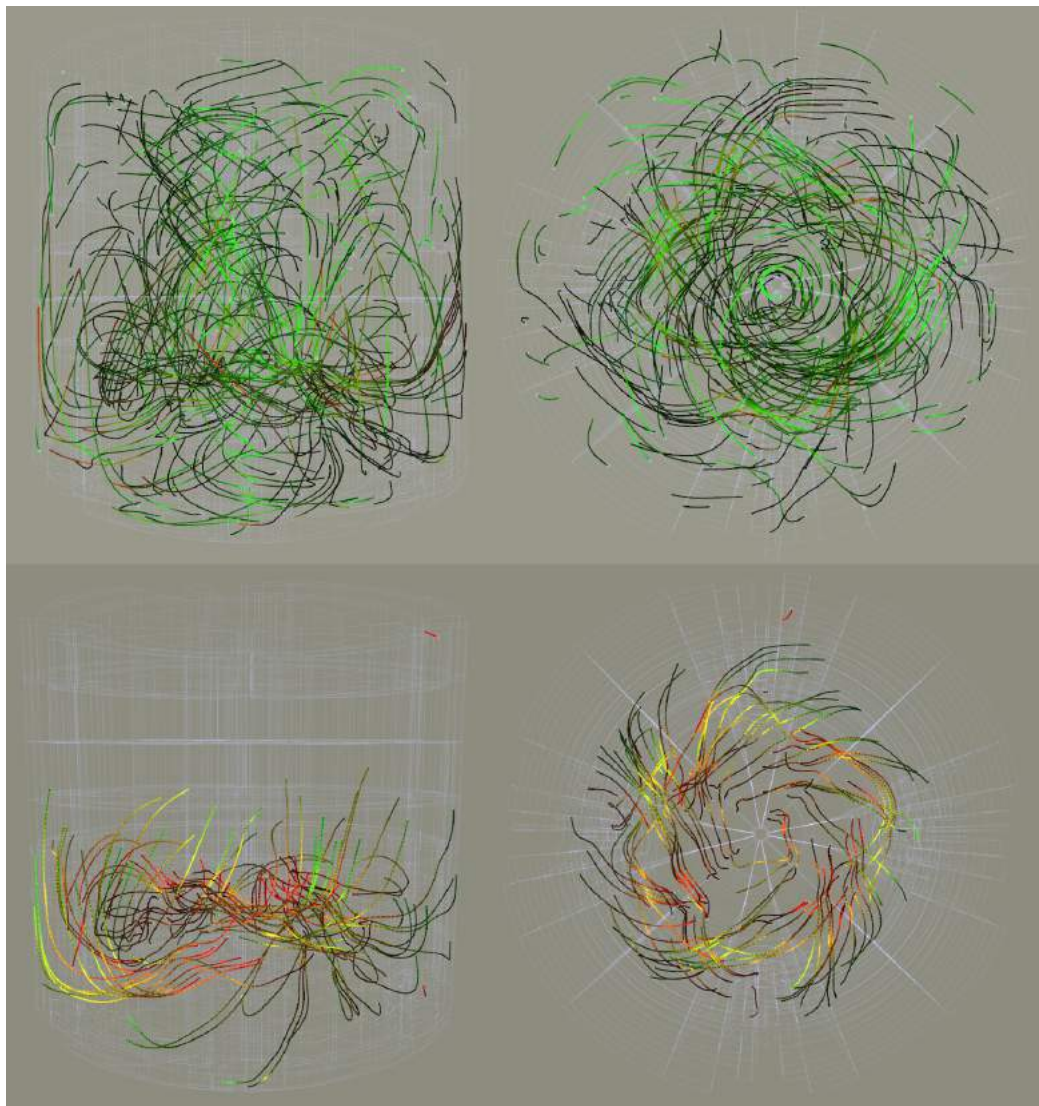
Figure 7.8: Euler integrated streamlines in the stirred tank dataset. 50 steps are computed for each streamline. Thus, the length of the line indicates the velocity of the fluid. *Top:* Streamlines were seeded using the a random point distribution module. Though, the many "spaghetti" give no clearly structured visualization several properties are illustrated over the whole volume. The flow is moving clockwise around a mid axis, see right, and moving downwards close to the center and upwards at the tank boundaries.*Bottom:* Random seed points were clamped by a value of the pressure field. The images show streamlines starting at high pressure in the lower part of the tank flowing outwards to the borders. The rotating propeller of the stirred tank causes the high pressure in the fluid, what is also shown in *figure 7.13*.

"accidentally" during implementation.

Figure 7.8 gives an overall impression of the motion of the fluid by seeding randomly in the stirred tank volume. The scalar pressure field is used to extract regions of high pressure, see *section 6.1.1*.

Some time measurements were done to analyze performance behaviors of the implemented algorithms. *Figure 7.9* shows the setup that was used in the stirred tank. Here, streamlines are seeded along a horizontal line on 12 seed points close to the center of the tank.

Time was measured during the first integration process, where all *Uni-GridMapper*s, used to speed up computation of local grid coordinates, have to be created for each visited block, *section 5.4.3*. In a repeated computation previously computed *UniGridMapper*s are reused, leading to a speed up in the repeated computation of the streamline.

Figure 7.9 shows only those curvilinear multi-block outlines a *Uni-GridMapper* object has been created for.

Timings of Euler and Dop853 streamlines with a length of 50 and 500 steps were stopped and gathered in the following table[3]:

Integration	Steps per Line	Steps	Overall Time [sec]	Time per Step [sec]	Speedup
E50f	50	612	10.84	0.0177	
E50c	50	612	0.80	0.0013	13.55
E500f	500	5343	39.78	0.0074	
E500c	500	5343	6.506	0.0012	6.11
D50f	50	612	19.02	0.0307	
D50c	50	612	10.70	0.0170	1.78
D500f	500	5147	131.10	0.025	
D500c	500	5147	93.86	0.018	1.4

Following naming convention is used for the measurements: E stands for Euler and D for Dop853. This is followed by the number of integration steps, which is followed by f, denoting a first computation or c, denoting computation with all `Unigridmappers` of the visited blocks having been cached. The speed up always refers to the first and cached computation times.

When comparing the timings of the Euler integration with respect to the *UniGridMapper* initialization the speed up is about a factor of 10 (E50f/c and E500f/c in the table). The difference (half) of the speed up comparing the 50 and 500 steps lines could be the consequence of the increasing distance

---

[3]Measurements were done in SVN Revision 1724 of *Vish* on a dell M1330 XPS latop with 4GB RAM, Intel Core2 Duo T8300 @ 2.4Ghz and NVidia Geforce 8400M GS graphics card. Compiled in Windows Vista 32 using gcc 3.4.2 (mingw special) in debug mode.

between integration lines. All lines start closely together. Thus, a point of a streamline will probably be located in the same multi-block as a point of a different streamline that was computed before. The *UniGridMapper* can be reused in that case. However, with increasing integration the streamlines move apart and such a reuse becomes more and more unlikely, resulting in a lower speedup for long integration lines.

Comparing the same timings of the Dop853 integration the speed ups are much smaller (D50f/c and D500f/c in the table). During the computation many more local coordinates are computed densely on a streamline. Thus, inside a multi-block is the *UniGridMapper* is already reused several times. The creation process itself has a much smaller influence on the overall time. Also when computing long lines a decrease of the speed up occurs as perceived with the Euler integration.

When comparing Euler and Dop853 timings a similar factor of more than 10 can be observed when looking at E50c/D50c and E500c/D500c. This is a similar factor as observed in the Couetteflow table, *section 7.1*.

The initialization process has a much deeper impact on Euler. Thus, the Dop853 requires just about double time in that case, compare E50f/D50f and E500f/D500f.

Irritating is the difference in the total number of steps in the 500 steps lines: 5343 in case of Euler and 5147 in case of Dop853. If a point of a streamline gets out of the spatial domain line computation will stop before the final step number is reached. This can happen in case of Euler integration since the step size might be too long. However, it is unlikely to happen using the highly accurate Dop853 solver.

This could be related to a problem of finding the local coordinates at block boundaries. When a world point is located closely at a quite deformed boundary of a block a point can probably neither be found in one block or the other.

Similar can occur between cells inside the multi-block. But being inside the cell neighbors are known and used to overcome this numerical unaccuracy without using epsilons, see *listing 5.22* in *section 5.4*.

Similar can be done when neighborhood information of the blocks is available. Neighborhood information can easily be stored in the *Fiber Bundle* model as an additional *Fiber Representation*. Best, a file converter would already copy or compute this information into the F5 format.

Further investigation should be done here, since after revisiting the `LocalPointFinder` during the work on pathlines with Bidur Bohara and Nathan Brener (Computer Science Departments of LSU) this issue was thought to be tackled down completely, see [16] also included in *appendix C*.
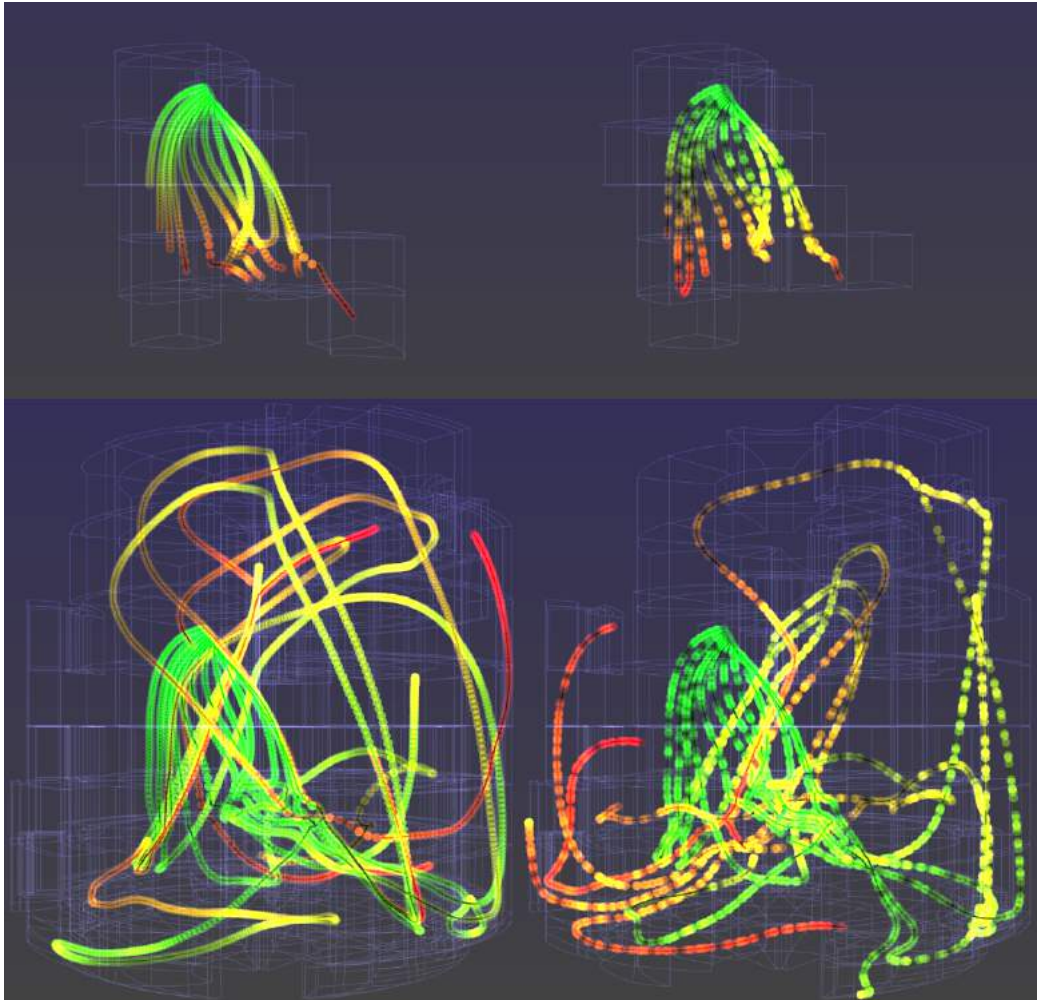
Figure 7.9: Streamlines seeded by a line of 10 seed points. Different parameters were used to compare the performance gain by caching the `Unigridmapper` objects. Multi-blocks touched by the streamlines are shown by their outlines. *Top Left:* 10 Euler integrated streamlines, 50 steps. *Top Right:* 10 Dop853 integrated streamlines, 50 steps. Due to the short length they are very similar to the Euler integrated lines. *Bottom Left:* 10 Euler integrated streamlines, 500 steps. *Bottom Right:* 10 Dop853 integrated streamlines, 500 steps. The streamlines differ a lot from the Euler integrated lines. Because of the long length the difference in the integration is bigger. Still the Euler integrated lines give a very good approximation. Data of measurements is collected in the table.
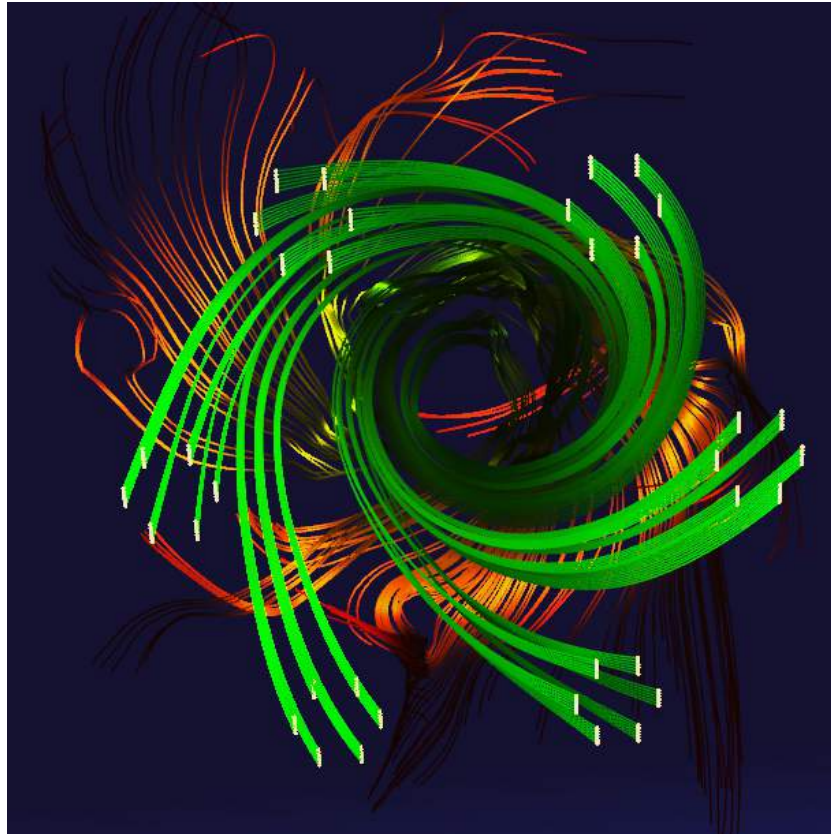
Figure 7.10: Streamlines seeded by a grid created by two convolution processes. A line is convolved with a smaller circle. The resulting grid is again convoled with a bigger circle. The stirred tank is shown from top viewing in negative z-direction. Streamlines flow downwards while spinning around the axis. The streamline ribbons remain ribbons in a quite steady circular flow until they reach a turbulent region where they are torn apart (region of the propeller).

Figure 7.10 illustrates how a multiple convolution of seeding grids helps to analyze the fluid flow. Ribbons of streamlines are created by seeding densely along short lines. These lines are grouped in small circles and finally copied around the axis of the tank to cover a bigger region.

The streamlines seeded by this geometry visualize the flow locally by the ribbons as well as in wider regions. Also, the many lines do not produce hardly readable "spaghetti" as seen in *figure 7.8*. The strong geometrical symmetry is a result of the convolution of circles and lines. The symmetry enhances the visual result.

As described in the figure the ribbons get torn apart in the lower region of the stirred tank. This is also illustrated in *figure 7.11*. Here, tube-like streamline are seeded at a similar location as in *figure 7.10*. Additionally, the streamlines are colored by the value of the magnitude of the velocity field. The labeled color-map, *section 5.5*, illustrates the magnitude values. The blue to magenta colors represent a high magnitude value on the streamline.

Using the *Fiber Bundle* and *Vish* module approach any valid grid object can be used to seed streamlines. First, only the seed point grids described in *section 6.1.2* were used. During Werner Benger's work on an iso-surface module suitable for curvilinear grids, we realized that the iso-surface is a well defined *Fiber Grid* object and, thus, valid for seeding streamlines.

One can extract features of the fluid simulation by an iso-surface, for example, of the pressure or Laplacian field, and can seed a streamline from the surface. *Figure 7.13* illustrates how a pressure iso-surface revealing the propeller was used to seed the streamlines computation. More examples and details can be found in [1] also included in *appendix C*.
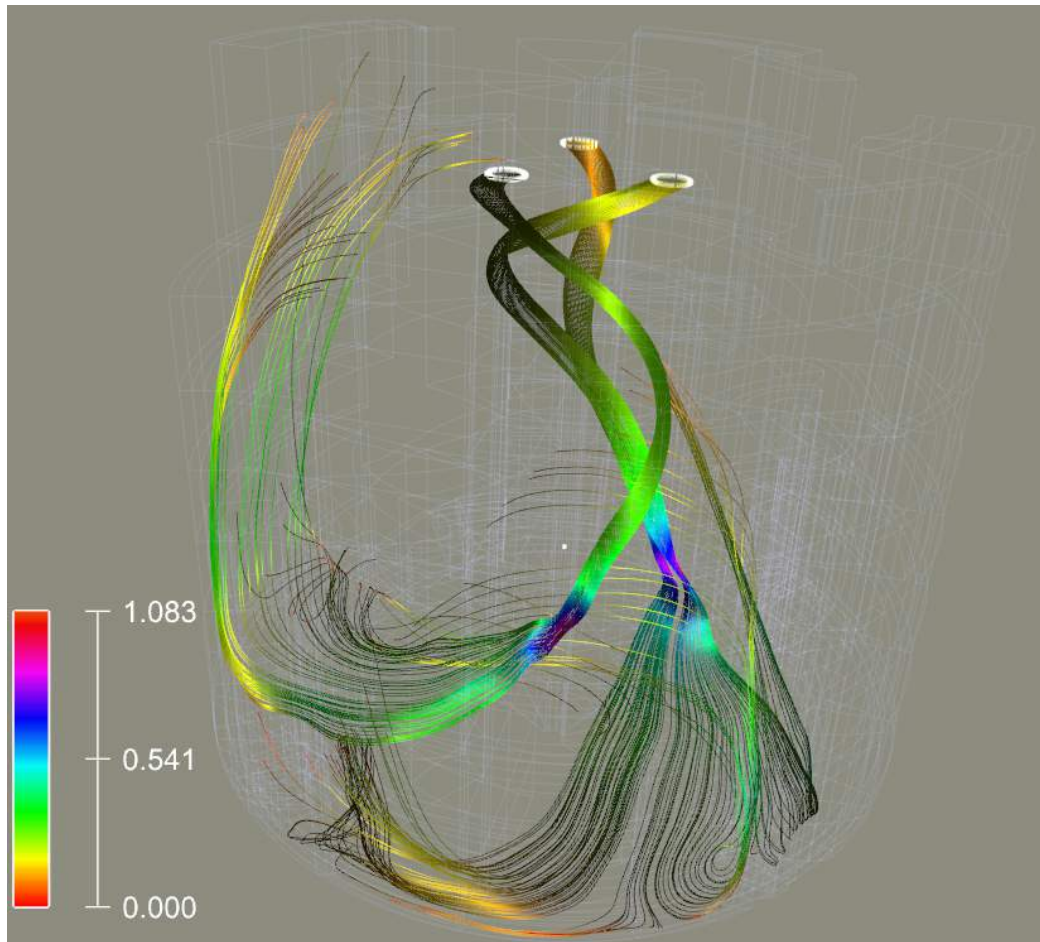
Figure 7.11: Tube-like streamlines seeded in the upper region around the axis of the stirred tank. This illustration brings together several of the developed components: Streamline computation in a curvilinear vector field, illuminated line rendering textured by a color-map which is driven by a scalar field on the line grid, showing touched multi-blocks and displaying a HUD for labeling the color map. Similar to *figure 7.10* steady and more turbulent regions can be identified. Turbulent regions are related to high magnitudes as illustrated by the blue and magenta color on the stream lines tearing apart the stream-tubes.
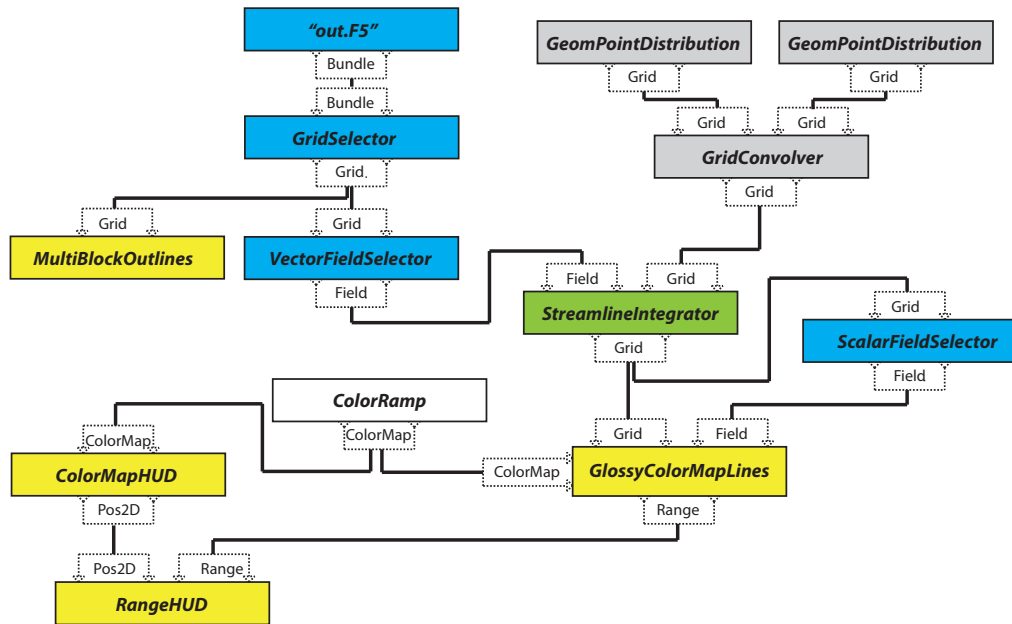
Figure 7.12: Schematic *Vish* network of the visualization shown in *figure 7.11*. It consists basically of the same components as illustrated in *figure 7.3*. Seeding geometry creation is done by convolving two grid objects (grey). Additional rendering modules enhance the pure streamline visualization (yellow). The visualization of touched multi-blocks of the curvilinear data grid is enabled by connecting the `MultiBlockOutlines` module. A color-map `ColorRamp` is used to texture the rendered lines. Thus, the scalar field of the computed vector magnitude has to be extracted from the line grid by using a `ScalarFieldSelector` which is then connected to the field input of `GlossyColorMapLines`. The HUD display labeling the color-map is created by connecting the two HUD rendering modules to the color-map and the data range.
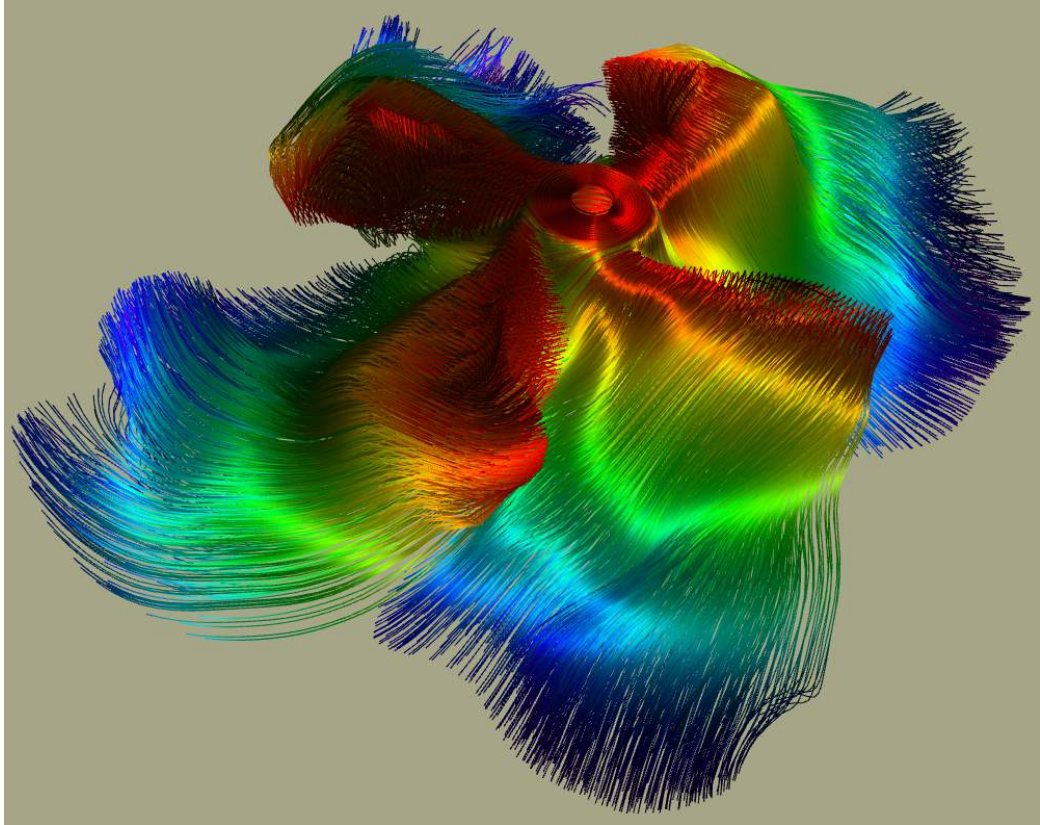
Figure 7.13: Streamlines seeded by an iso-surface created from a scalar pressure field. The iso-surface is stored as a *Fiber Grid* object in the *Fiber Bundle* and is used as the seeding grid for the streamline computation module. This was not a planned feature for streamline visualization. It is, finally, the reward of using highly reusable data models and algorithms.

7.3 Visualizing Geodesics in a sampled Schwarzschild Metric

The aim of the visualization of the Schwarzschild metric was to look at a well known simple metric and to reduce the computational complexity for the purpose of error checking and development.

First, the metric itself had to be computed. So I implemented an analytic creator module that samples the Schwarzschild metric on a uniform grid, which could be used as an numerical input data field for the geodesic computation module. First I tried to keep things simple and computed the metric in the xy-plane only, reducing it to $2D$ spatial coordinates. I thought I could cut off the time coordinate as well and do a purely spatial geodesic computation using the geodesic module in a `metric33` field, *section 6.2*.

So I reduced the metric tensor to

$$g = \left\{ \begin{array}{ccc} \left(\frac{1-2m}{r}\right)^{-1} & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 sin^2\theta \end{array} \right\} = \left\{ \begin{array}{ccc} \left(\frac{1-2m}{r}\right)^{-1} & 0 & 0 \\ 0 & r^2 & 0 \\ 0 & 0 & r^2 \end{array} \right\} \tag{7.2}$$

with $\theta = \frac{\pi}{2}$.

This metric was transformed into Cartesian coordinates by applying *equation (2.15)* and stored on each point on a uniform grid object.

However, in some regions these pure spatial geodesics looked pretty well, whereas in other areas they were distracted strongly from the gravitational center and some geodesics even got bent out of the xy-plane. Also, the result changed with different sample resolutions of the grid. While searching for the error in the algorithm I found two errors regarding some indexing and a missing normalization of the derivatives of the metric tensor used to compute the Christoffel symbols. Since derivatives were computed in array index space they had to be divided by the according cell's size.

The computed geodesics now resided on the xy-plane and did not, or just barely, change with different grid resolutions. But the distraction still occurred.

I introduced a module (`ComputeQddot44`) that computes the coordinate acceleration vector $\ddot{q}$ on any given input vector field to visually explore the distraction of the geodesics. It takes the metric field and a $3D$ vector field as input for initial directions $\dot{q}$. It computes the Christffel symbols and $\ddot{q}$ using the same template functions as the geodesics and outputs $\ddot{q}$ as a $4D$ vector field, see *figure 7.14*.

I then created a new module that extracts the $3D$ spatial coordinates of a $4D$ vector field: `Vector4ToVector3AndScalar`. Now, all *Vish* modules for rendering $3D$ vectors can be used to illustrate the spatial part of $\ddot{q}$.
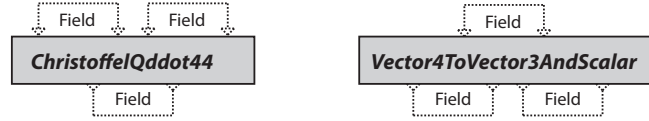
Figure 7.14: Two *Vish* modules that are used to compute $\ddot{q}$ in metric field taking the field and an arbitrary $3D$ vector field as output. Via the `Vector4ToVector3AndScalar` the spatial coordinates of a $4D$ vector can be extracted and handed to a common rendering module.

Then I started checking various parts of the algorithm by making comparisons to hand-made calculations using *OpenOffice spread-sheets*. I started to debug the computation of $\ddot{q}$ at point $(-1, 0, 0)$ with initial direction towards the center of mass $(1, 0, 0)$. At this point g_{rr} should be comparable to the transformed g_{xx} and finally $\ddot{q}$, since the other components compute to 0:

$$\ddot{q} = \begin{pmatrix} \ddot{q}_r \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \ddot{q}_x \\ 0 \\ 0 \end{pmatrix} \tag{7.3}$$

After I could relate these values and had done many expansions of Christoffel symbols on paper and summing ups in *spread-sheets* I switched to another point and initial direction: point $(0, 1, 0)$ and initial direction $(1, 0, 0)$. Again, trying to relate results, $\ddot{q}_r$ and $\ddot{q}_y$, still avoiding to do the tedious coordinate transformations by hand. While doing again expansions of necessary terms of sums and Christoffel symbols I realized that the time coordinate of the metric could not be neglected, since it had an influence on the spatial coordinates.

So, after a good week of debugging I added the time coordinate to the metric and switched to a geodesics computation in a `metric44` tensor field, initially by adding the corresponding module that samples the $4D$ metric on a uniform grid. Again I started to visually explore $\dot{q}$. Since the computation algorithms were all developed as template classes and functions, only little work had to be done for to enable full $4D$ computation. After correcting another error regarding a wrong sign the results looked very well.

Then I verified the pure numerical geodesic by comparing them to a Schwarzschild geodesic computed using the analytic field in polar coordinates. This geodesic computes the Christoffel symbols and $\ddot{q}$ analytically in polar coordinates, see *equation (2.70)*. Both geodesics use the `Dop853` to solve the differential equation.

Figure 7.15 illustrates this verification of the numerical Cartesian geodesics. Four different parameter settings are shown. The geodesics are seeded at point $(1, 0, 0)$ in direction $(0, 1, 0)$. The numerical geodesic is shown in green to red (thick line) and the analytic Schwarzschild geodesic in light blue (thin line). Two different grid resolutions and two different values for the mass of the black hole are shown.

With $m = 0.31$ (upper figures) the numerical geodesic matches the analytic perfectly even in the lower grid resolution. When slightly increasing the mass to $m = 0.3334$ the numerical geodesic diverges from the analytic geodesic after about a quarter of the circular motion. Increasing the grid resolution helps to converge back to the analytic solution and it matches almost the full circle, as shown in the lower right figure of *figure 7.15*.

The numerical geodesic breaks close to the event horizon of the Schwarzschild metric, which is illustrated as a light blue thick circle. This is perfectly expected behavior. Here, the coordinates chosen for the numerical representation of the metric have a singularity and time and space coordinates start to switch. The analytic geodesic is computed in polar coordinates, which has no singularity at the event horizon and, thus, spirals towards the center of gravity plunging through the event horizon.

The geodesics almost form the so-called the photon orbit of the black hole. This orbit is found at a distance where geodesics are caught in a perfect circular trajectory around the heavy mass.

When debugging for the errors, I found looking at the $\ddot{q}$ vector to be a good method to analyze the metric itself and also properties of the mapping Christoffel symbols. I used two different approaches to visualize a vector field of $\ddot{q}$ vectors, which are illustrated in *figure 7.16*. The vector $\dot{q}$ vector field was created using two geometric point distribution modules and a grid subtraction. The upper left figure illustrates two vector fields: $\dot{q}$ as blue arrows and $\ddot{q}$ as yellow arrows. Each starting point of a vector illustrates a photon moving in the direction of the blue arrow, coordinate-accelerated being in the direction of the yellow arrow. The same $\ddot{q}$ field is shown in the lower left figure using vector speckles.

Vector speckles are elliptic Gaussian transparent volumes that show directions of vectors, as introduced in [13]. In contrast to the arrow technique they do not visualize the magnitude of the vector as clearly. Arrows become very hard to read especially when there are drawn many of them, especially in 3D. Vector speckles give a better visual impression. Perception of the direction is enhanced by coloring which is inspired from the physical red and blue shift. If a vector is pointing away from the observer the according speckle is colored in red. It is colored in blue when moving towards the observer and white when getting parallel to the camera image plane. It also fades to white
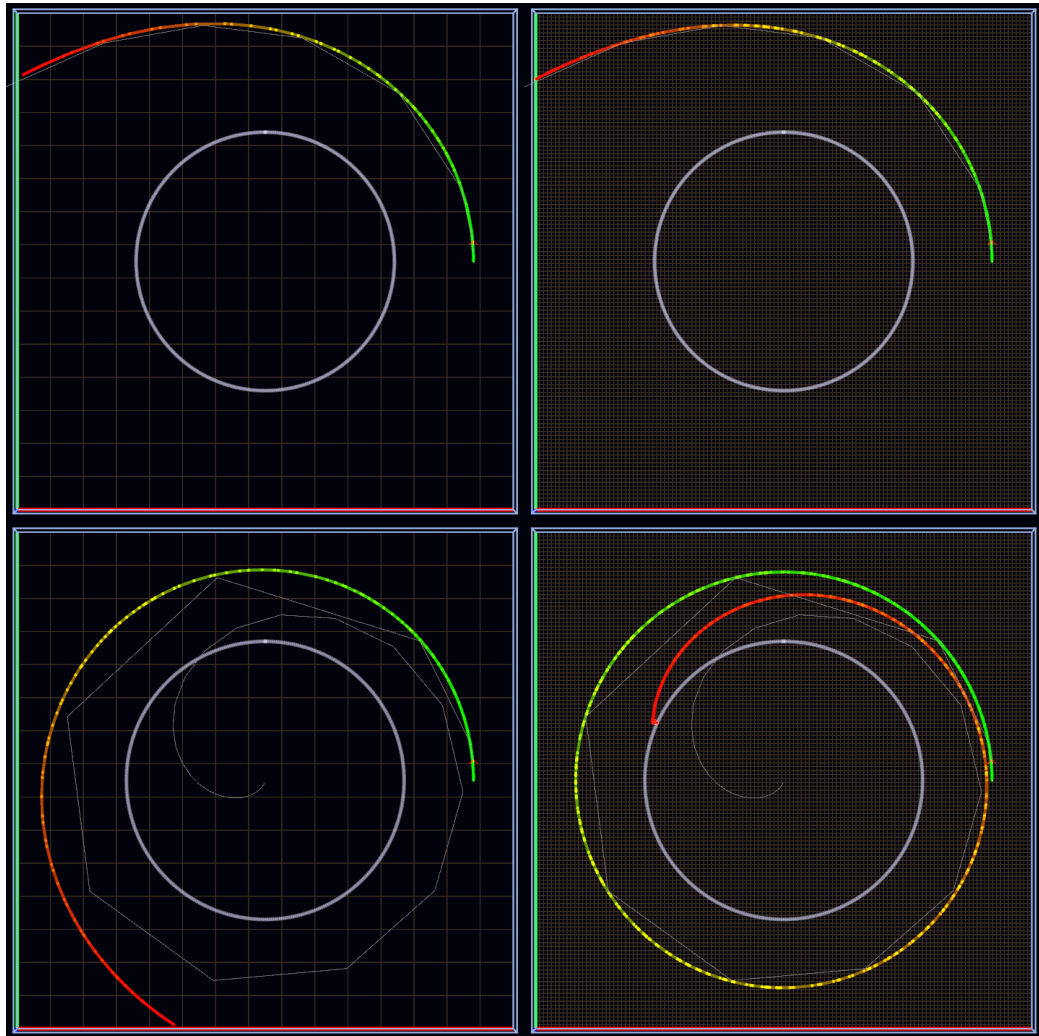
Figure 7.15: Verification of the numerical geodesic computed in a sampled Schwarzschild metric (green and red line) by comparison to the geodesic of the analytic metric (thin light blue line). The thick light blue circle illustrates the event horizon ($r = 2 \cdot m$) of the black hole. The brown lines show the cells of the uniform grid. Geodesics are computed in xy-plane, indicated by the accordingly colored lines at the border of the bounding boxes. *Top:* Mass $m = 0.31$. The geodesics match perfectly, also using the coarse (16×16) grid. *Bottom:* Mass $m = 0.3334$. An increase of the grid resolution (128×128) leads to a better match of the numerical geodesic, which breaks at the event horizon because the coordinate representation of the metric has a singularity here. The analytic geodesics is computed in polar coordinates that has no singularity at the event horizon and, thus, spirals towards the center.

when the vector magnitude becomes small. Several parameters control the sharpness of this transition.

Thus, the color of a speckle depends on the vector direction, their magnitude and the camera position. Coloring changes when the camera is rotated. To enable the coloring feature of the speckles the camera was slightly rotated around the x-axis in positive rotation direction. Otherwise all vectors would have been parallel to the camera plane.

The speckles in the upper half of the lower left figure of *figure 7.16* are colored in blue because they point towards the center and the camera. Whereas, the speckles in the lower half also point towards the center but away from the camera and are, thus, colored in red.

The singularity around the event horizon (black circle) is much better visualized than in the according image using the arrows. However, subtle changes in the vector magnitude are better shown by the arrows.

When looking at the upper left figure the magnitude of the coordinate acceleration is greater the closer a photon is located to the mass. However, it decreases towards the axis in direction of movement going through the center of the black hole. Here, the magnitude decreases towards an the axis parallel to the x-axis.

The right figures of *figure 7.16* illustrate the coordinate acceleration of particles with no initial speed. They all become attracted to the center of gravity symmetrically. The change in the vector directions inside the event horizon is clearly visible also in the bottom figure by the colors of the vector speckles.

The next *figure 7.17* illustrates the coordinate acceleration $\ddot{q}$ (yellow arrows) dependent on different directions of velocities $\dot{q}$ (blue arrows) of photons. The photons are located on a quadratic shape around the center of gravity. They all are coordinate-accelerated towards the center with a magnitude dependent on the moving direction. The maximum is located to the side of the center and the minimum in the direction of the center, with reference to the direction of movement.

This property holds when the velocity is rotated. The figures illustrate four rotations of the velocity. Starting in positive x direction (top left) and rotating to positive y direction (bottom right).

The velocity vector is transformed to the coordinate acceleration vector by the Christoffel symbols. The figures illustrate that a more complex structure than, for example, a linear transformation matrix is necessary. In case of the latter the angle between the velocity and the coordinate acceleration would remain constant over rotation. Here, the transformation by the Christoffel symbols control the magnitude of the coordinate acceleration.

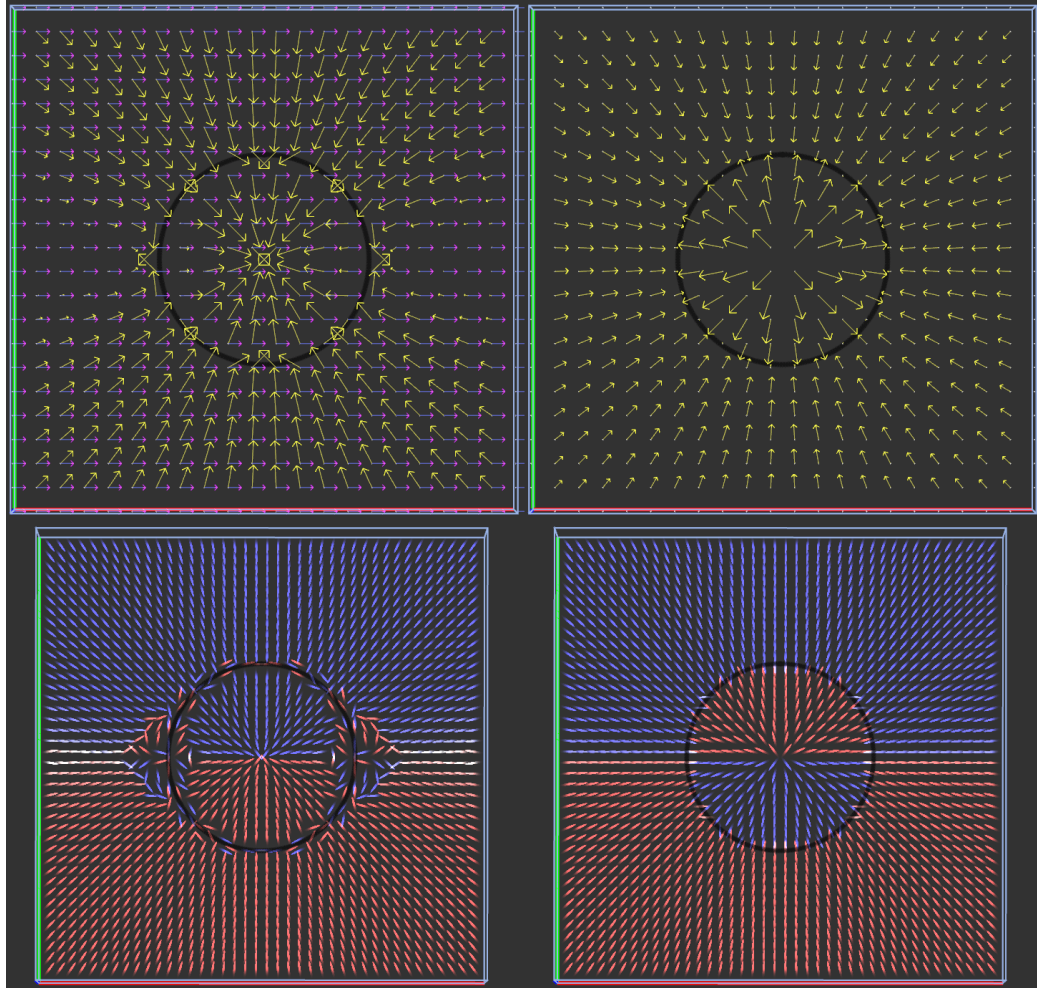*Figure 7.18* relates to *figure 7.17*. The initial directions are equal for

Figure 7.16: Schwarzschild metric is explored by the coordinate acceleration of moving photons (left) and the coordinate acceleration of particles at rest(right). *Top:* The initial velocity is shown by blue and the coordinate acceleration by yellow arrows. The vector magnitudes are clearly visible. 20×20 arrows are drawn. *Bottom:* Coordinate Acceleration is illustrated by vector speckles. Directions are clearly visible. The singularity at the event horizon is better recognized. 40×40 speckles are drawn. Increasing the number of speckles often increases the visual appearance, while the image becomes unreadable if the number of arrows is too high.
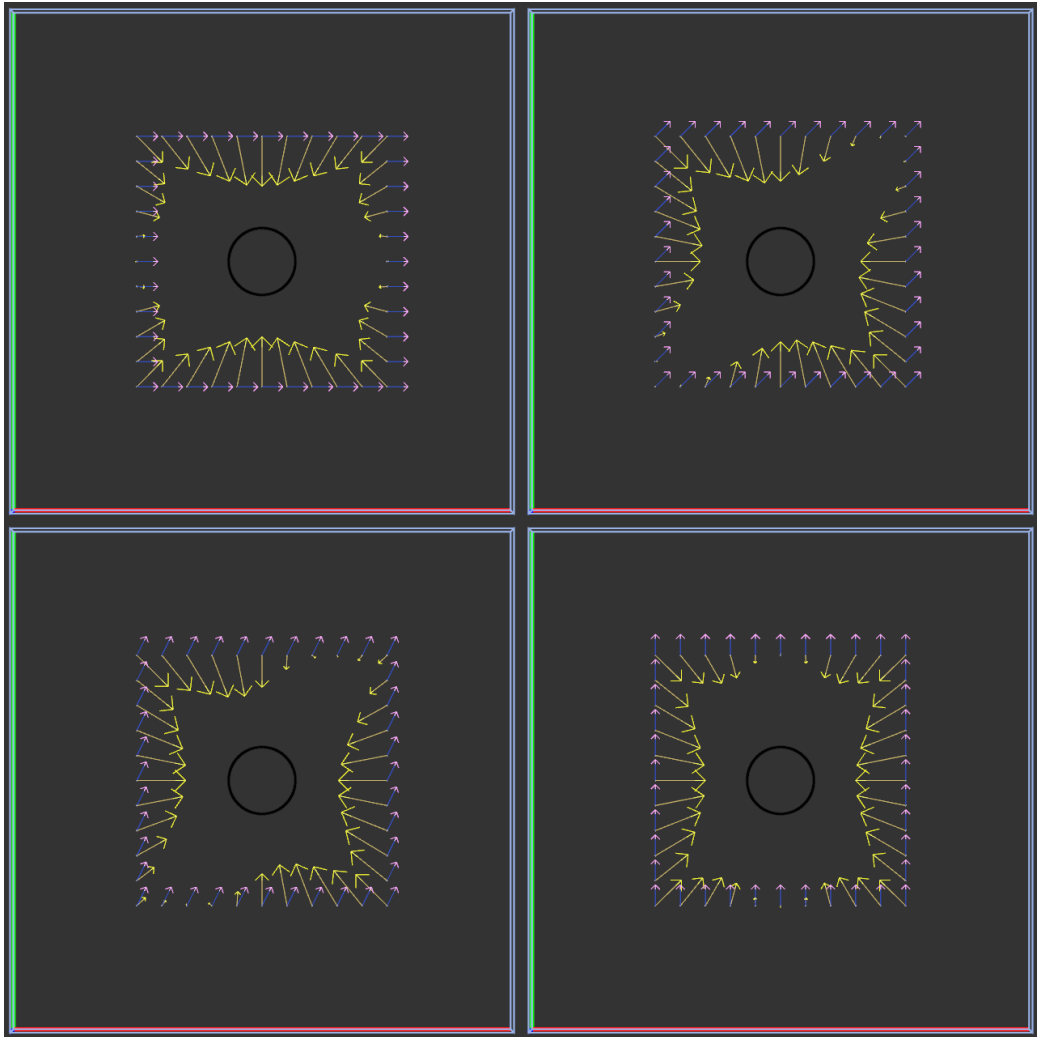
Figure 7.17: Illustration of the coordinate acceleration $\ddot{q}$ on photons moving in different directions caused by the Schwarzschild metric. The transformation by the Christoffel symbols is visualized. Velocity is illustrated as blue and coordinate acceleration as yellow arrows. Photons travel in positive x-direction (top left) and are rotated to positive y-direction (bottom rigth).

each sub-figure. Instead of the square of arrows, vector speckles are drawn in a regular grid. Initial direction become visible by the white color of the speckles, where vector magnitude becomes small.

The two-dimensional Schwarzschild metric finally is explored by a fan of geodesics. *Figure 7.19* shows 26 geodesics seeded at $x = -1$ in positive x direction. The metric field has a resolution of 128×128. The upper figures show geodesics the field with mass $m = 0.01$. The mass was increased to $m = 0.1407$ for the two lower figures. The step sizes of the integration are directly visualized by the yellow dots on the lines.

Comparing the upper figures, a coarse Euler and accurate Dop853 integration, it turns out that the resulting geodesics are very similar. The Euler integration yields very good results.

I had to introduce a breaking criterion for the Euler integration, such that geodesics will not overshoot through the singularity. A combination of three criteria turned out to work very well. The Euler integration stops when:

- The new step size becomes greater than a constant s.

- The angle of the new line segment to the old line segment exceeds a constant angle ξ.

- The computation of the norm of the new direction vector gets infinite.

When either criterion is reached the integration stops. Practical values for the constants are: $S = 4$ and $\xi = \pi/3$. These values are used throughout the figures. These criteria could easily be made configurable by the user in the GUI. Similar criteria could be used to improve the Euler integration by a adaptive step size control.

The Dop853 integration was not equipped with any new breaking criterion. It automatically refines the step size at the singularity and stops because the integration gets too small some when.

Comparing the lower figures in *figure 7.19* shows that geodesics far from the event horizon are very similar. However, if a geodesic gets close to the even horizon the integration methods yield different results. For example, the fifth geodesic (counted from top) is orbiting halfway around the mass using Euler (bottom left), but orbiting fully around the center using Dop853.

The geodesics are perfectly symmetrical as expected for the symmetrical metric field. Geodesics getting too close to the center fall into the black hole and their bending decreases with distance from the center.

There is quite a difference in the computation speed of the integration methods here. Some time measurements of the computations shown in *figure*
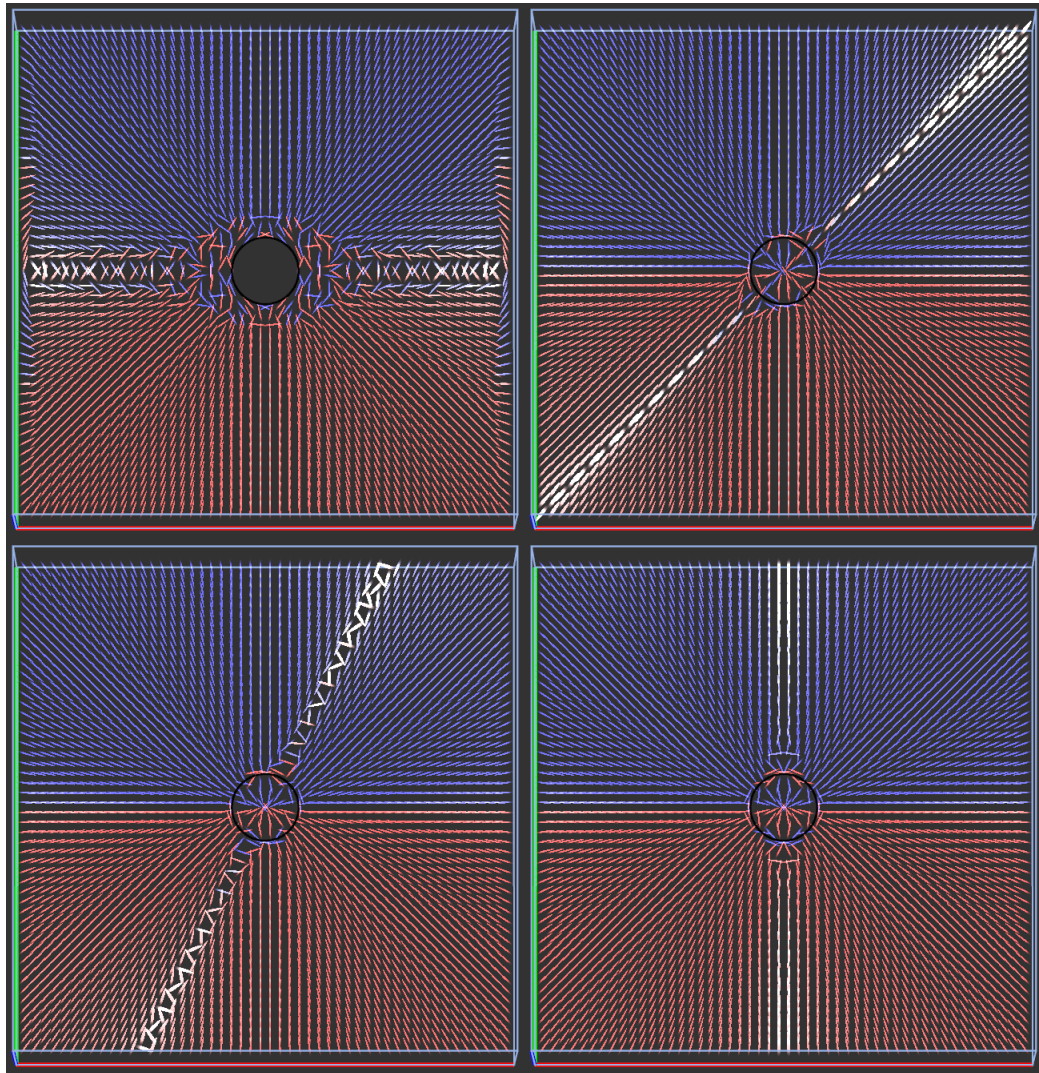
Figure 7.18: Viusalization of $\ddot{q}$ computed from different velocities. This figure relates to *figure 7.17*. Directions of the velocity are $(1, 0, 0)$ at *top left*, $(1, 1, 0)$ at *top right*, $(0.5, 1, 0)$ at *bottom left* and $(0, 1, 0)$ at *bottom right*. These directions are shown by the speckles in white color because here the magnitude of $\ddot{q}$ becomes small.

7.19 are gathered in the following tabluar[4]:

Integration	Steps	Stepsize	Overall Time [sec]	Time per Step [sec]
Euler $m = 0.0100$	1276	0.5236	0.67	0.0005
Dop853 $m = 0.0001$	2619	-	19.95	0.0076
Euler $m = 0.1407$	1012	0.5236	0.515	0.0005
Dop853 $m = 0.1407$	6402	-	48.81	0.0076

The integration for one step differs by a factor of about 10. The big difference in the overall computation time comes from the huge difference in the number of integration steps. Also, the Dop853 has to do about eight function evaluations per integration step. When a geodesic approaches the event horizon the adaptive Dop853 is fighting with the singularity until it finally breaks. A better and user defined breaking criterion would speed up the Dop853 integration.

The pure visualization of the geodesics can be enriched by showing the coordinate acceleration along the integration lines. *Figure 7.20* shows such enhanced geodesics. The integration line module stores the directions of the integration steps on the line grid. Again, the `ComputeQddot44` module is utilized to compute the coordinate acceleration. In fact, $\ddot{q}$ could be stored in the line grid as well. Althought utilizing `ComputeQddot44` could be avoided it is still be more flexible approach, since the module can be used with any vector field.

In the upper figures $\ddot{q}$ is illustrated by arrows and in the lower figure vector speckles are used. In the left figures geodesics are integrated using Dop853 using the same parameters as in *figure 7.19*. The right figures are computed using Euler with two different step sizes. Again, the speckles give a better visual impression.

---

[4]Measurements were done in SVN Revision 1820 of *Vish* on a dell M1330 XPS latop with 4GB RAM, Intel Core2 Duo T8300 @ 2.4Ghz and NVidia Geforce 8400M GS graphics card. Compiled in Windows Vista 32 using gcc 3.4.2 (mingw special) in debug mode.
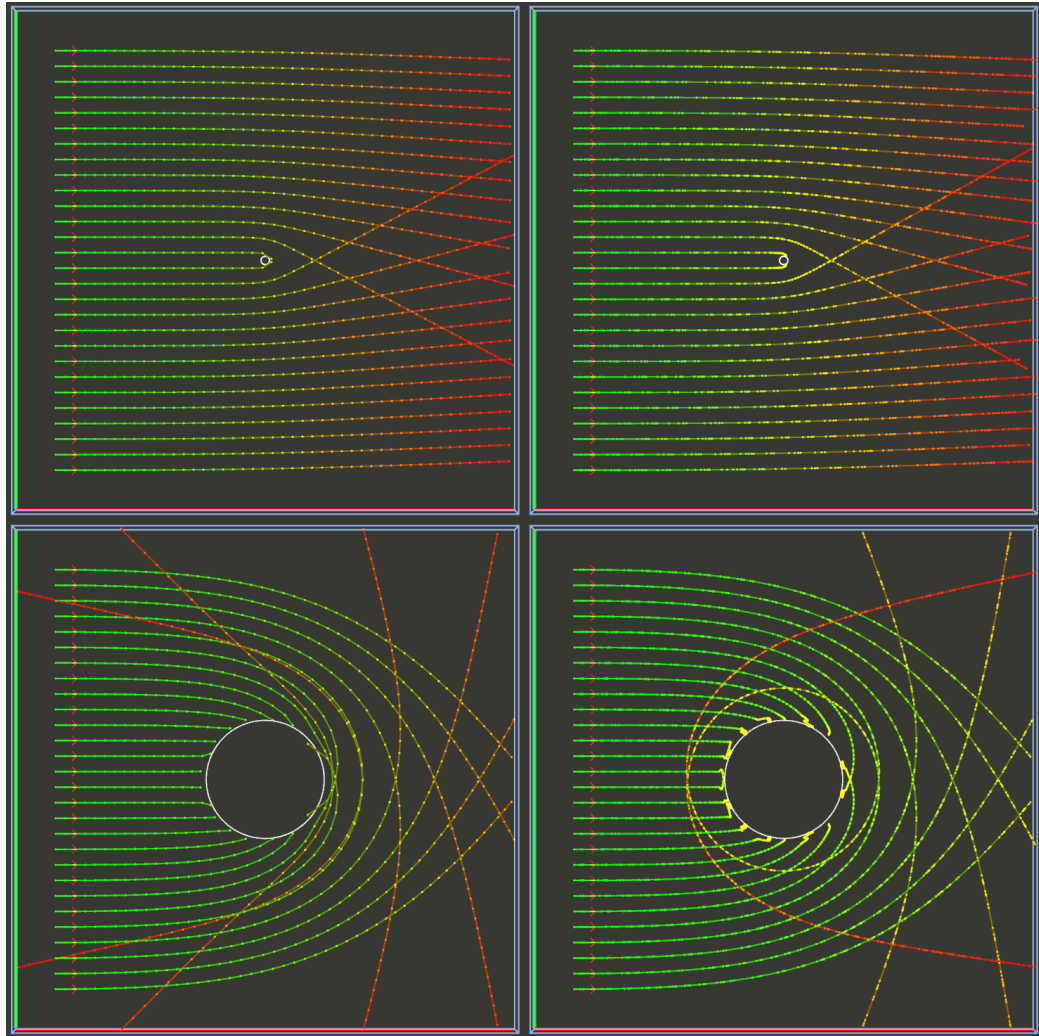
Figure 7.19: A fan of 26 geodesics is seeded at $x = -1$ in positive x direction. The left figures are computed using Euler and a step size of 0.5326 and the right figures are computed using Dop853. The mass for the upper figures has a value of $m = 0.01$ and in the lower figures $m = 0.2$. Geodesic are symmetric in y direction. Euler and Dop863 integration match very well with small mass and big distance from the center. When getting close results differ, compare lower images: the fifth geodesic counting from top.
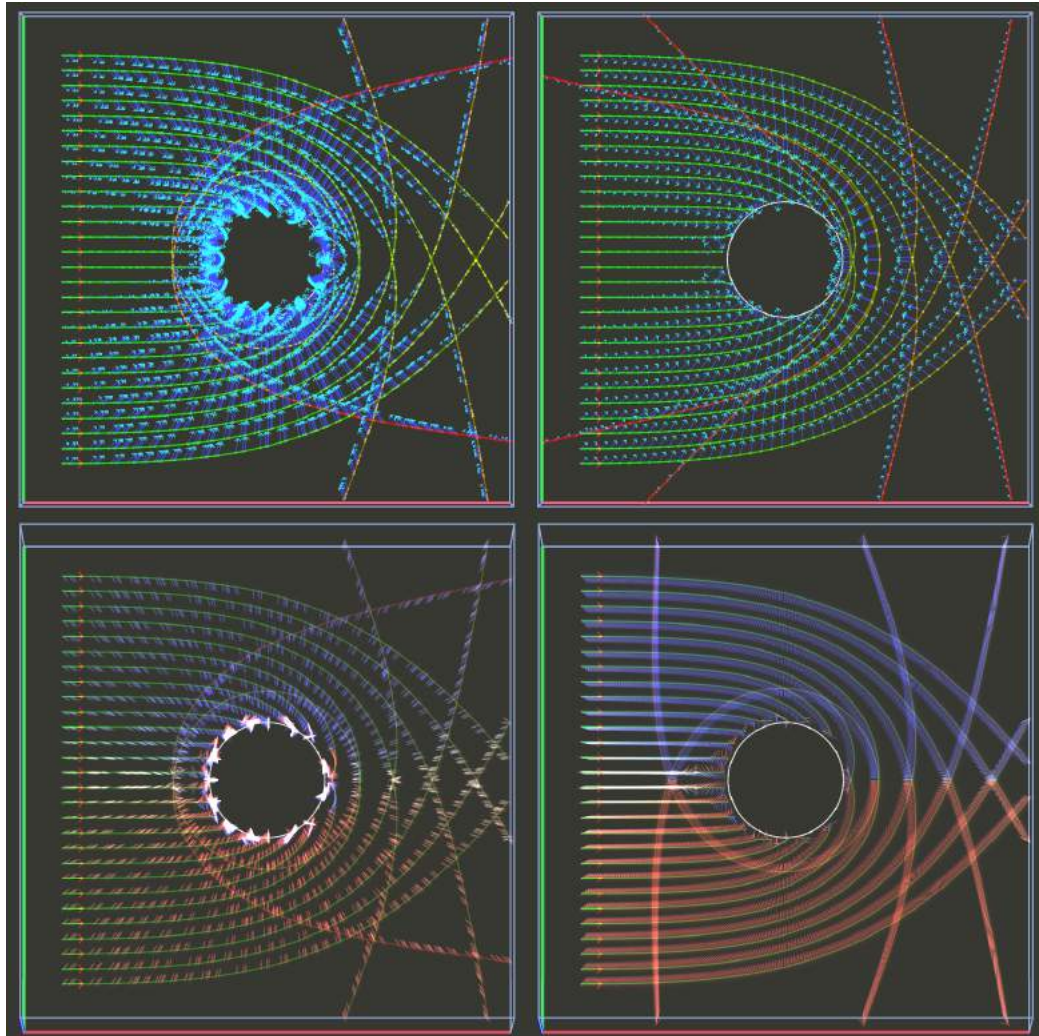
Figure 7.20: Geodesics enhanced by showing the coordinate acceleration along the lines. *Left:* Dop853 integration. *Right:* Euler integration with different step sizes. Arrows are used in the upper figures, which become hard to read when too many arrows are drawn, see center of left upper figure. Vector speckles illustrate the coordinate acceleration in the lower figures. They remain readable even when many speckles are drawn.

The next step was to move to the full 4D Schwarzschild metric with variable θ, see eq(). I extended the metric sampling module and started to analyze the space-time first by looking at $\ddot{q}$. *Figure 7.21* shows different parameters for the mass using, again, the speckle technique. Also $\dot{q}$ is pointing in positive x-direction. The singularity at the event horizon is illustrated clearly by the three axis aligned speckle planes.

Then I verified the spherical symmetry of the metric by seeding geodesics on a circular shape in the yz-plane. *Figure 7.22* illustrates the geodesics that get bent symmetrically around the black hole. Mass was increased starting from $m = 0.1$ to $m = 0.3$. The symmetry is best shown in the left lower figure where all geodesics intersect in one point after having orbited around the center of mass.

Another figure illustrating was created by seeding geodesics on a vertical and horizontal line in the yz-plane. *Figure 7.23* shows the result. I noticed some geodesics that are reflected at the event horizon, see left figure. Six geodesics starting close to the center of the seeding cross get reflected. Increasing the sampling of the metric field from $64 \times 64 \times 64$ to $128 \times 128 \times 64$ yielded a better result, see right figure. Thus, I used the increased sampling for the following analysis.
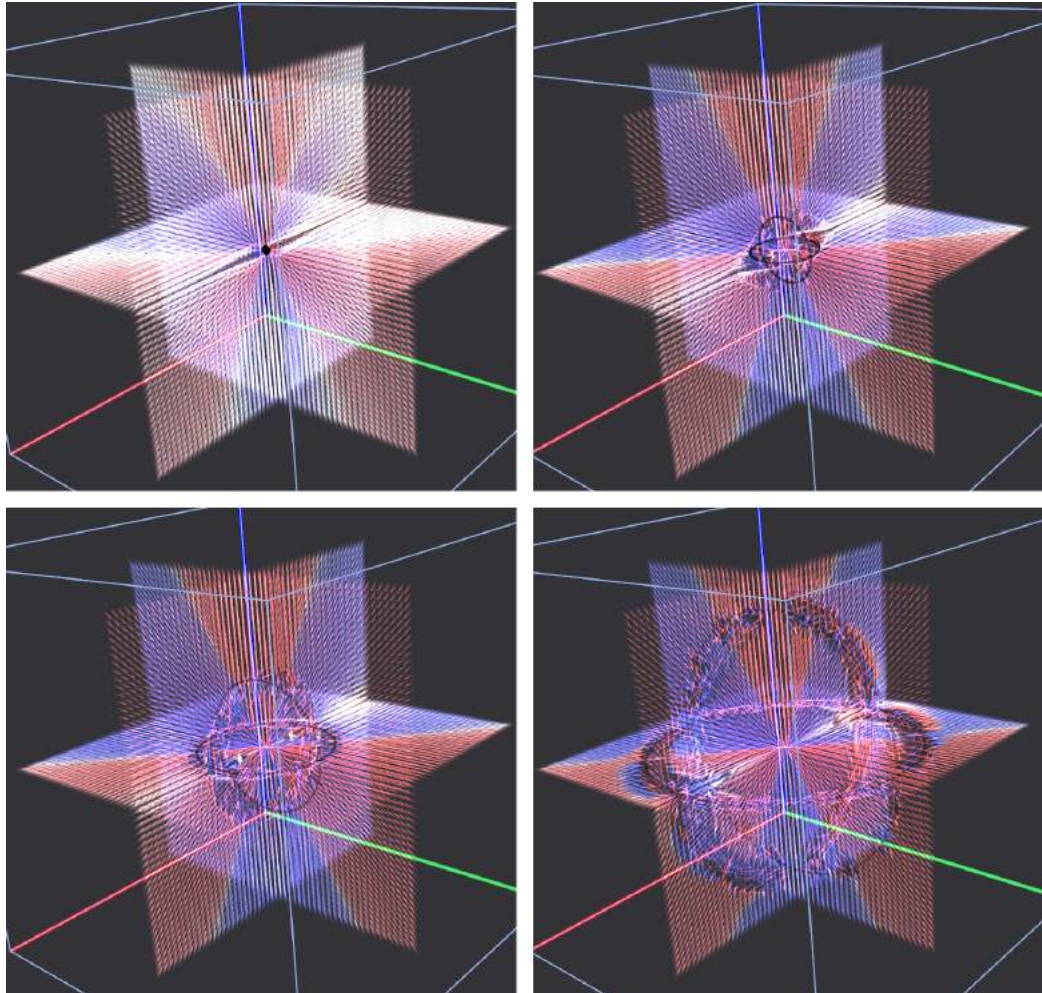
Figure 7.21: Vector speckles illustrating $\ddot{q}$ sampled on three axis aligned planes of a 3D Schwarzschild based on a $\dot{q}$ going in positive x direction. The coordinate acceleration is pointing to the center showed by the direction and color of the speckles. The spherical singularity at the event horizon(black circles) is clearly illustrated. Also, the small magnitude of the vectors at the x-aligned axis going through the center of the black hole is indicated by the white color.
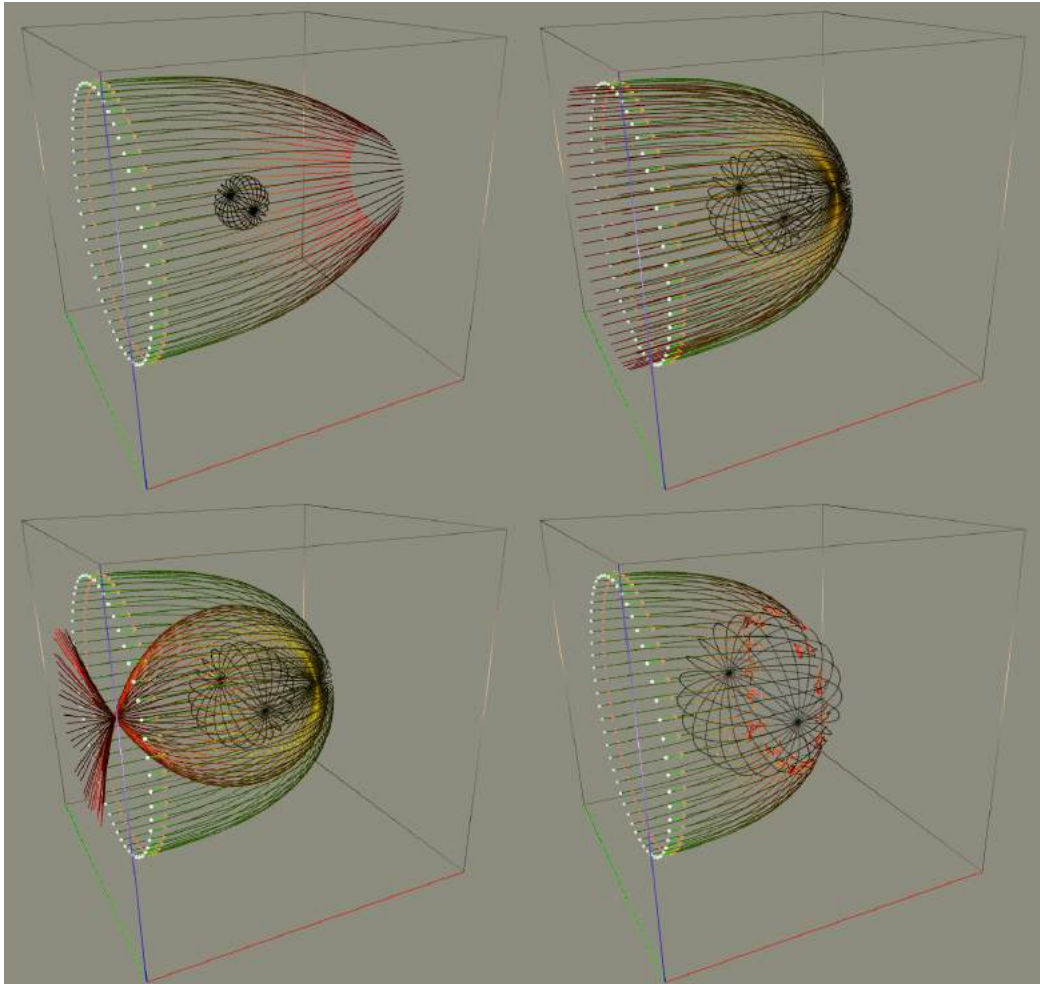
Figure 7.22: Geodesics seeded on a circular shape in yz-plane, $r = 2.0$. The black sphere shows the event horizon. The spherical symmetry of the Schwarzschild metric is illustrated using different values of m. *Top:* Geodesics are computed using Euler integration, $m = 0.1$ and $m = 0.2$. *Bottom:* Dop853 integration is used, $m = 0.194$ and $m = 3.0$
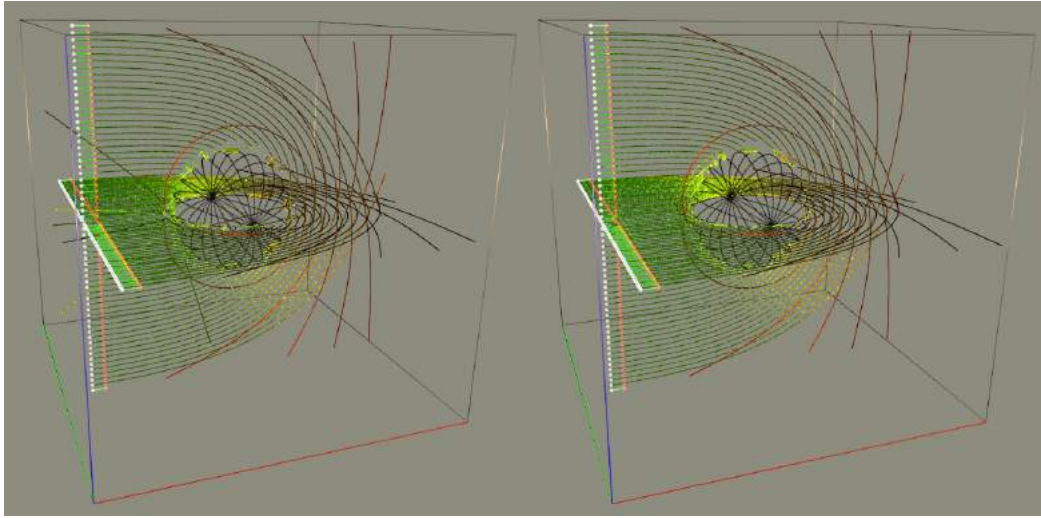
Figure 7.23: Geodesics seeded on a cross like shape. The symmetry of the metric field results in symmetric geodesics. Dop853 integration is used. *Left:* Too coarse sampling resolution yields in reflected geodesics ($64 \times 64 \times 64$). *Right:* Increased resolution gives better results. No geodesics are reflected at the singularity ($128 \times 128 \times 64$).

The next two figures illustrate Geodesics enriched with further information. The $4th$ dimension of the coordinate acceleration was mapped as a color on the geodesics in *figure 7.24*. As photons approach the black hole they become coordinate-accelerated in time: the stronger the closer. After passing the black hole coordinate acceleration in time becomes zero (dark red color).

Seeding a tube-like shape visualizes contraction or, in this case, expansion of the photons as they move in the space time. This can be interpreted as visualization of the Riemann tensor, *section 2.1.8*. The deviation of the geodesics are well illustrated by the tube-like shape visualizing the tensor of rank four.

In *figure 7.25* vector speckles where added to visualize the spatial coordinate acceleration, like in *figure 7.20*. Now, the full $4D$ coordinate acceleration is visualized along the geodesics. This setup was chosen for analyzing the light paths in the Kerr metric in the following chapter.
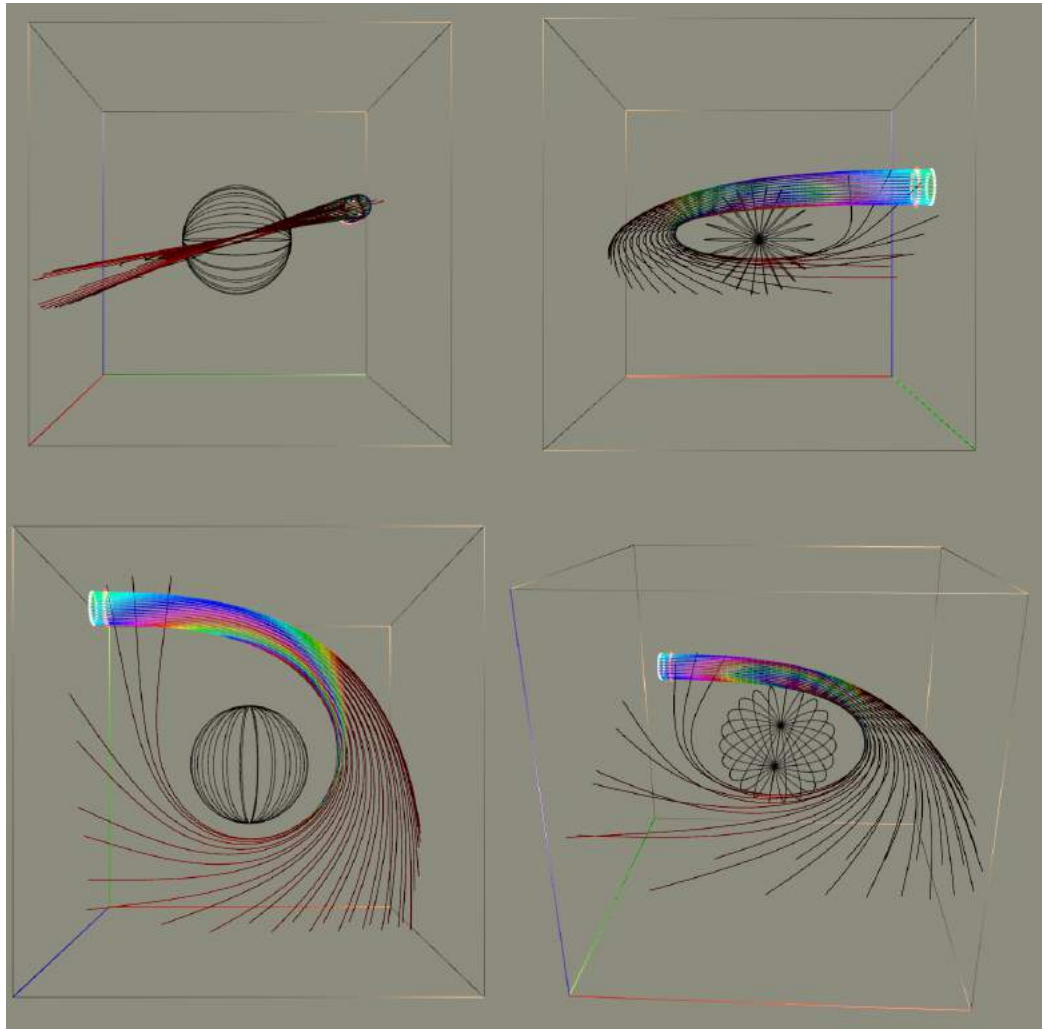
Figure 7.24: Geodesics forming a tube-like shape. Expansion is illustrated. The color on the lines represent the time component of $\ddot{q}$. It becomes zero after passing the black hole. To visualize the geometry of the $3D$ lines three axis aligned camera setups and one free camera position were chosen. The tube-like shape reveals the deviation of the geodesics and thus visualizes the Riemann tensor, which is a tensor of rank four.
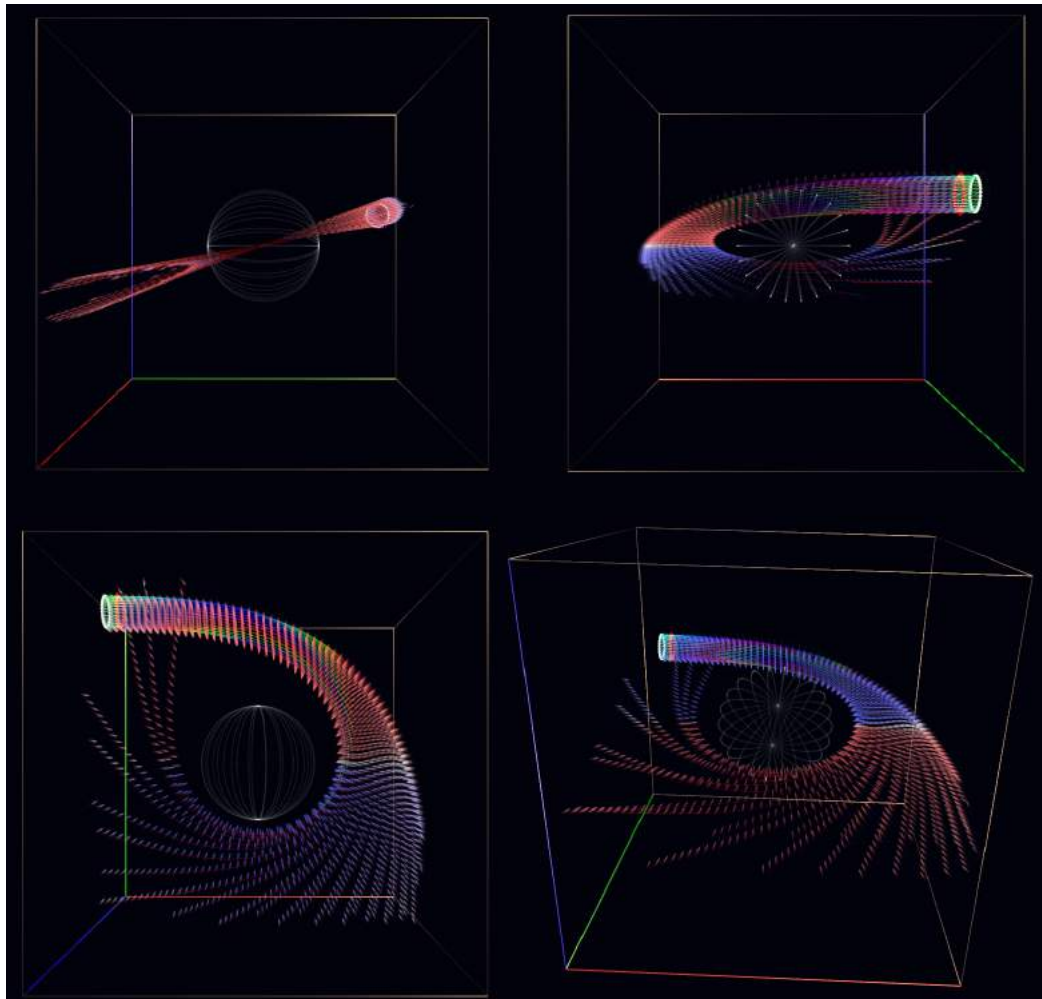
Figure 7.25: Like *figure 7.24*. Vector speckles were added to illustrate the spatial components of $\ddot{q}$.
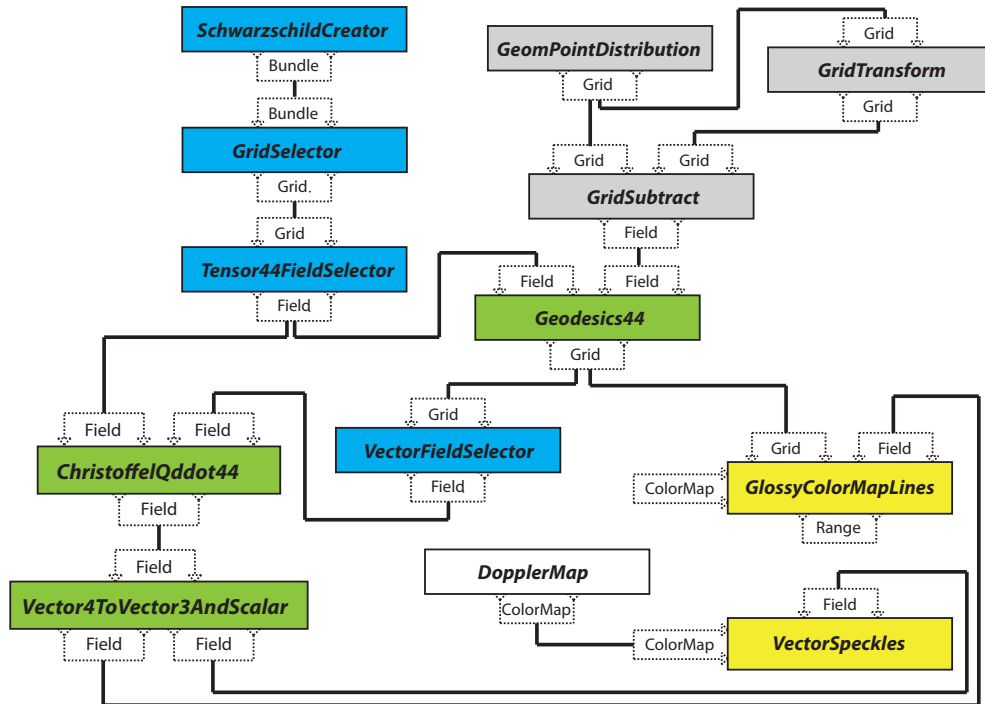
Figure 7.26: Schematic *Vish* network of the visualization shown in *figure 7.25*. Seeding geometry is copied and transformed to compute a vector field by a grid subtraction (grey). This field defines the initial conditions for the computation module. The four dimensional tensor field is extracted from the creator module and handled to the geodesic computation module `Geodesics44`. The lines grid is passed to the line rendering module `GlossyColorMapLines`. From the line grid the tangential direction field is extracted and used to compute $\ddot{q}$ by `ChristoffelQddot44`, which must also be connected to the metric tensor field. The resulting four dimensional coordinate-acceleration is split into a three dimensional spatial vector field and a scalar representing time by `Vector4ToVector3AndScalar`. The scalar field is passed to the line rendering module driving the coloring and the vector field is passed to the `VectorSpeckles` drawing color mapped speckles of $\ddot{q}$ along the geodesics.

7.4   Visualizing Geodesics in a sampled Kerr Metric

After verifying the geodesic computation and exploring the symmetrical Schwarzschild field, a more complex spacetime is analyzed using the previously described techniques. This chapter focuses on the visualization of the Kerr metric, *section 2.2.3*. Thus, emphasis is on the figures. Textual explanations are kept short. "A picture is worth a thousand words". The previous chapter should have been read to be able the read and interpret the figures correctly.

Again, the a *Vish* module was created to sample the metric field on a uniform grid. Therefore, *equation (2.74)* and *equation (2.15)* were utilized. An additional parameter now controls the angular momentum of the black hole around the z-axis. The Kerr metric is still a stationary field without evolution over time. Thus, the geodesics computed here are an analogue to streamlines in CFD.

To explore the Kerr metric I use series of figures with varying angular momentums. Often more than one camera perspective is necessary to illustrate the $3D$ geometry unambiguously.

First some figures show the coordinate-acceleration vector by given initial speed, again, always along the positive x-direction. I tried several seeding geometries. First, the $3D$ cross similar to *figure 7.21*, see *figure 7.27*. The angular momentum is increased by constant mass. Comparing the xy-planes of the two upper figures shows that the coordinate acceleration is starting to spin clockwise around the center of gravity. What is shown clearly is that the field loses its spherical symmetry around the center.

Next, geodesics were seeded in the xy-plane of the metric in x-direction. Again, starting the exploration in a reduced two/dimensional case. *Figure 7.28* and *figure 7.29* show a series of eight figures with increasing angular momentum. The angular momentum is from left to right and from top to bottom:

$a = 0.0,$
$a = 0.01,$
$a = 0.05,$
$a = m = 0.15;$
$a = 0.25,$
$a = 0.5,$
$a = 0.75,$
$a = 1.0.$

The illustrations with $a > m$ maybe have no physical meaning. Here, the

event horizon at r_+ becomes undefined, *section 2.2.3*. However, the parameter analysis was also done with some out-of-range values since they show characteristics in an over-exaggerated way.

The white or black sphere in the following figures represents the event horizon of a Schwarzschild black hole with the corresponding mass. When looking at the locations where the geodesics break, it is observed that the event horizon r_+ of the Kerr black hole shrinks as the angular momentum increases.

The first figure in *figure 7.28* illustrates how the geodesics on the upper half are bent toward the center and collide earlier with the black hole, while the geodesics on the lower half are bent away from the black hole. The singularity moves inwards with increasing angular momentum. Geodesics that are bent away from the black hole coordinate-accelerate in time component again after having passed the black hole.

When increasing the angular momentum over the value of mass, geodesics on the lower half are straightened out by the rotation, see right top figure of *figure 7.29* ($a = 0.5$) and finally become reflected.
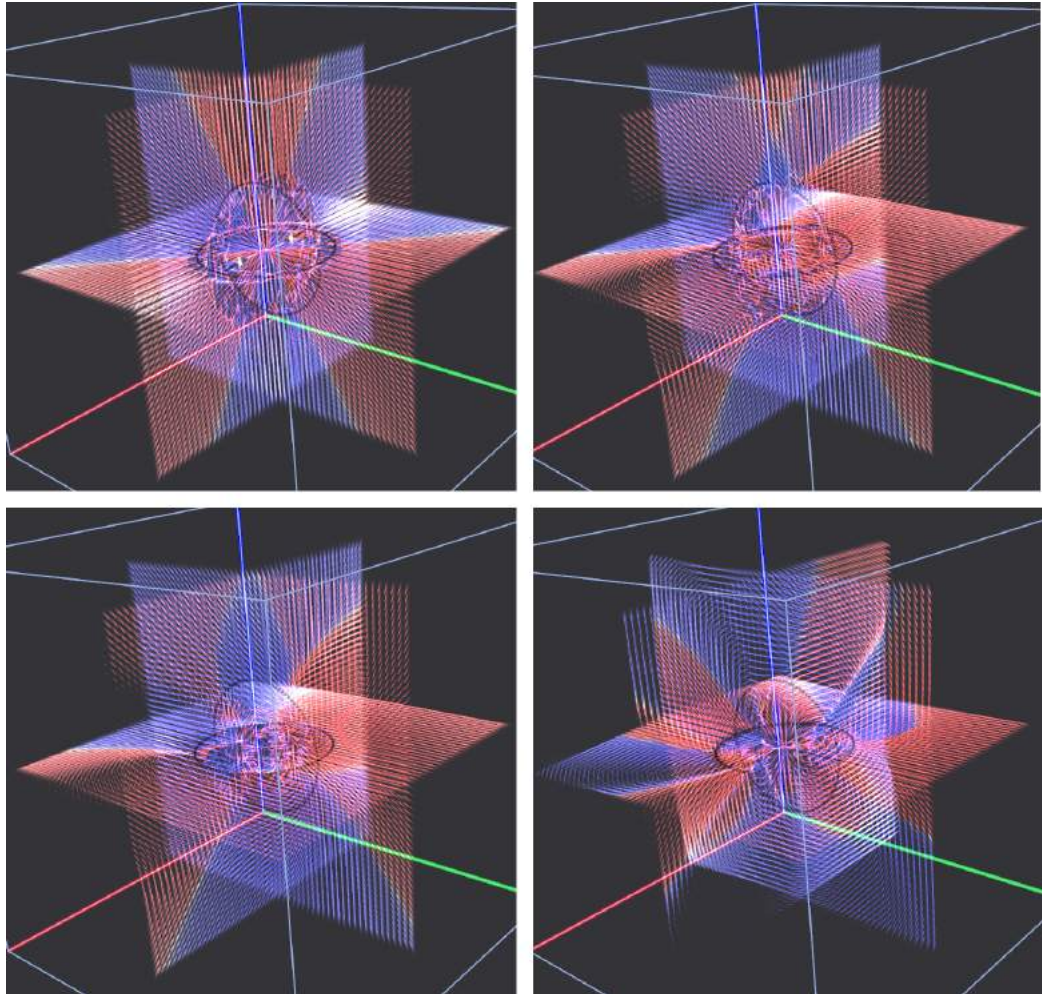
Figure 7.27: Vector speckles on axis aligned planes in a Kerr metric illustrating $\ddot{q}$ with $\dot{q} = (1, 0, 0)$. The angular momentum a is increased, but holding the mass $m = 0.2$. From *left* to *right* and *top* to *bottom*: $a = 0.0$, $a = 0.1$, $a = m = 0.2$, $a = 1.0$. The Kerr event horizon at r_+ shrinks when the angular momentum is increased. The corresponding Schwarzschild event horizon is illustrated by the circles (black). The coordinate acceleration is spinning around the center, what is illustrated on the xy-plane.
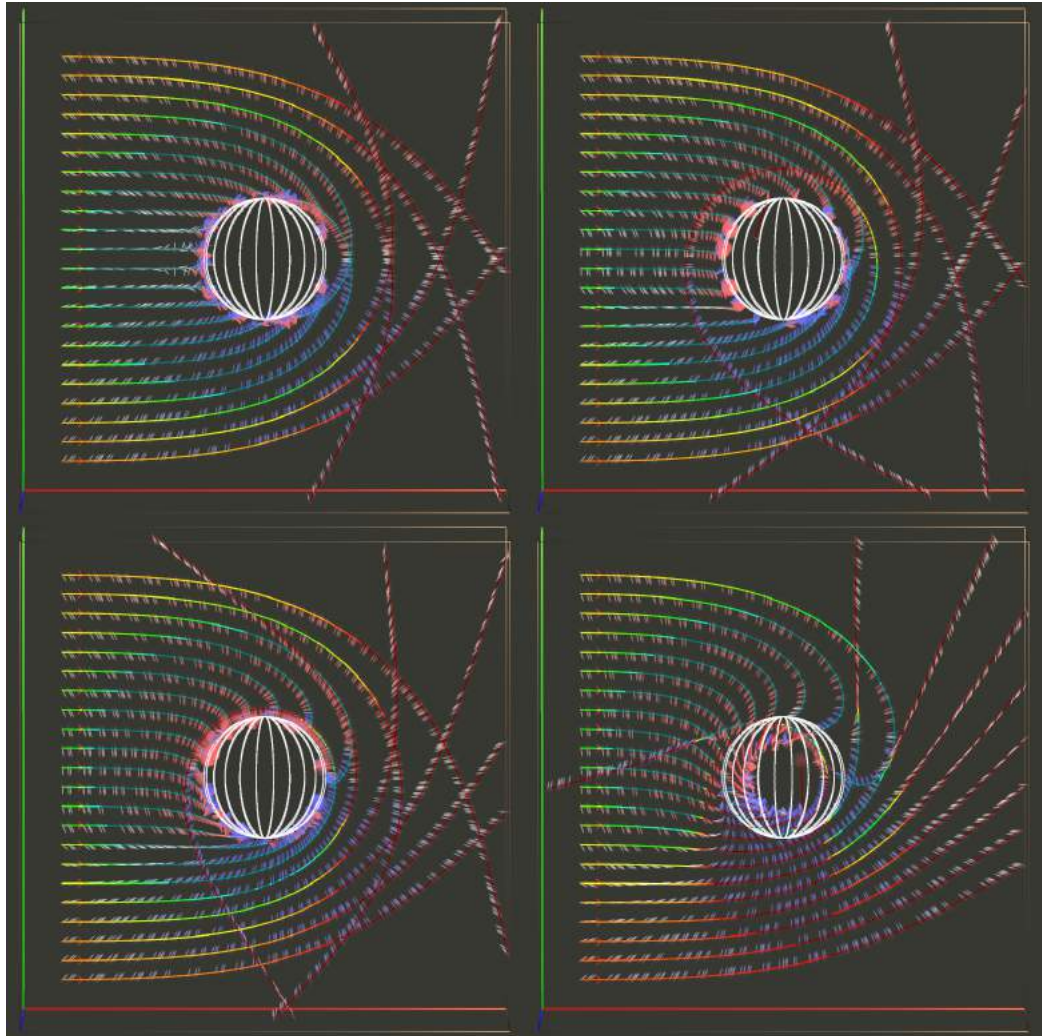
Figure 7.28: Geodesics seeded in the xy-plane on the line $x = -1.0$ in positive x-direction. With constant mass $m = 0.15$ the clockwise angular momentum a is increased. Geodesics in the upper half get more attracted to the black hole, whereas the geodesics in the lower half are less attracted. The r_+ event horizon shrinks with increasing angular momentum. When geodesics pass the black hole they are not further coordinate-accelerated in time.
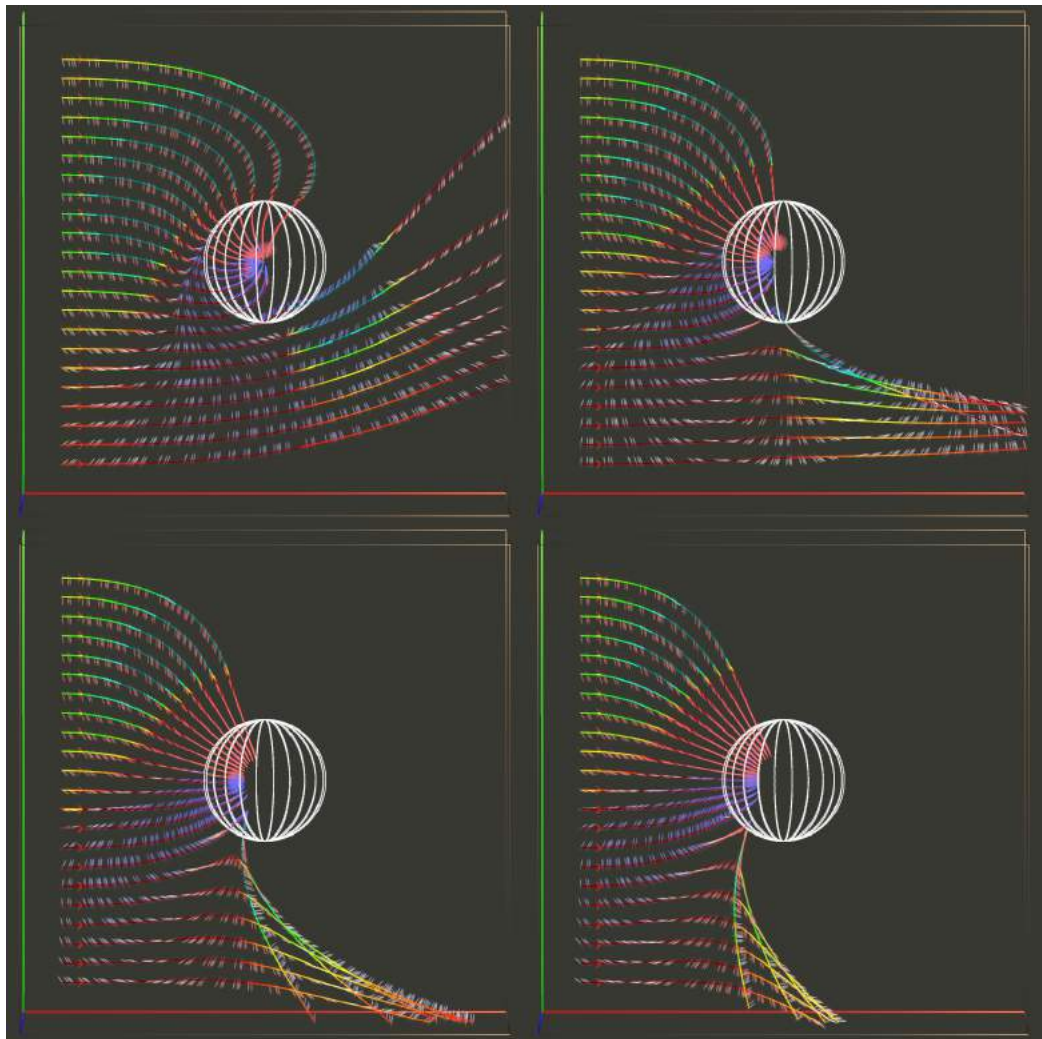
Figure 7.29: Continuation of *figure 7.28*. Here $a > m$ and hence the illustrations have no physical meaning.

Before seeding geodesics in the $3D$ volume I tried to look at the coordinate acceleration of the whole volume. First I tried to do some cubic slices as shown in *figure 7.30*. Seeding was done by using three `GeometricPointDistribution` modules for the cubics together with two `GridAdder` operations and one `GridTransform` with one `GridSubtractor` to create the $\dot{q}$ vector field.

The field is symmetric in the beginning (top left). Here, also the spherical singularity is observable. When increasing the angular momentum one can see that the directions are influenced by the momentum. With small momentum the influence applies at the nearer regions. As it grows also regions further away from the center are changing directions. The spherical singularity vanishes as it shrinks into the smallest cube.

The vector-speckle module allows to animate the speckles in a repeated cycle motion to indicate the direction. The animated speckles give a very good visual impression of $\ddot{q}$.

I tried a different seeding strategy over the whole volume. The `RandPointDistribution` was used to create randomly distributed speckles. *Figure 7.31* again shows different values for the angular momentum, which is zero in the top left figure. Surprisingly, the randomly distributed speckles give a quite good illustration over the volume and certain regions of the coordinate-acceleration flow become visible. Interactive camera rotation helps a lot for exploration. Illustrating axis-aligned views reveals some properties of the randomly speckled flow field.

Figure 7.32 illustrates some properties of the coordinate acceleration flow field based on a movement in positive z-direction. The flow field is symmetrical to one plane in the upper and lower images. It is even symmetric to two planes in the mid figures. The left figures show an angular momentum of $a = 0.2$ and the right figures $a = 1.0$. Mass is set to $m = 0.2$ for all figures.
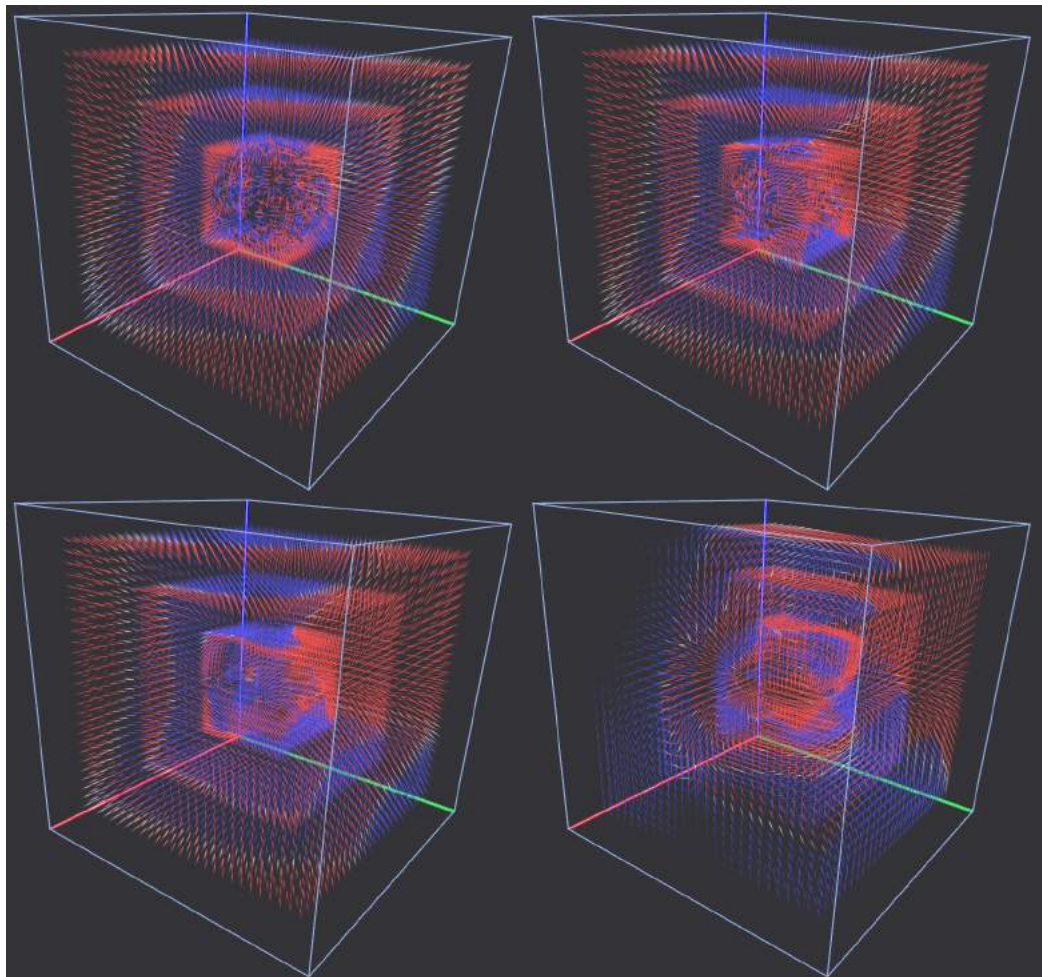
Figure 7.30: Vector speckles on cubic shapes around the center of mass illustrate $\ddot{q}$. *From left to right and top to bottom: $a = 0.0$, $a = 0.1$, $a = m = 0.2$, $a = 0.5$.* The spherical singularity vanishes and the spherical symmetry of $\ddot{q}$ is lost with increasing angular momentum.
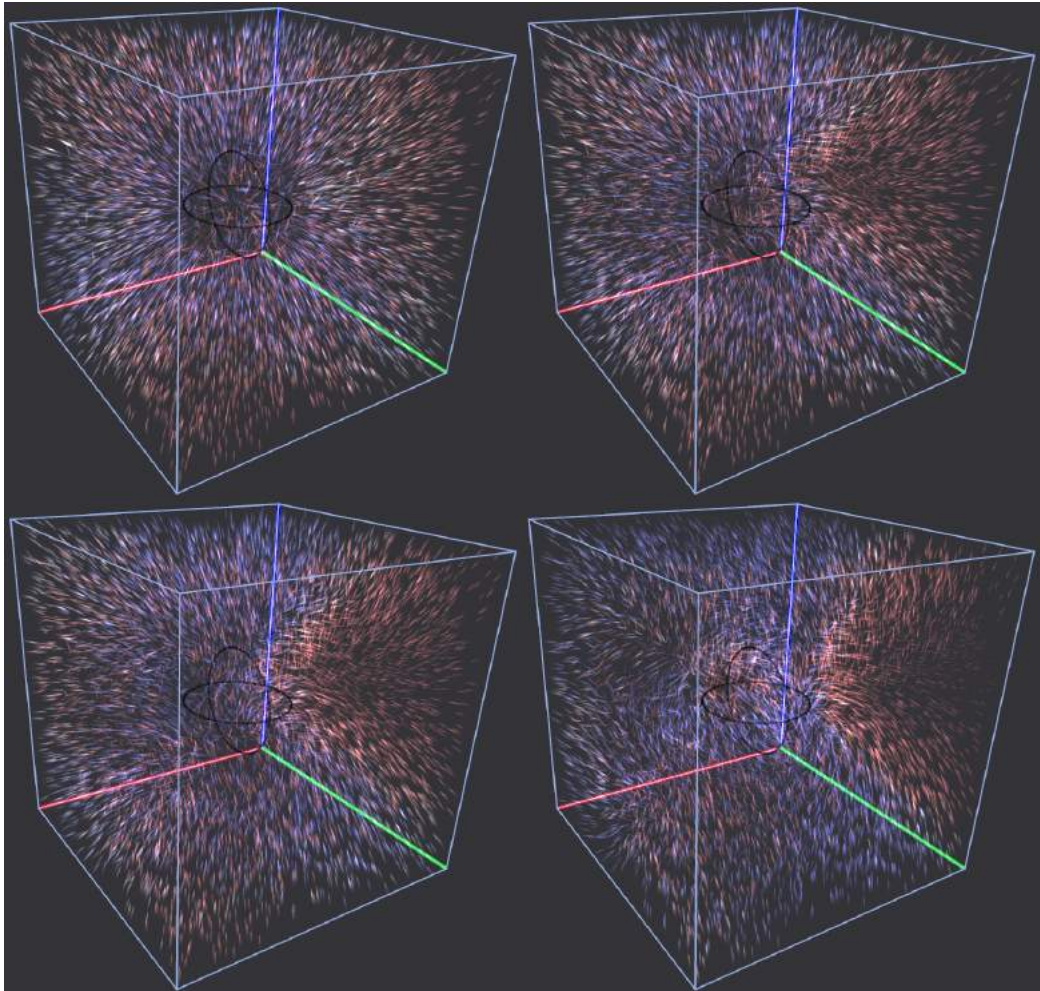
Figure 7.31: Randomly distributed vector speckles illustrate $\ddot{q}$ over the whole grid volume. From left to right and top to bottom: $a = 0.0$, $a = 0.1$, $a = m = 0.2$, $a = 0.5$. This method gives a good impression over the whole volume showing vortices and flow concentrations, not visible in *figure 7.30*. Especially when enabling the repeated animation of the speckles the flow is visualized very well. On paper this is not possible. But some axis-aligned illustration of the same kind reveal some properties as well, see *figure 7.32*.
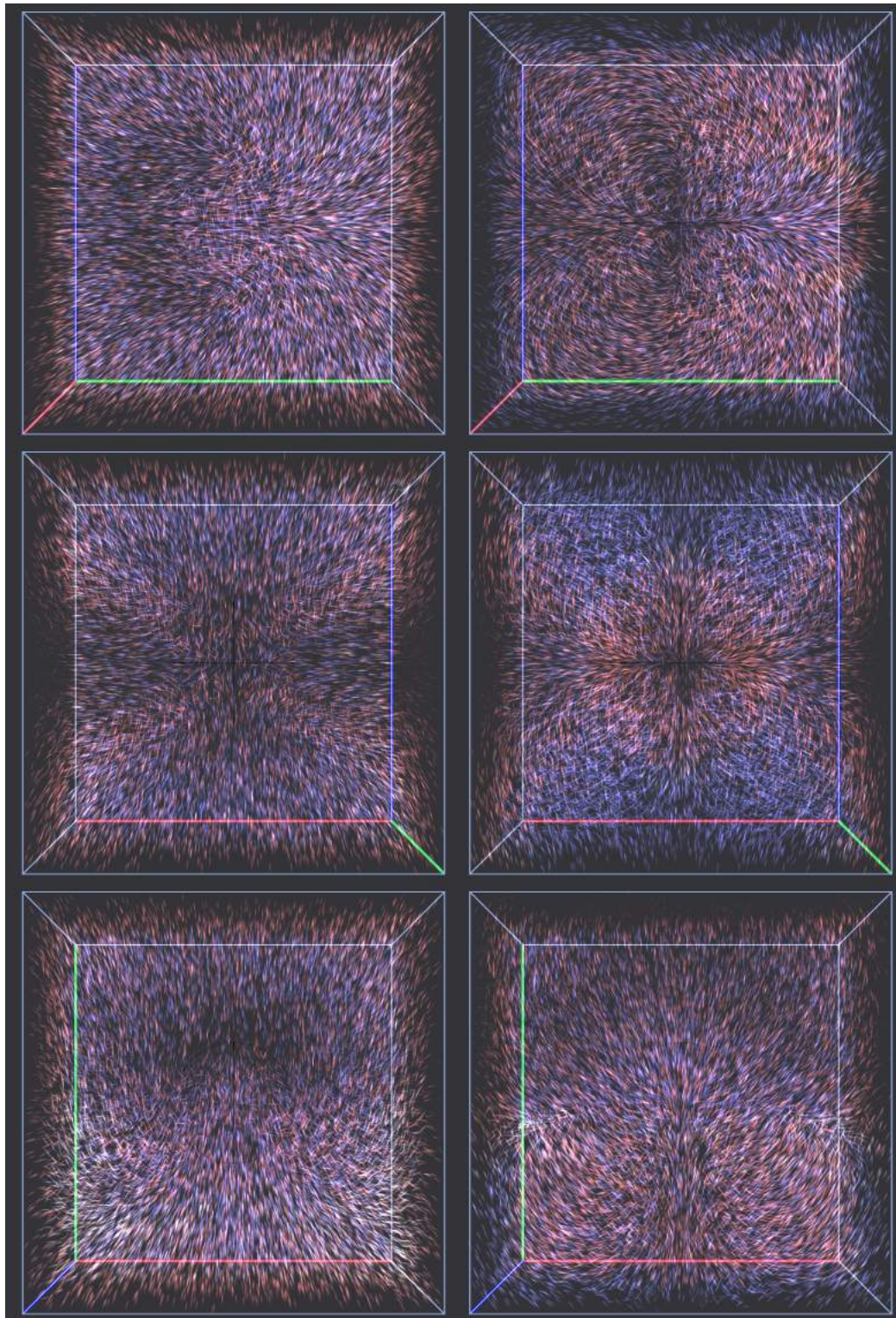
Figure 7.32: Axis-aligned camera perspectives illustrating the coordinate acceleration flow in positive x-direction of two different angular momentums. Symmetries of the flow field is clearly visualized. *Left*: $a = 0.2$. *Right*: $a = 1.0$.

Next, geodesics on a centered vertical line are illustrated in *figure 7.33* and *figure 7.34*, once again showing a series with increasing angular momentum. The upper left figure is similar to *figure 7.23*. By increasing the momentum the geodesics below and above the event horizon are bent in direction of the positive y axis and are less attracted to the center of gravity. The geodesics are symmetrical to the xy-plane.
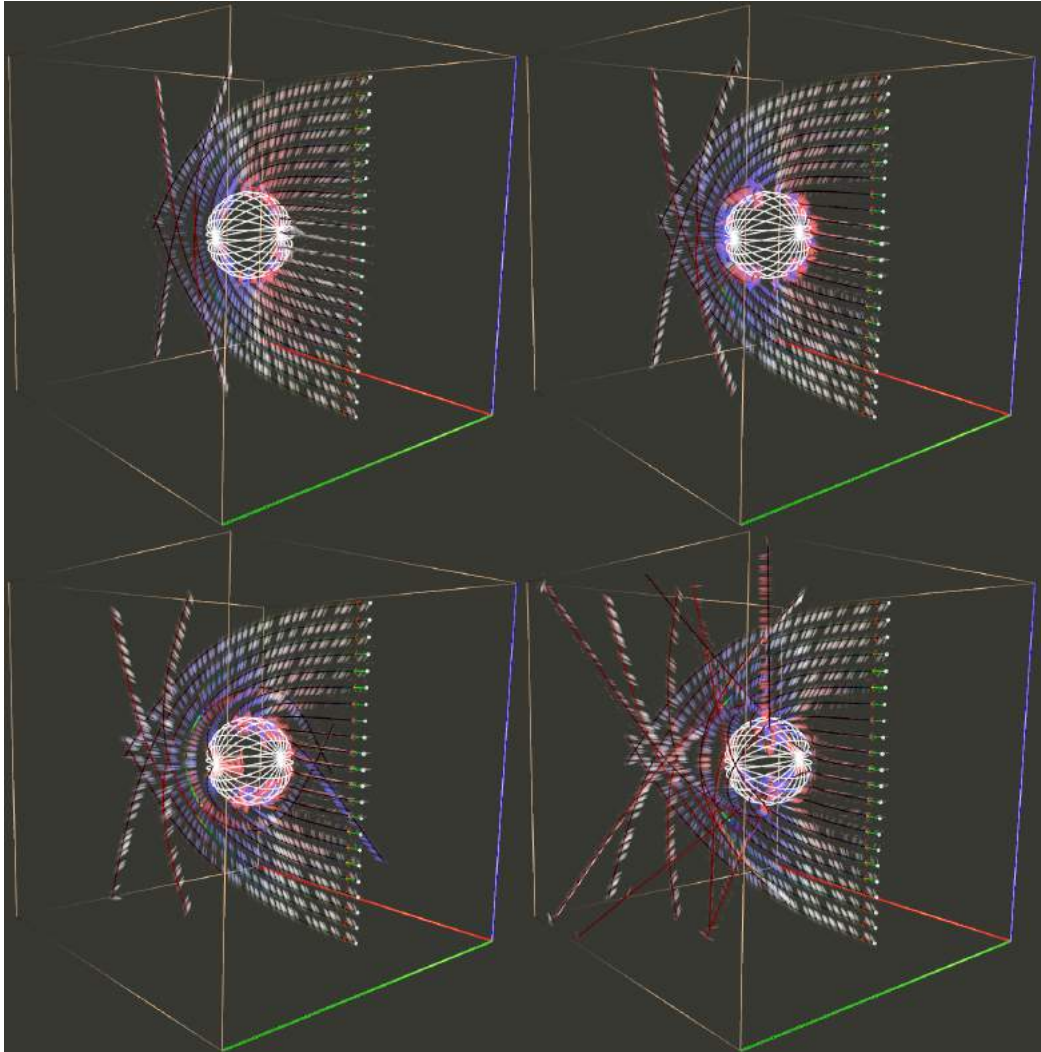
Figure 7.33: Geodesics seeded in positive x direction with $m = 0.2$ and $a = 0.0$, $a = 0.01$, $a = 0.1$ and $a = 0.15$. With introducing the angular momentum the geodesics above and below the event horizon are bent in positive y-direction.
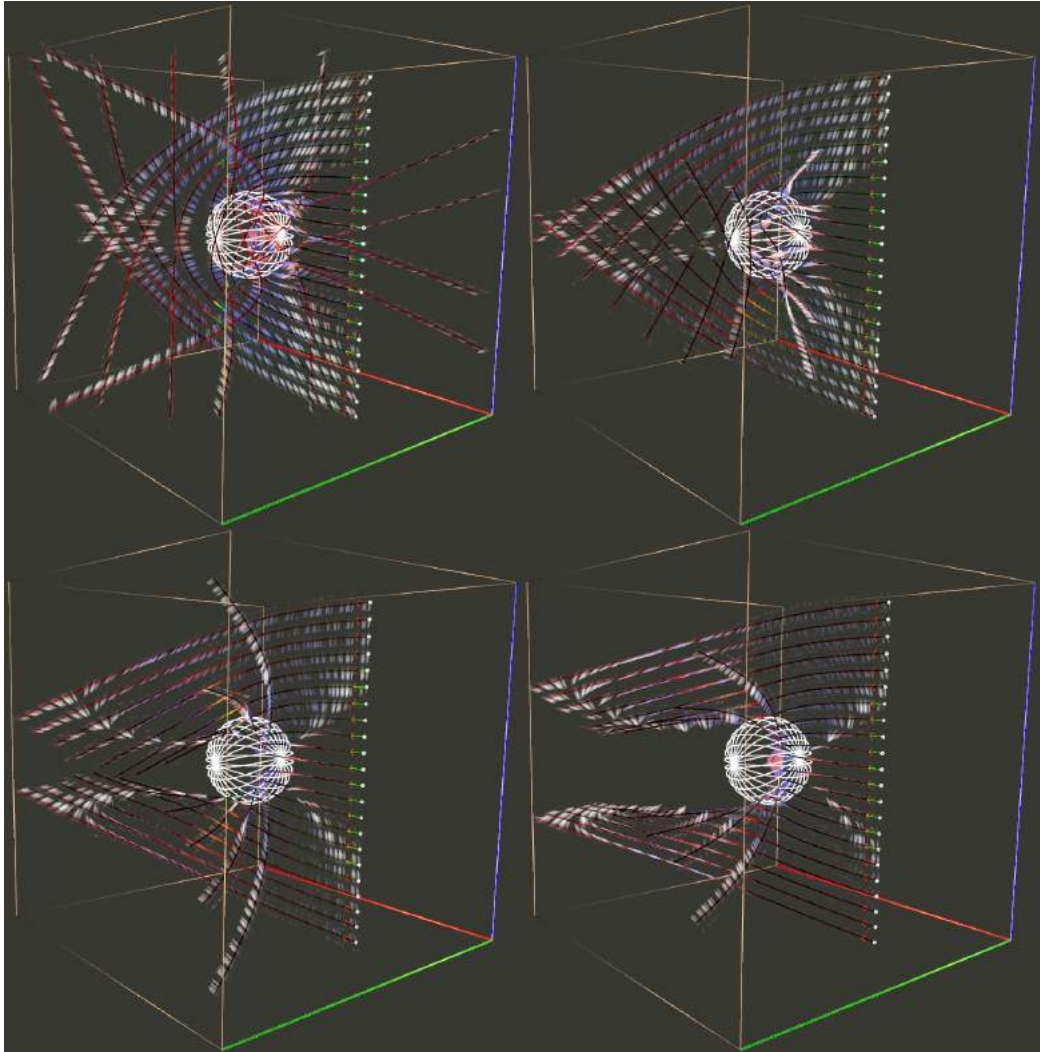
Figure 7.34: Continuation of *figure 7.33*, $a = 0.25$, $a = 0.5$, $a = 0.75$ and $a = 1.0$ (unphysical). In the upper left figure the sixth geodesic from the bottom is torn towards the black hole and bent upwards. When passing the center and is bent outwards, now escaping the heavy mass. In the lower right figure geodesics above and below the event horizon get straightened but are torn in positive y-direction.

A similar setup as in the left lower figure of *figure 7.22* is used to further explore the metric. Again, geodesics are seeded on a circle in the yz-plane, such that they all intersect in one point, if no angular momentum is present and the angular moment is increased. The following five figures show the series from three different axis aligned camera perspectives.

When looking at the spacetime from top, see *figure 7.35* and upper figures of *figure 7.36*, a similar behavior to the 2D analysis can be observed. Geodesics with a greater distance to the xy-plane through the center of gravity are less influenced by the angular momentum. Reflections only occur close to that plane with very strong momentum. Geodesics are symmetric to the xy-plane, which is illustrated in the lower figures of *figure 7.36* and in *figure 7.37*, *figure 7.38* and *figure 7.39*.
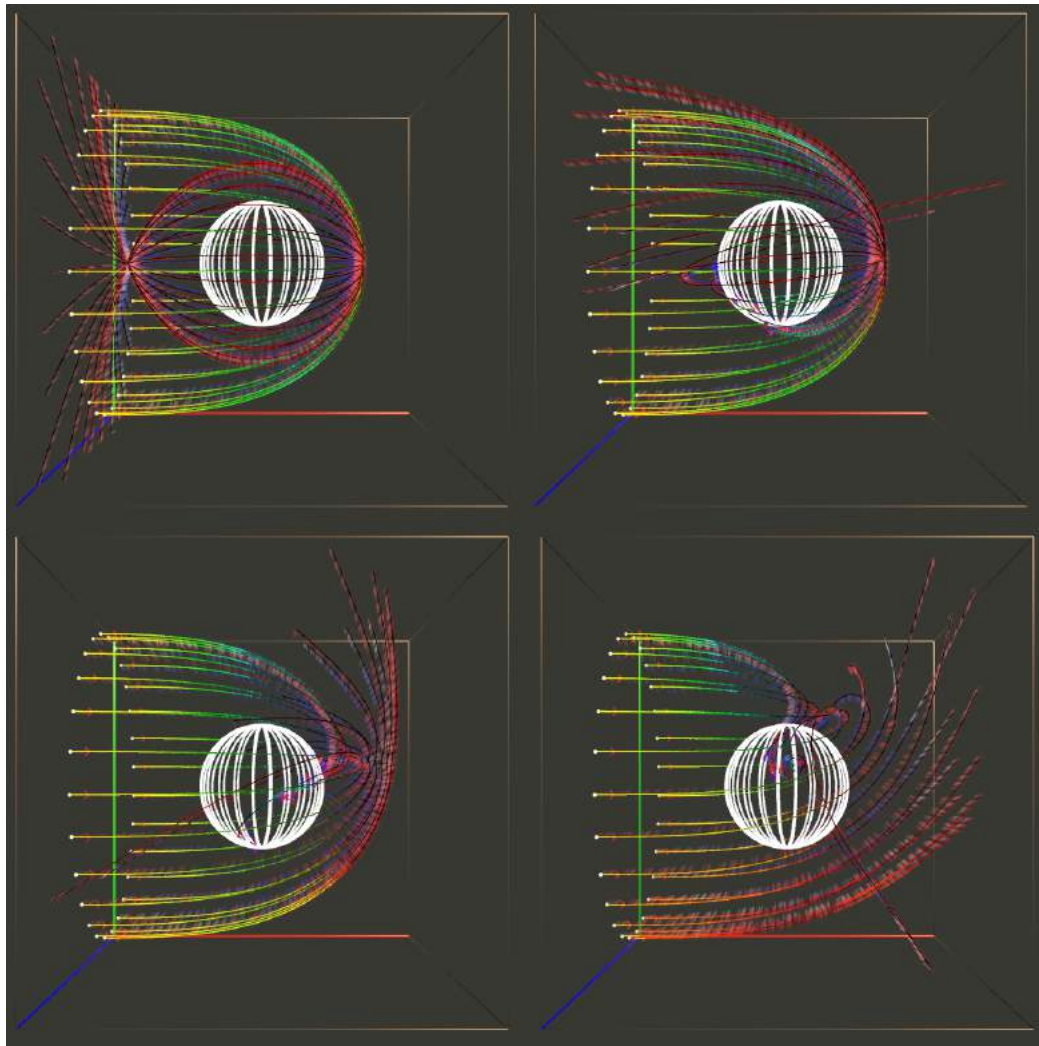
Figure 7.35: Geodesics seeded in positive x-direction on a circle in the yz-plane: $m = 0.2$, $a = 0.0$ (top left), $a = 0.01$ (top right), $a = 0.05$ (bottom left), $a = 0.2$ (bottom right). View in negative z-direction. With $a = 0$ geodesics are almost located on the photon orbit of the Schwarzschild black hole. A slight increase in the angular momentum breaks the symmetry (top right). Geodesics seeded in the upper part of the figure are curved stronger towards the event horizon. Some of them already fall through it. In contrast, the geodesics on the lower part are less attracted towards the center. Further increase of the momentum intensifies this effect. Again, the shrinking of the coordinate singularity can be seen, as geodesics break inside the former Schwarzschild radius (white). *Bottom right:* The geodesic in the mid of the figure is first bent in y-direction and then, when getting closer to the xy-plane strongly coordinate-accelerated toward the center in negative y-direction.
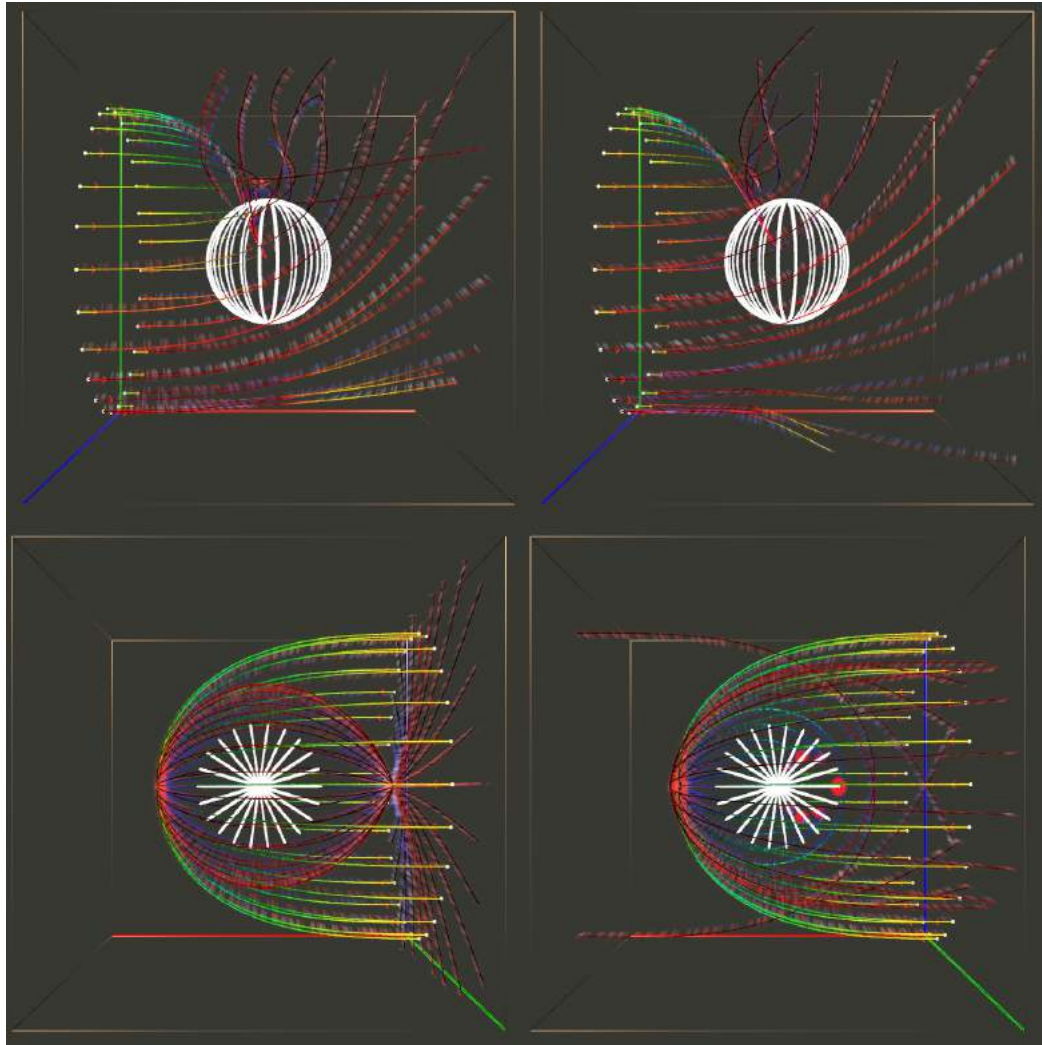
Figure 7.36: *Top:* Continuation of *figure 7.35*: $a = 0.5$ and $a = 1.0$ (unphysical). *Bottom:* View in negative y-direction: $a = 0.0$ and $a = 0.01$. Geodesics are symmetric to the xy-plane through the center of gravity. The symmetry to the xy-plane is clearly illustrated.
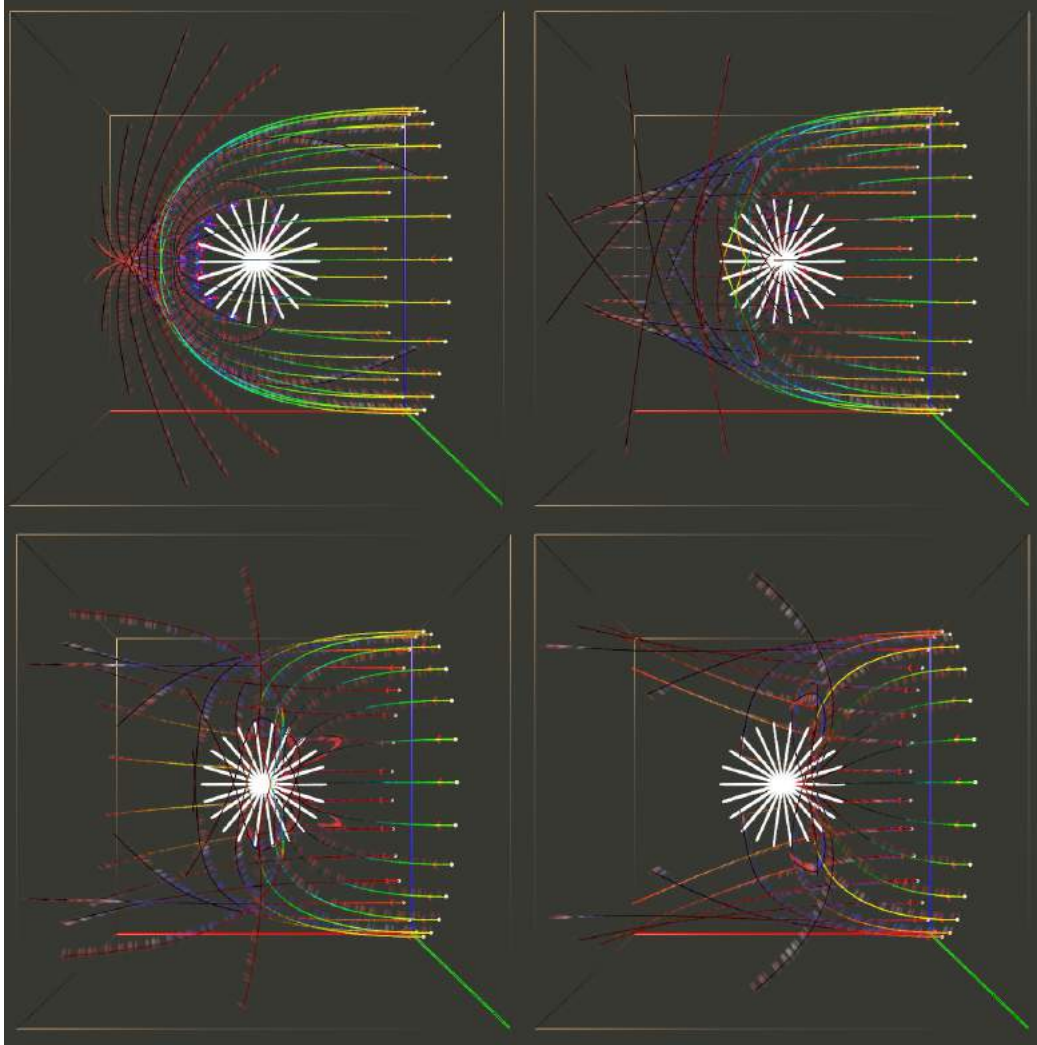
Figure 7.37: Continuation of bottom figures of *figure 7.36*: $a = 0.05$ (top left), $a = 0.2$ (top right), $a = 0.5$ (bottom left) and $a = 1.0$ (bottom right). Geodesics above and below the center are less attracted to the center with increasing angular momentum.
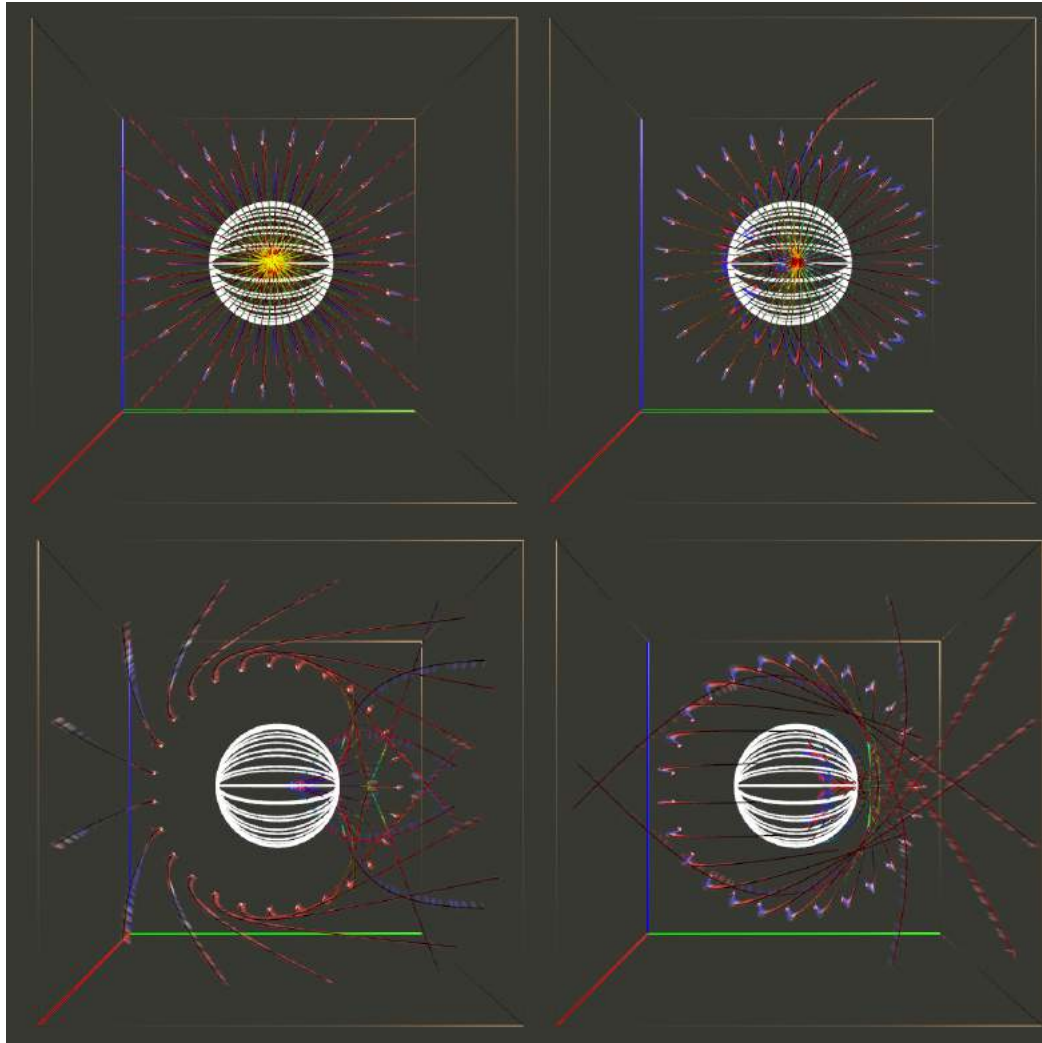
Figure 7.38:   View in negative x-direction: $a = 0.0$ (top left), $a = 0.01$ (top right), $a = 0.05$ (bottom left), $a = 0.2$ (bottom right). Geodesics are symmetric to the xy-plane through the center of gravity.
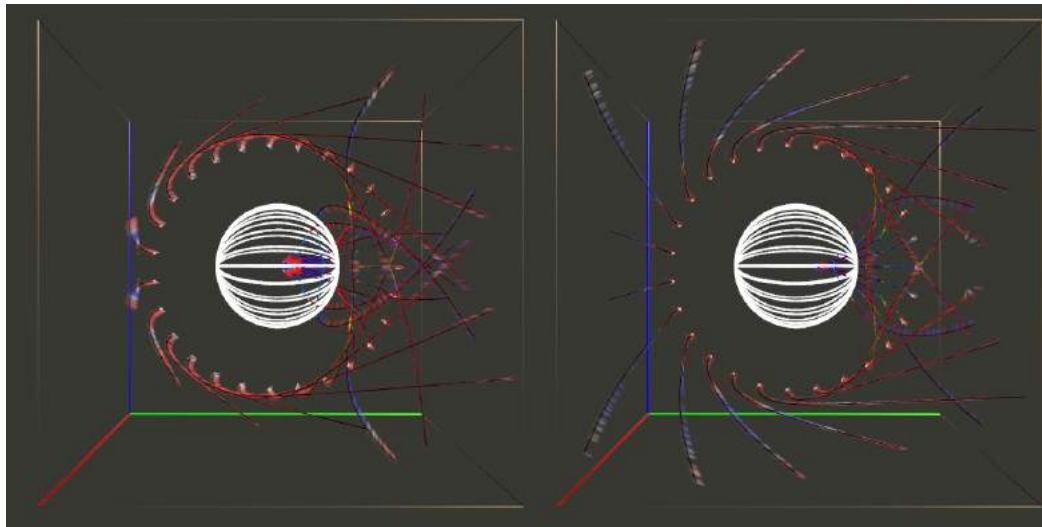
Figure 7.39: Continuation of *figure 7.38*: $a = 0.5$ and $a = 1.0$ (unphysical).

Finally, tubes of geodesics are seeded, similar to *figure 7.25*. The tubes are created using a grid convolution along a diagonal line in the yz-plane. Besides the path and coordinate-acceleration of the geodesics, the deviation of neighboring geodesics is illustrated, too, again visualizing the Riemann tensor.

For example, the lower bundle of geodesics in the bottom left figure of *figure 7.43* gets contracted and bent upwards as it approaches the center of gravity. As it passes by, it is bent downwards again while expanding.

Again, the series of figures illustrates the same mass with increasing angular momentum.

The utilized *Vish* module network is almost identical to the one illustrated in *figure 7.26*. The only difference is the creation module for the metric tensor field. The `KerrMetric44` module is used instead. An additional geometric point distribution used for convolution is added to the seeding modules as well.

The `vis` script that creates the network for the following figures is illustrated in *listing 7.1* and can be found in the *Vish* SVN: `$VISH/data/KerrGeodesics44_Tubes.vis`
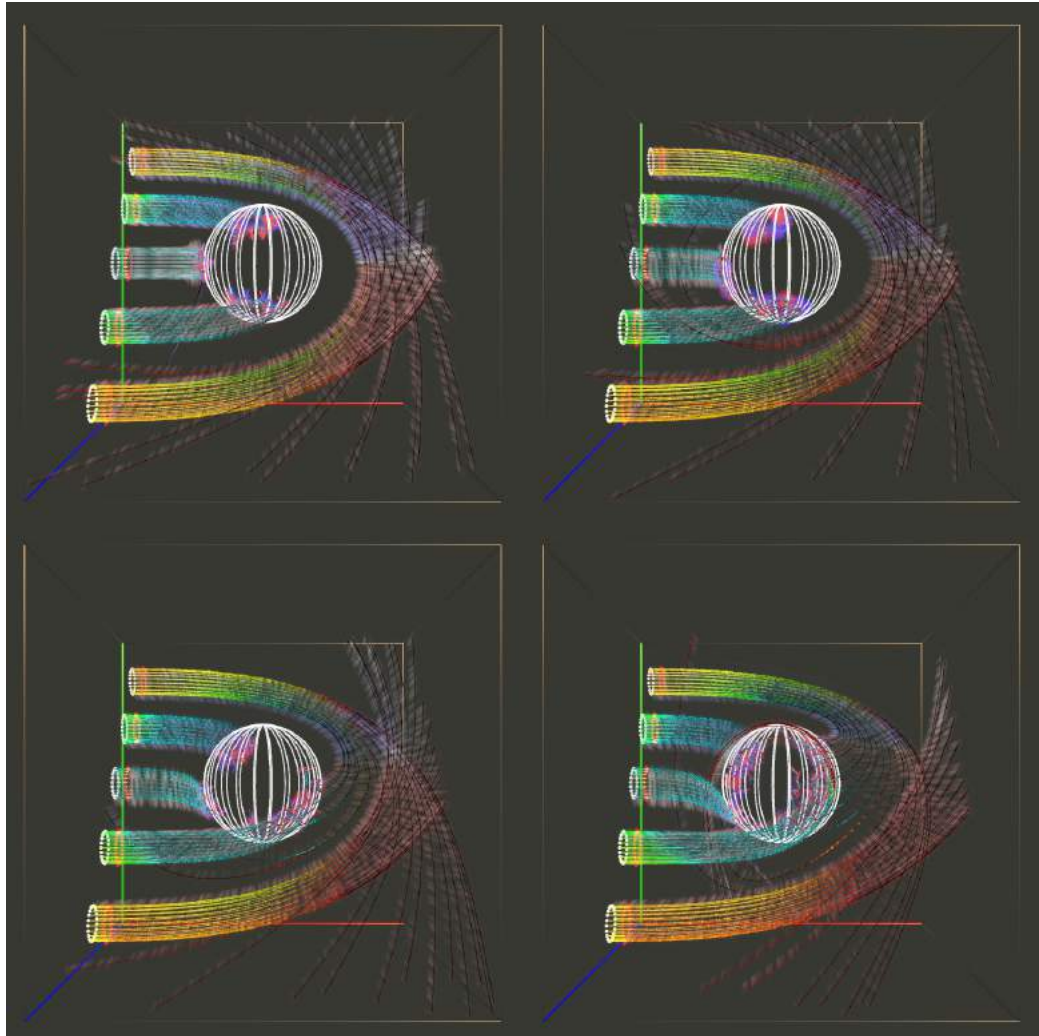
Figure 7.40: Geodesics seeded in positive x-direction on circles convolved by a diagonal line in the yz-plane. View in negative z-direction. The path of the geodesics, the coordinate acceleration in $4D$ (color on line and vector speckles) and the deviation of neighbored geodesics (Riemann tensor) are visualized: $a = 0.0$ (top left), $a = 0.01$ (top right), $a = 0.1$ (bottom left) and $a = m = 0.2$ (bottom right). *Bottom right:* The geodesics of the upper tube of the figure is stronger coordinate-accelerated to the center. The coloring shows that this happens not in space but also in the time coordinate.

Figure 7.41: Over exaggerated and unphysical continuation of *figure 7.40*: $a = 0.25$ (top left), $a = 0.5$ (top right), $a = 0.75$ (bottom left) and $a = 1.0$ (bottom right).

Figure 7.42: Like *figure 7.40* but view in negative y-direction: $a = 0.0$ (top left), $a = 0.01$ (top right), $a = 0.1$ (bottom left) and $a = m = 0.2$ (bottom right). The tubes close to the center axis are contracted, whereas the outer tubes are expanding. Time acceleration is increasing towards the center. Symmetry is lost with increase of the angular momentum.
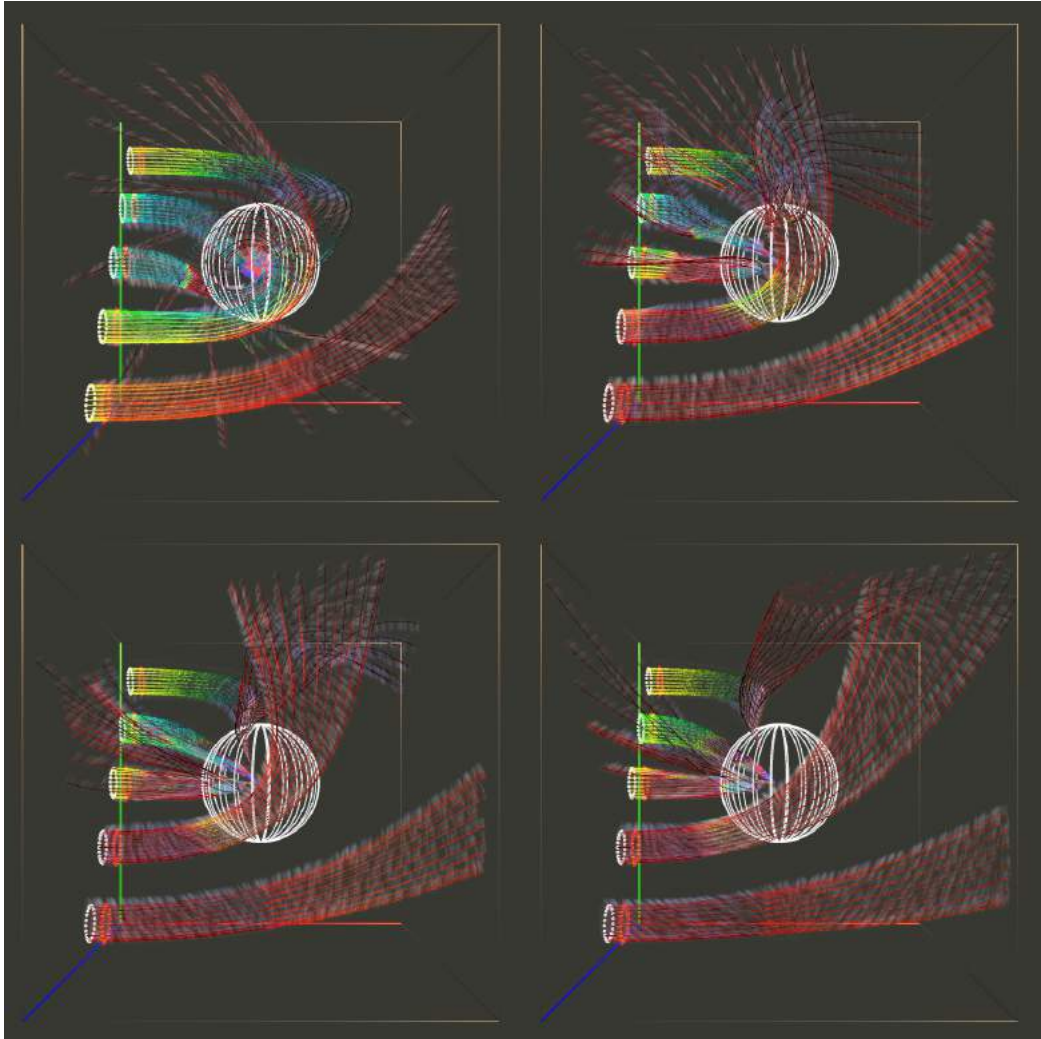
Figure 7.43: Unphysical continuation of *figure 7.42*: $a = 0.25$ (top left), $a = 0.5$ (top right), $a = 0.75$ (bottom left) and $a = 1.0$ (bottom right).
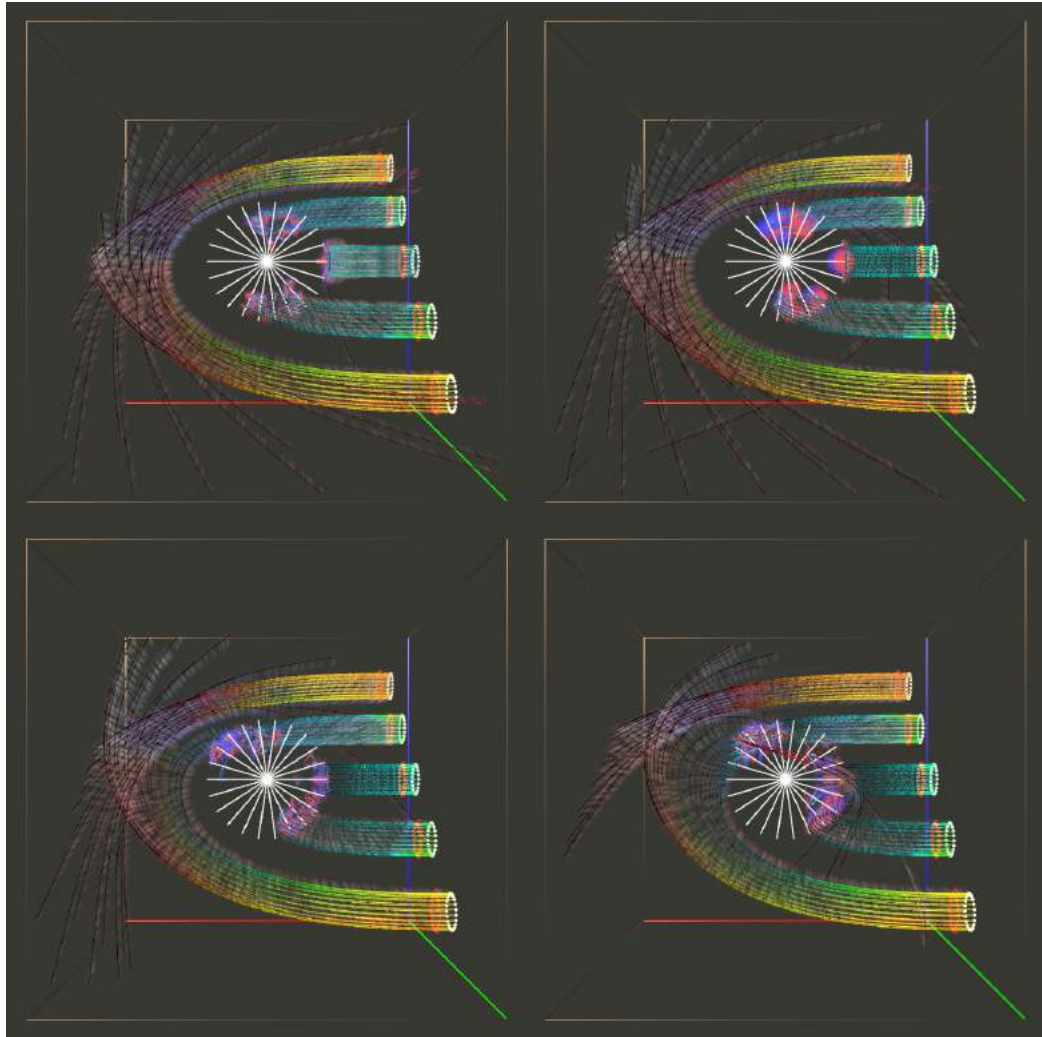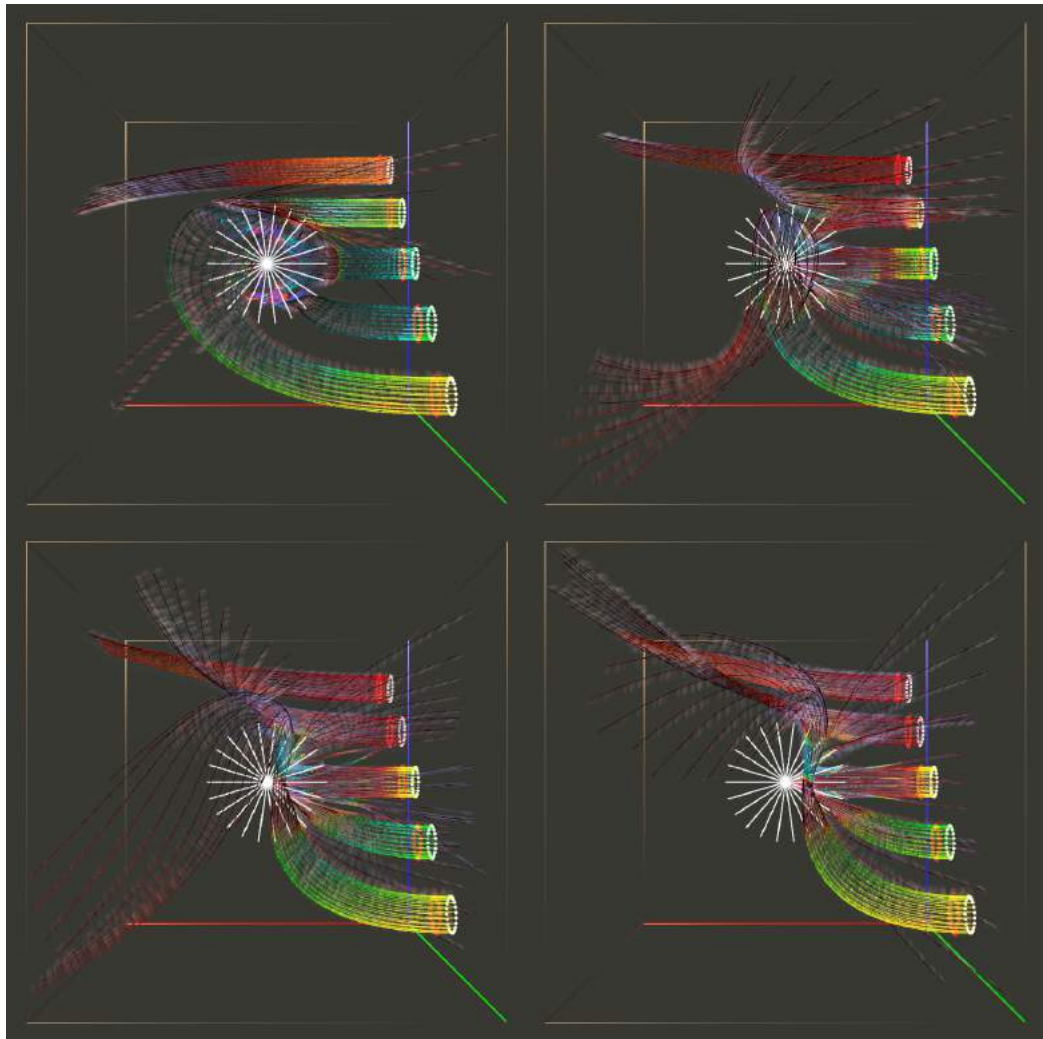
Listing 7.1: The vis script defines the *Vish* module network for the illustrations in *figure 7.40*. The script is located in the SVN: `$VISH/data/KerrGeodesics44_Tubes.vis`

```
 1 CreateBundle/KerrMetric44 blackhole
 2 blackhole.timesteps=10
 3 blackhole.m=0.2
 4 blackhole.a=0.2
 5 blackhole.gridsizeXY=128
 6 blackhole.gridsizeZ=64
 7
 8 Fiber/Grid bh_grid
 9 bh_grid.spacetime=>blackhole
10 <bh_grid>
11
12 Fiber/Tensorfield4D bh_field
13 bh_field.grid=>bh_grid
14 <bh_field>
15
16 Utility/Point3D Point
17 Point.range{Viewer1}=1
18 Point.x{Viewer1}=0.0
19 Point.y{Viewer1}=0.0
20 Point.z{Viewer1}=0.0
21 <Point{Viewer1}>
22
23 Utility/Point3D Point2
24 Point2.range{Viewer1}=1
25 Point2.x{Viewer1}=−1.0
26 Point2.y{Viewer1}=0.0
27 Point2.z{Viewer1}=0.0
28 <Point{Viewer1}>
29
30 Utility/Point3D PointDir
31 PointDir.range{Viewer1}=1
32 PointDir.x{Viewer1}=1
33 PointDir.y=>Point.y
34 PointDir.z=>Point.z
35 <PointDir{Viewer1}>
36
37 Converters/FloatsToRotor Rotate
38 Rotate.yang=90
39
40 Converters/FloatsToRotor Rotate2
41 Rotate2.zang=90
42
43 CreateFiber/PointDistribution GridPoints
44 GridPoints.position=>Point
45 GridPoints.rotation=>Rotate
```

```
46  GridPoints.length=0.2
47  GridPoints.length_subdivs=10
48  GridPoints.height=0.1
49  GridPoints.height_subdivs=10
50  GridPoints.type=Circle
51
52  CreateFiber/PointDistribution  GridPoints2
53  GridPoints2.position=>Point2
54  GridPoints2.rotation=>Rotate2
55  GridPoints2.length=2.0
56  GridPoints2.length_subdivs=5
57  GridPoints2.height=1.8
58  GridPoints2.height_subdivs=20
59  GridPoints2.type=Line
60
61  GridOperations/GridConvolver  conv
62  conv.childgrid=>GridPoints
63  conv.grid=>GridPoints2
64  conv.position=>Point
65  Network.iconify(conv)
66
67  Converters/FloatsToRotor  rot3
68  Network.iconify(rot3)
69
70  Utility/Point3D  scal
71  scal.range=2.0
72  scal.x=1.0
73  scal.y=1.0
74  scal.z=1.0
75  Network.iconify(scal)
76
77  GridOperations/Transform  trans
78  trans.grid=>conv
79  trans.position=>PointDir
80  trans.rotation=>rot3
81  trans.scale=>scal
82  Network.iconify(trans)
83
84  GridOperations/- Sub
85  Sub.grid=>conv
86  Sub.direction=>trans
87  <Sub{Viewer1}>
88  Network.iconify(Sub)
89
90  Display/BoundingBox  BB
91  BB.grid=>bh_grid
92  Network.iconify(BB)
93
94  Display/VectorArrows  Arrs
```

```
 95  Arrs.datafield=>Sub
 96  Arrs.scale=−1.0
 97  Network.iconify(Arrs)
 98
 99  Display/Vertices Verts
100  Verts.grid=>conv
101  Verts.brightness=1.0
102  Verts.scale=−1.0
103  Network.iconify(Verts)
104
105  Display/Dreibein drei
106  drei.grid=>bh_grid
107  drei.thickness=28
108  Network.iconify(drei)
109
110  Compute/Geodesics44 geodesics
111  geodesics.inputfield=>bh_field
112  geodesics.startfield=>Sub
113  geodesics.linelength=100
114  geodesics.stepsize=0.5
115
116  Colormaps/Colorramp ramp
117  ramp.color=0.2
118
119  Display/GlossyColorMapLines lines
120  lines.grid=>geodesics
121  lines.ghostypoints=on
122  lines.pointsize=0.5
123  lines.linewidth=2.0
124  lines.fieldcolormap=>ramp
125  lines.colorlength=off
126
127  Fiber/Vectorfield vf
128  vf=>geodesics
129
130  Compute/ChristoffelQddot44 christi
131  christi.tensorfield=>bh_field
132  christi.qdotfield=>vf
133
134  Operations/DoubleBinary mult
135  mult.operation=mult
136  mult.in=>blackhole.m
137  mult.operator=4.0
138  Network.iconify(mult)
139
140  CreateFiber/PointDistribution HorizonGrid
141  HorizonGrid.length=>mult.out
142  HorizonGrid.type=Sphere
143  HorizonGrid.length_subdivs=500
```

```
144  HorizonGrid.height_subdivs=10
145  Network.iconify(HorizonGrid)
146
147  Display/Vertices HVerts
148  HVerts.grid=>HorizonGrid
149  HVerts.scale=-2.2
150  HVerts.brightness=1.0
151  Network.iconify(Verts)
152  Network.iconify(HVerts)
153
154  Convert/Vector4ToVector3AndScalar vec4Tovec3
155  vec4Tovec3.field=>christi
156  <vec4Tovec3>
157  Network.iconify(vec4Tovec3)
158
159  lines.secondaryfield=>vec4Tovec3
160
161  <blackhole>
162
163  Fiber/Vectorfield vf2
164  vf2=>vec4Tovec3
165  <vf2>
166
167  Colormaps/DopplerMap dpmap
168
169  Display/VectorSpeckles speck
170  speck.colormap=>dpmap
171  speck.vectorfield=>vf2
172  speck.intensity=0.2
173  speck.size=5.0
174  speck.velocityscale=3.0
175
176  Backgrounds/Background back
177  back.blue=0.2
178  back.green=0.22
179  back.red=0.22
180
181
182  Display/GridLegend gl
183  gl.boundingbox=>BB
184  gl.linewidth=0
185  gl.linescale=0
186  Network.iconify(gl)
```
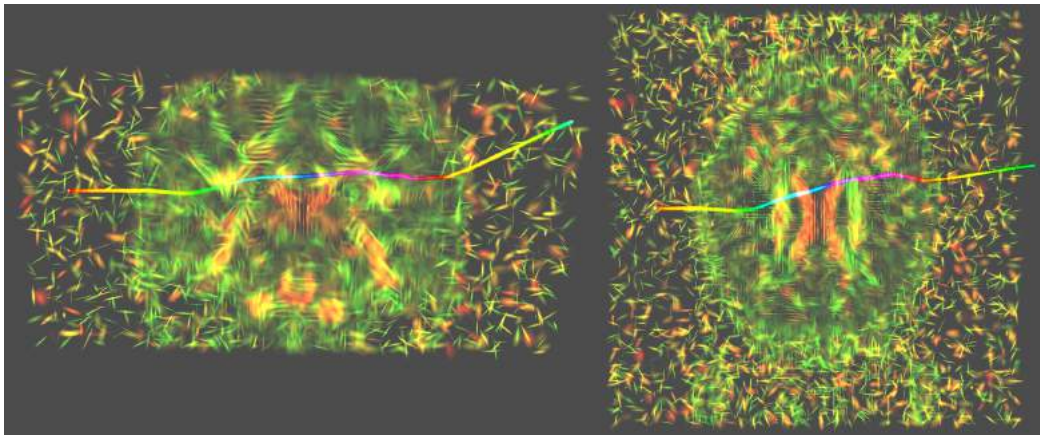
Figure 7.44: One geodesic is passing through a $3D$ diffusion tensor field of a MRI scan of a brain. Properties of the diffusion tensor field are directly visualized using *Vish*s tensor splats. Here, the tensor splats reveal the spatial domain of the brain. When the geodesics approaches the brain tissue it starts to deform and curve depending on the diffusion tensors. Leaving the tissue it becomes a straight line again. *Left:* View in y direction. Tensor splats in xz plane. *Right:* View in z direction. Tensor splats in xy plane.

7.5 Fiber Tracking in MRI Data

Although computing and visualizing geodesics stemmed from general relativity and the analysis of curved space times, there are other scientific domains where such geodesics can be applied.

As described in *section 2.4* data generated from MRI scans can be characterized by tensor fields, see *equation (2.77)*. Thus, Geodesics can be applied to such tensor fields for analysis, as well. Geodesics are then torn toward high diffusion speed directions of the tensor field.

The dataset shown here was provided by Hagen Kitzler (Department of Neuroradiology at Dresden University of Technology) and is a real MRI scan of a human brain infected with a brain tumor. It is hard to differentiate between healthy and tumor brain regions. A direct tensor field visualization method, using so called tensor splates, see [9], can revealed tumor regions. There is a huge interest in good visualization methods.

This section is a proof of concept of developing and implementing visualization algorithms in a flexible and reusable way using the *Fiber Bundle* data model and the *Vish* environment. Developed tools can be used in different contexts and different scientific domains as well.
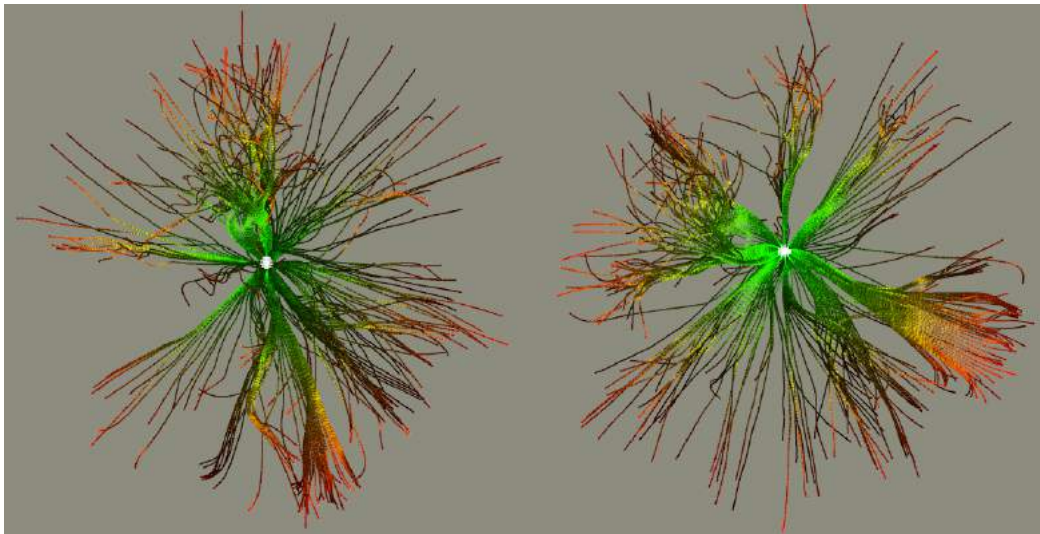
Figure 7.45: Geodesics seeded inside the brain tissue region. They start on a small sphere and are sent outwards. *Left:* View from in z-axis direction. *Right:* View in x-axis direction.

With the tensor field loaded into the *Fiber Bundle* model, the developed spatial geodesics module, *section 6.2.2*, was directly applicable to the diffusion tensor field.

First, the integration yielded just straight lines. An investigation uncovered that the diffusion tensor field is described by very small numbers. In contrast to the metric tensor field in general relativity the tensor field is not normalized in any kind. The diffusion tensor field can be scaled by any value.

Thus, I added a scaling factor to the integral line module, which allows to scale $\ddot{q}$ during integration. This is equivalent to multiplying the tensors of the diffusion field by a scalar factor. For the illustrations shown here a scaling factor of about 200000 was used to reveal some curvature.

Figure 7.44 illustrates one geodesic traveling through the brain tissue. Where diffusion occurs, it deforms and curves along its way attracted into directions of high diffusion speed. When sending a bundle of geodesics seeded closely together, they end up traveling in many different directions, as illustrated in *figure 7.46*. *Figure 7.45* shows two different views of geodesics seeded in the brain tissue and flowing outwards.

However, the question of whether geodesics can be an appropriate tool for brain analysis or even tumor localization goes beyond the scope of this thesis and could be the topic of further investigation. Coloring or driving the transparency of visualized geodesics might highlight regions of low curvature
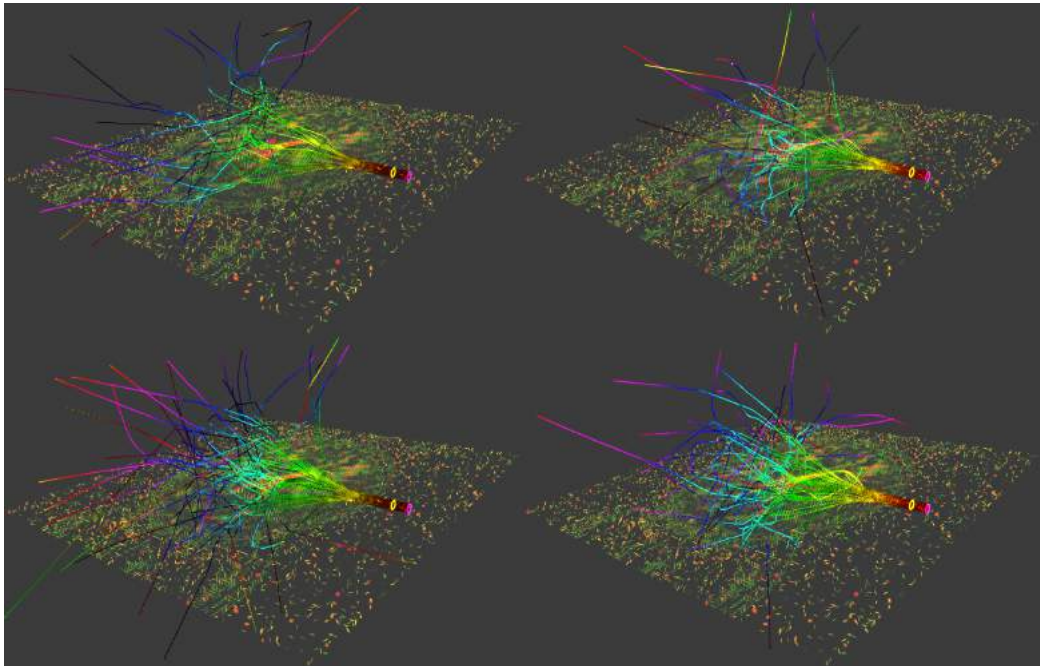
Figure 7.46: A bundle of geodesics passing through the MRI brain scan. Geodesics are seeded on the magenta circle in direction of the yellow circle. *Left:* Geodesics integrated using Euler. *Right:* Geodesics integrated using Dop853. *Top:* 22 seeding points on the circle. *Bottom:* 52 seeding points on the circle.

that could be an indication for tumor regions. Different seeding strategies can be interesting to investigate as well. Seeding in the direction of the maximal eigenvector could be explored.
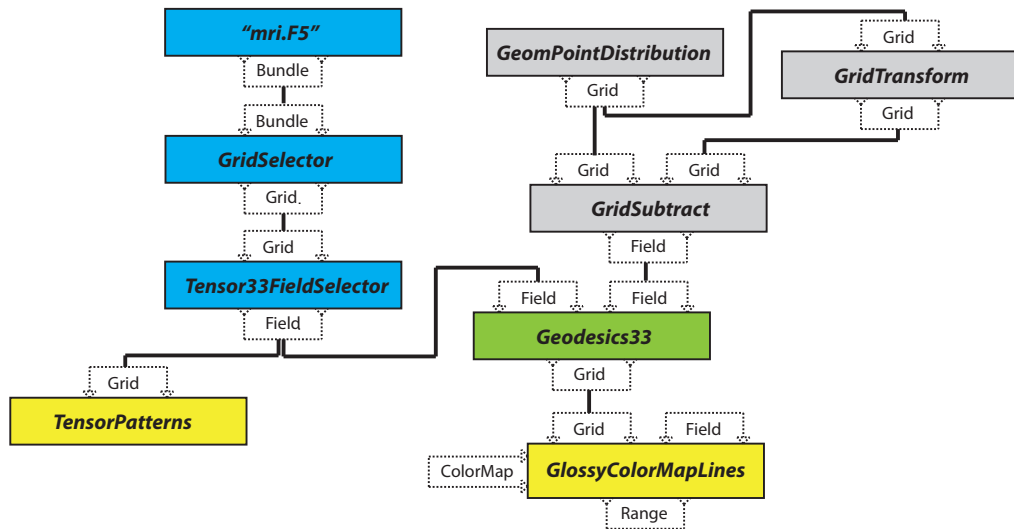
Figure 7.47: Schematic *Vish* network used to create the visualizations shown in *figure 7.44* and *figure 7.46*. The tensor field data is extracted from the module that load the *F5* file, by first selecting a grid object and then a tensor field (cyan). The seeding geometry is created using a `GeometricPointDistribution` which is copied and transformed to compute a direction vector field (grey). The `StreamlineIntegrator` does the computation and stores a grid of lines back into the bundle hosting the tensor field and provides a grid handle for the rendering module `GlossyColorMapLines`. The tensor splats are rendered using the `TensorPatterns` rendering module connected to the diffusion tensor field.

Chapter 8

Future Work

The thesis opened possibilities for future work and extensions in several directions. For example:

- Add AMR support to the `FindLocalFromWorldPoint` class.

- Introduce a geodesic module that takes time evolution of the underlying grid into account.

- Implement other types of integration lines, such as gradient lines in scalar fields or other types of CFD integration lines.

- Enhance the grid convolution by aligning the copied grid by a given direction. Then, for example, circles could be copied along a line and used for seeding again. Contraction and Expansion along a line could then be visualized by computing short integral lines of the copied geometry.

- Introduce the `GridEvaluator` to re-sample any grid based data on a different grid, such as sampling a pressure field on a stream line grid or on a density iso-surface.

- Provide a convenient data export functionality of *Fiber Bundles* to *Vish*. Then any *Fiber* data, such as a pressure field sampled stream line can be exported to *F5*. Other independent programs could then be used to perform CFD calculations or visualizations on the streamlines.

- The rendering module can be enhanced. For example, adding transparency could help to visually extract information. Curvature measures mapped to visual parameters could enhance illustrations. Extrusion of geometric shapes along lines could help to add even more information to the displayed lines.

- More work can be spent of the frameworks to enhance usability and reduce source code in the application modules. More convenience functions could be added to operate with the *Fiber Bundle* data library.

- More data exploration of MRI data and combination of visualization techniques could possibly reveal techniques for better brain tumor detection.

Chapter 9

Conclusion

The framework I implemented allow to visualize streamlines in vector fields and geodesics in 3D spatial or 4D spacetime metrics. Underlying data may be given on uniform or multi-block grids. Coarse and highly accurate integration schemes are provided, and interactive 3D line visualizations cached on the *GPU* can be used for exploration. Functionality can easily be extended.

The developed techniques have been verified on simple test case scenarios (Couette Flow and Schwarzschild metric). They were used to explore the flow of fluids in a stirred tank and the curvature of spacetime of the rotating black hole (kerr metric). The rudimentary geodesic visualization was enhanced by colors and vector speckles [*section 7.3*] to depict the four-dimensional coordinate-acceleration along the integration lines. Any metric data, such as data stemming from numerical relativity, can be loaded when provided in the correct *F5* format [8].

The generally greater amount of work spent on re-usable and flexible designs and data models was rewarded by opening unexpected possibilities. The major achievements of this thesis are:

- The 3D visualization framework I implemented as part of this thesis makes investigation of data stemming from numerical relativity or computational fluid dynamics more accessible to both application scientists and scientific visualization programmers.

- My advances in techniques for operations on *Fiber Grids* make definitions of several seeding geometry possible. In particular, the invention of the *grid convolution* has yield a powerful generic tool for geometry creation, as described in [1].

- The highly modular implementation approach allow enabled seeding streamlines originating from an iso-surface and visualizing the

coordinate-acceleration along geodesics.

- Finding a common template interface for general line integral computation has reduced the actual code for geodesic computation tremendously. Other types of integral lines can be added with a minimum of source code development (with probably less than 200 lines of source code).

- Separating the handling of the underlying numerical discretization grids from the computational algorithms via the `LocalFromWorldPoint` class allows to formulate algorithms independent of the grid structure. My work provided the essential basis for Bidur Bohara and Nathan Brener at the computer science department of the Louisiana State University, [16]. Moreover, this method allows to easily re-sample on any grid, given data on another grid.

- I reviewed and clarified the utilized research software environment by creating a tutorial and complementing documentation. This was important for developing the concepts of the framework. Together with the detailed descriptions throughout the thesis. This provides a good getting-started-documentation for other researchers.

- The utilization of the framework is demonstrated. Several applications illustrate visualization possibilities on test and research scenarios.

- Images and videos of the scientific visualizations can be created in very high image quality. Thus they are applicable not only for research but also for to public outreach.

Chapter 10

Fazit

Mit dem von mir entwickelten Softwareframework können Stromlinien in Vektorfeldern und Geodäten in drei- oder vier dimensionalen Raumzeiten visualisiert werden. Die numerischen Daten können auf uniformen oder kurvilinearen Multi-Block-Gittern gegeben sein. Ein grobes und ein hochpräzises Integrationsverfahren wird für die Integration bereitgestellt. Die interaktive 3D Darstellung wird auf der Graphikhardware gecached.

Die Verfahren wurden an zwei einfachen Szenarien getestet und verifiziert (Couette-Strömung und Schwarzschildmetrik). Es wurden dann die Strömung in einem Mischtank und die Raumzeitkrümmung eines rotierenden schwarzen Loches (Kerr-Metrik) visualisiert. Die Darstellung der Geodäten wurde durch Einfärben und Beifügen von Vektor-Sprenkeln erweitert und damit die zugehörige vier-dimensionale Koordinatenbeschleunigung visualisiert. Jedes metrische Tensorfeld, das im korrekten Datenformat (F5) bereitgestellt wird, siehe [8], kann so visualisiert werden, also im besonderen numerisch relativistische Simulationsdaten.

Der Mehraufwand, investiert in Konzepte, wiederverwendbare Designs und Datenmodelle, wurde mit unvorhergesehener Funktionalität belohnt. Die Haupterrungenschaften der Arbeit sind:

- Die in dieser Arbeit entwickelten Visualisierungs-Tools ermöglichen die Untersuchung von Daten und damit verbundenen Fragestellungen aus Simulationen der numerischen Relativitätstheorie und der Strömungsmechanik. Die entwickelten Softwarekomponenten können einfach erweitert werden. Davon profitieren Wissenschaftler sowohl als Anwender als auch als Visualisierungs-Programmierer.

- Mit den hervorgegangenen Operationen auf *Fiber Grid*s kann eine Vielfalt an Startpunktgeometrien erzeugt werden. Besonders mit

meiner Erfindung der "Gridfaltung" können komplexe Gebilde einfach erzeugt werden, siehe [1] oder Anhang C.

- Visualisierungstechniken, wie zum Beispiel eine Isofläche als Startgeometrie für Stromlinien vorzusehen oder Geodäten mit Vektor-Sprenkel zu auszurüsten, sind durch Verwendung sehr modularer Implementierungsansätze entstanden.

- Die Entwicklung eines gemeinsamen C++ Template Interfaces für die Berechnung von Integrallinien ermöglichte eine sehr kompakte Implementierung der Geodätenberechnung. Weitere Integrallinien können nun ebenfalls mit minimaler Quellcodelänge implementiert werden. (Mit wohl weniger als 200 Zeilen Quellcode.)

- Die Abstraktion der Berechnungsalgorithmen von den notwendigen Operationen zur Handhabung der diskreten Gitter wurde durch die Klasse `LocalFromWorldPoint`[1] erreicht. Damit können Algorithmen nun unabhängig vom Diskretisierungsschema implementiert werden. Meine Arbeit ermöglichte jene von Bidur Bohara und Nathan Brener am Institut für Informatik an der Louisiana State University, siehe [16] oder Anhang C. Es ermöglicht eine einfache Umrechnung von Daten, die auf einem Gittertyp gegeben sind, auf einen anderen Gittertyp.

- Die kritische Begutachtung und Analyse der Softwareumgebungen, die in der Forschung entstanden sind, brachte Erkenntnisse, die für die Entwicklung meiner Framework Konzepte wichtig waren. Ich erstellte ein Tutorial und ergänzte die Dokumenation. Zusammen mit den ausführlichen Beschreibungen in dieser Arbeit bietet dies einen gute Startbasis für jeden Wissenschaftler, der die Softwareumgebungen verwenden möchte.

- Die Anwendung des Frameworks wurde demonstriert. Die Visualisierungsmöglichkeiten wurden in Testszenarien und einem Forschungsszenario, [Kapitel 7.2], vorgestellt.

- Bilder der wissenschaftlichen Visualisierungen wurden in sehr guter Qualität erstellt und können somit auch für Öffentlichkeitsarbeit verwendet werden.

---

[1] "LokalerVonWeltPunkt"

Bibliography

[1] Werner Benger; Marcel Ritter; Somnath Roy; Sumanta Acharya and Feng Jijao. Fiberbundle-Based Visualization of a Stirred-Tank Flow. In *WSCG' 2009 Communication Papers Proceedings*, 2009.

[2] Keith Andrew and Charles G. Fleming. Space-time geometries characterized by solutions to the geodesic equations. *Computers in Physics*, 6(5):498–505, Sep/Oct 1992.

[3] Autodesk. Maya. http://usa.autodesk.com, 2010.

[4] Werner Benger. Simulation of a black hole by raytracing. In R.A. Puntigam F.W. Hehl and H. Ruder, editors, *Relativity and Scientific Computing - Computer Algebra, Numerics, Visualization*, pages 2–3, Berlin Heidelberg New York, 1996. Springer Verlag. `http://www.photon.at/~werner/bh/`.

[5] Werner Benger. Voids: Der Einfluss der kosmologischen Konstanten auf die Vakuumblasen im expandierenden Universum. Master's thesis, Leopold Franzens Universität Innsbruck, 1997.

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

[7] Werner Benger. Light++ license. http://svn.origo.ethz.ch/wsvn/f5/doc/copyright.html, 2010.

[8] Werner Benger. The Fiber Bundle HDF5 Library. http://www.fiberbundle.net, 2010.

[9] Werner Benger, Hauke Bartsch, H.-C. Hege, H. Kitzler, A. Shumilina, and A. Werner. Visualizing Neuronal Structures in the Human Brain via Diffusion Tensor MRI. *International Journal of Neuroscience*, 116(4):pp. 461–514, 2006.

[10] Werner Benger, Andrew Hamilton, Mike Folk, Quincey Koziol, Simon Su Princeton, Erik Schnetter, Marcel Ritter, and Georg Ritter. Using geometric algebra for navigation in riemannian and hard disc space. In Vaclav Scala and Dietmar Hildenbrand, editors, *GraVisMa 2009 - Computer Graphics, Vision and Mathematics for Scientific Computing*, 2009. submitted.

[11] Werner Benger, Georg Ritter, and René Heinzl. The Concepts of VISH. In 4th *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007*, pages 26–39. Berlin, Lehmanns Media-LOB.de, 2007.

[12] Werner Benger, Georg Ritter, Marcel Ritter, and Wolfram Schoor. Beyond the visualization pipeline. In 5th *High-End Visualization Workshop, Baton Rouge, Louisiana, March 18th - 21st, 2009*. Berlin, Lehmanns Media-LOB.de, 2009.

[13] Werner Benger, Georg Ritter, Simon Su, Dimitris E. Nikitopoulos, Eamonn Walker, Sumanta Acharya, Somnath Roy, Farid Harhad, and Wolfgang Kapferer. Doppler speckles - a multi-purpose vectorfield visualization technique for arbitrary meshes. In *CGVR'09 - The 2009 International Conference on Computer Graphics and Virtual Reality*, 2009.

[14] Werner Benger, Shalini Venkataraman, Amanda Long, Gabrielle Allen, Stephen David Beck, Maciej Brodowicz, Jon MacLaren, and Edward Seidel. Visualizing katrina - merging computer simulations with observations. In *Workshop on state-of-the-art in scientific and parallel computing, Umeå, Sweden, June 18-21, 2006*, pages 340–350. Lecture Notes in Computer Science (LNCS), Springer Verlag, 2006.

[15] Paul A Blaga and Cristina Blaga. Bounded radial geodesics around a kerr-sen black hole. *Classical and Quantum Gravity*, 18(18):3893, 2001.

[16] Bidur Bohara, Farid Harhad, Werner Benger, Nathan Brener, Sitharama Iyengar, Marcel Ritter, Kexi Liu, Brygg Ullmer, Nikhil Shetty, Vignesh Natesan, Carolina Cruz-Neira, Sumanta Acharya, Somnath Roy, and Bijaya Karki. Evolving time surfaces in a virtual stirred tank. In Vaclav Skala, editor, *18th International Conference on Computer Graphics, Visualization and Computer Vision'2010*, 2010.

[17] Steve Bryson. Virtual Spacetime: An Environment for the Visualization of Curved Spacetimes via Geodesics Flows. Technical Report RNR-92-009, NASA, Mar 1992.

[18] David M. Butler and Steve Bryson. Vector-Bundle Classes Form Powerful Tool for Scientific Visualization. *Computers in Physics*, 6(6):576–584, Nov/Dec 1992.

[19] David M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, sep/oct 1989.

[20] B. Carter. Axisymmetric black hole has only two degrees of freedom. *Phys. Rev. Lett.*, 26(6):331–333, Feb 1971.

[21] CCT. Sciviz. http://sciviz.cct.lsu.edu/projects/vish, 2010.

[22] Chad Clark, William A. Hiscock, and Shane L. Larson. Null geodesics in the Alcubierre warp drive spacetime: the view from the bridge. *Gerneral Relativity and Quantum Cosmology*, (gr-qc/9907019), July 1999.

[23] Chris Doran and Anthony Lasenby. *Geometric Algebra for Physicists*. Springer-Verlag London, 2009.

[24] EGO. European gravitational observatory. http://www.ego-gw.it, 2010.

[25] Albert Einstein. Die Grundlage der allgemeinen Relativittstheorie . *Annalen der Physik*, 354:762–822, 1916.

[26] George F R Ellis and Henk Van Elst. Deviation of geodesics in flrw spacetime geometries, 1997.

[27] Eyeon. Fusion6. http://www.eyeonline.com/Web/EyeonWeb, 2010.

[28] Oliver Fechtig. Physikalische Aspekte und Visualisierung von stationären Wurmlöchern. Master's thesis, Institut für Theoretische Physik I, Universität Stuttgart, 2004.

[29] Max Plank Institute for Gravitational Physics. Geo600. the german-british gravitational wave detector. http://www.geo600.org, 2010.

[30] Inc. Free Software Foundation. The gnu operating system. http://www.gnu.org, 2010.

[31] Geofrey Furnish. Container-Free Numerical Algorithms in C++. *Computers in Physics*, 12(3), May/June 1998.

[32] Khronos Group. Opengl. http://www.opengl.org, 2010.

[33] The HDF Group. Hdf. http://www.hdfgroup.org/HDF5, 2010.

[34] Ernst Hairer, Syvert Paul Norsett, and Gerhard Wanner. *Solving Differential Equations I.* Springer-Verlag Berlin Heidelberg, 2000.

[35] Andrew Hamilton. Black Hole Flight Simulator. http://casa.colorado.edu/ ajsh/bhfs/screenshots, 2010.

[36] James B. Hartle. *Gravity: An Introduction to Einstein's General Relativity.* Addison Wesley, 2003.

[37] Computational Engineering International. Ensight. http://www.ensight.com, 2010.

[38] Harald J W Mller Kirsten. *Classical Mechanics and Relativity.* World Scientific, 2008.

[39] LIGO. The laser interferometer gravitational-wave observatory (ligo). http://www.ligo.caltech.edu, 2010.

[40] Horst Lippmann. *Angewandte Tensorrechnung.* Springer Verlag, 1993.

[41] Herbert Mang and Gnther Hofstetter. *Festigkeitslehre.* Springer Verlag, second edition, 2004.

[42] Daisuke Miyazaki. Inverse matrix of 2-by-2 matrix, 3-by-3 matrix, 4-by-4 matrix. http://www.cvl.iis.u-tokyo.ac.jp/ miyazaki/tech/teche23.html, 2010.

[43] NCSA. Movies from the edge of spacetime. http://archive.ncsa.illinois.edu/Cyberia/NumRel/MoviesEdge.html, 2010.

[44] Nobelprize.org. The nobel prize in physics 1993. http://nobelprize.org/nobel_prizes/physics/laureates/1993/press.html, 2010.

[45] Nokia. Qt. http://qt.nokia.com, 2010.

[46] OpenDX.org. Opendx. http://www.opendx.org, 2010.

[47] Peter J. Pahl and Rudolf Damrath. *Ingeniuerinformatik.* Springer Verlag, 2000.

[48] Dave Partyka. [vtk-developers] stl in vtk header files. http://www.vtk.org/pipermail/vtk-developers/2009-June/006082.html, 2010.

[49] Marcel Ritter. Introduction to HDF5 and F5. Technical Report CCT-TR-2009-13, Center for Computation and Technology, Louisiana State University, 2009.

[50] Randi Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Mala, and Mike Weiblen. *OpenGL Shading Language*. Addison-Wesley Longman, Amsterdam, 3rd edition, 2009.

[51] Dave Schreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Longman, Amsterdam, 7th edition, 2009.

[52] SGI. Standard template library programmer's guide. http://www.sgi.com/tech/stl, 2010.

[53] Vaclav Skala. International conferences in central europe on computer graphics, visualization and computer vision. http://wscg.zcu.cz, 2010.

[54] Detlev Stalling. *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Konrad-Zuse-Zentrum für Informationstechnik(ZIB), und Freie Universität Berlin, Fachbereich Mathematik und Informatik, 1998.

[55] Norbert Straumann. *Complete Maya Programming. An Extensive Guide to Mel and C++ Api*. Morgan Kaufmann, 2003.

[56] Inc. Tecplot. Tecplot. http://www.tecplot.com, 2010.

[57] Dimitri van Heesch. Doxygen. http://www.doxygen.org, 2010.

[58] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates - The Complete Guide*. Addison Wesley, 2003.

[59] Todd L. Veldhuizen. Using C++ template metaprograms. 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[60] Todd L. Veldhuizen. C++ templates are turing complete. Technical report, Indiana University, 2003.

[61] John Vince. *Geometric Algebra: An Algebraic System for Computer Games and Animation*. Cambridge University Press, 2007.

[62] VSG. Avizo. http://www.vsg3d.com/vsg_prod_avizo_overview.php, 2010.

[63] Corvin Zahn. Vierdimensionales Raytracing in einer gekrmmten Raumzeit. Master's thesis, Universität Stuttgart, 1991.

[64] ETH Zürich. Origo. http://vish.origo.ethz, 2010.

Appendix A

Acknowledgements

Appendix B

Definition of Fiber Bundles

Here follows the missing mathematical definition of fiber bundles. As the concept of fiber bundles was more an inspiration than a strict mathematical basis for developing the *Fiber Bundle* library and the *F5* file format I decided to move the according definitions to the appendix.

Definitions are taken from [49] and [6], where more in depth information can be found.

Definition 5:
A subset $A \subseteq X$ of a topological space (X, τ) is a **neighborhood** of an element of $p \in X$ iff it contains an element O of τ that contains p:
$A \subseteq X \upsilon(p) \Leftrightarrow \exists O \in \tau : p \in O, O \subseteq A$

Definition 10:
The **cartesian product** $X \times Y$ of **two topological spaces** X, Y with the respective neighborhood sets $\upsilon(x) \subset \mathcal{P}(X), \nu y \subset \mathcal{P}(Y)$ of the points $x \in \mathcal{P}$, $y \in \mathcal{P}$ is a topological **space**, if the neighborhood sets $\upsilon(x, y)$ of the point $(x, y) \in X \times Y$ are given by $\upsilon(x, y) = \{U \in \upsilon(x), V \in \nu(y) : U \times V \subset W : W\}$.

Definition 50:
Let E and B being topological spaces and $f : E \to B$ be a continuous map. (E, B, f) is called a **fiber bundle**, if there exists a space F, such that the union of the inverse image of f of a neighborhood $U_b \subset B$ of each point $b \in B$ is homeomorph to $U_b \times F$:
(E, B, f) **fiber bundle** $\Leftrightarrow \exists F : \forall b \in B : \exists U_b : f^{-1}(U_b) \simeq U_b \times F$ "[6]

Appendix C

Related Publications

During the work on the thesis I was involved in four related publications: [1], [12], [10] and [16]. I participated at the WSCG (Winter School of Computer Graphics, [53]) in Plzen, Czech Republic in 2009 and 2010 and presented the communication paper [1] during the conference.
The publications are included here in the appendix.

Fiberbundle-Based Visualization of a Stirred-Tank Flow

Werner Benger

Feng Jijao

Center for Computation &
Technology
at Louisiana State University
239 Johnston Hall
USA, LA 70803, Baton Rouge

werner@cct.lsu.edu

fjiao@cct.lsu.edu

Marcel Ritter

Center for Computation &
Technology at LSU

and

Department of Computer Science

University of Innsbruck
Technikerstrasse 21a
Austria, A-6020 Innsbruck

marcel@cct.lsu.edu

Sumanta Acharya

Somnath Roy

Louisiana State University
Department of Mechanical
Engineering
USA, LA 70803, Baton Rouge

acharya @me.lsu.edu
sroy13@lsu.edu

ABSTRACT

We describe a novel approach to treat data from a complex numerical simulation in a unified environment using a generic data model for scientific visualization. The model is constructed out of building blocks in a hierarchical scheme of seven levels, out of which only three are exposed to the end-user. This generic scheme allows for a wide variety of input formats, and results in powerful capabilities to connect data. We review the theory of this data model, implementation aspects in our visualization environment, and its application to computational fluid dynamic simulation of flow in an impeller-stirred tank. The computational data are given as a velocity vector field and a scalar pressure field on a mesh consisting of 2088 blocks in curvilinear coordinates.

Keywords
Computational fluid dynamics, data model, stream lines, scientific visualization.

1 INTRODUCTION

Computational fluid dynamics (CFD) has advanced significantly in the last few years and can now provide high fidelity temporally and spatially resolved numerical data. This data is based on meshes that range from a few million cells to tens of million cells, and for several hundred thousand time steps, with data files that are of the order of terabytes. A key challenge therefore is the ability to easily and cost-effectively mine this data for key features of the flow field and to display these spatially evolving features in the space-time domain of interest. In this work, we present an integrated interdisciplinary effort that takes utilizes of a generic approach to handle scientific data sets leading to new visualization capabilities in a natural way.

The CFD dataset is obtained from a large eddy simulation (LES) of flow inside a stirred tank reactor (STR), such as depicted in Figure 1. The STRs are widely used as mixing devices in chemical industry. The STR that we investigated here is a cylindrical

Figure 1: Stir Tank
(Courtesy Dow Chemicals)

tank with a hemispherical bottom, vertical baffles mounted along the cylindrical walls, and rotating impellers consisting rectangular blades with 45° pitch angle mounted on a shaft passing through its center, see Figure 2.

As the impeller rotates, its blades pump the fluid axially downward towards the bottom of the tank. The fluid-jet then hits the hemispherical bottom wall and sets in a circulating motion of fluid within the tank promoting mixing between the top and the bottom of the tank.

The calculations are done on a multi-block curvilinear mesh as shown in Figure 3. Calculations are done on nearly three million cells using a multi-block body fitted finite volume

Figure 2: Impeller geometry

flow solver. Since the tank contains a set of impeller blades that are rotating and also contains stationary
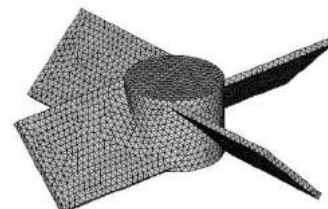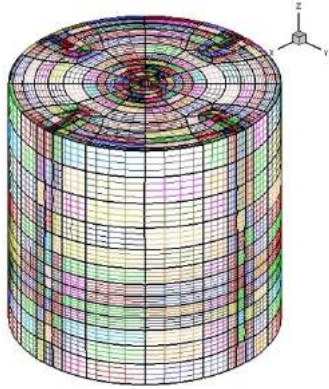
Figure 3: Boundaries of the 2088 curvilinear blocks of the computational mesh

features such as baffles on the outer walls, special algorithms (Immersed Boundary Methods) are used to accommodate moving interfaces. The method intrinsically uses a fixed curvilinear mesh fixed to the stationary features of the geometry and tracks or flags the location of the nodes adjacent to the moving surfaces or boundaries. At these flagged nodes, forcing terms/interpolations are performed to satisfy the boundary conditions at the moving surfaces. This approach has been extensively tested in an earlier communication [TRHA07] where computational solution for a similar stirred tank flow has been validated with experimental observations.

The visualizations presented here show the axial pumping as well as the circulation zones suggesting the mechanisms of fluid mixing inside the tank. As the impeller blade moves, the fluid jet stream coming out of the top and side wall of the impeller blade interact with each other to form a vortical motion following which fluid elements move in azimuthal direction [RA07]. These vortex structures are identified from the existences of pressure minima.

While traditional commercial visualization packages such as Tecplot (www.tecplot.com) and Ensight (www.ensight.com) are commonly used for visualization, they have inherent limitations. It is therefore important to be able to develop sophisticated visualization capabilities that allow a variety of features and controls not commonly available with the commercial software. The VISH visualization environment [BRH07] employed here has been designed to allow a step in this direction.

2 DATA MODEL CONCEPT

"*A common denominator for scientific data already exists, we just have to use them.*" was the paradigm under which Butler & Pendley [BP89] proposed the mathematics of fiber bundle as a foundation for a data model that is able to describe all cases needed for scientific visualization in a common way, since the mathematics behind all the different implementation is an universal language of science already. Yet a common data model has hardly been used, with the IBM Data Explore one of the few exceptions. The idea of using a fiber bundle as inspiration for a common data model has been revived by Benger [Ben04] to handle the complexity of data stemming from general relativity. These concepts have been built upon and further refined by Heinzl [HEI07] to develop generic software components. In these approaches, concepts have been based on the mathematics of topology and differential geometry to describe data sets.

2.1 Mathematics of Fiber Bundles

A fiber bundle is in the mathematical sense is set of a total space E and a base space B with a projection map f such that the union of fibers of a neighborhood U of each point of B is homeomorphic to $U \times F$ with F the so called fiber space and the projection of $U \times F$ is U again. It is also said that the space F "fibers" over the base space B. If the total space E can be written globally as $E=B \times F$, then E is called a "trivial bundle". The paradigm is that numerical data sets that are usually needed for scientific visualization can be formulated as trivial bundles. In practice it means that we may distinguish data sets by their properties in the base space and the fiber space. At each point of the – discretized – base space we have some data in the fiber space attached.

The structure of the *base space* is described as a CW-complex, which categorizes the topological structure of an n-dimensional base space by a sequence of k-dimensional *skeletons,* with the dimensionality of the skeletons ranging from zero to *n*. These skeletons carry certain properties of the data set: the 0-skeleton describing vertices, the 1-skeleton the edges, 2-skeleton the faces, etc., of a triangulation of some mesh. For structured grids the topological properties are given implicitly.

The structure of the *fiber space* is (usually) not discrete and given by the properties of the geometrical object residing there, such as a scalar, vector, co-vector, tensor. Same as the base space, the fiber space has a specific dimensionality, though the dimensionality of the base space and fiber space is independent. If the fiber space has vector space properties, then the fiber bundle is called a vector bundle. The most simple *vector bundle* is the *tangential bundle* of a manifold, which consists of the manifold as base space and the space of tangential vectors at each point. With *n* the dimensionality of the manifold, the dimension of the tangential bundle is 2*n*.

Basically a fiber bundle is a set of points with neighborhood information attached to each of them. An n-dimensional array is a very simple case of a fiber bundle (with neighborhood information given implicitly).

2.2 Benefits of Fiber Bundles

The concept of a fiber bundle leads to a natural distinction of data describing the base space and data describing the fiber space. This distinction is not common use in computer graphics, where topological properties (base space) are frequently intermixed with geometrical properties (coordinate representations). Operations in the fiber space can however be formulated independently from the base space, which leads to a more reusable design of software components. Coordinate information, formally part of the base space, can as well be considered as fiber, leading to further generalization. The data sets describing a fiber are ideally stored as contiguous arrays in memory or disk, which allows for optimized array and vector operations. Such a storage layout turns out to be particularly useful for communicating data with the GPU using vertex buffer objects: fibers are basically vertex arrays in the notation of computer graphics.

2.3 The 7-Level Hierarchy

In the data model implementation of [Ben04] data are formulated in a graph of maximally seven levels, each level representing a certain property of the entire data set. These levels, constituting a "Bundle", are:

1. Slice
2. Grid
3. Skeleton
4. Representation
5. Field
6. (Fragment)
7. (Compound Elements)

Actual data arrays are stored only below the "Field" level. An actual data set is described by which data sets exist in which level. The actual meaning of each level is described elsewhere [Ben04], [BRH07], [Ben08]. Only two of these hierarchy levels are exposed to the end-user, these are the "*Grid*" and "*Field*" levels.

2.4 Bundles, Grids and Fields

An entire dataset, including all time steps or any information given on a parameter space in general, is denoted as a *Bundle*, following the mathematical term of a fiber bundle. The objective is to formulate all data that is used for scientific visualization within this *Bundle*. A *Grid* is subset of data within the Bundle that refers to a specific geometrical entity, for instance a mesh carrying data such as a triangular surface, a data cube, a set of data blocks from a parallel computation, etc. A *Field* is the collection of data sets given as numbers on a specific topological component of a *Grid*, for instance floating point values describing pressure or temperature.

The names of *Grids* and *Fields* are arbitrary and specified by the user. All other levels of the data model have pre-defined meanings and values which are used internally to describe the properties of the *Bundle* as construction blocks. The usage of these construction blocks constitutes a certain language to describe a wide range of data sets. For instance, a *Slice* is identified by a single floating point number representing time (generalization to arbitrary-dimensional parameter spaces is possible); a *Skeleton* is identified by its dimensionality, index depth (relationship to the vertices of a Grid) and refinement level; a *Representation* is identified via some reference object, which may be some coordinate system or another *Skeleton*. The meaning of such identifiers is only used internally, but transparent to the user. The lowest levels of *Fragments* and *Compound* describe the internal memory layout of a *Field* data set and are optional. They are described in detail in [Ben08].

2.5 Formulating Data

The existence of data sets in the hierarchy of a *Bundle* defines a data set. We will give some examples here on the data sets that have been involved in our work with the stir tank computational fluid dynamics data.

2.5.1 Isosurface

An isosurface is the explicit polygonal representation of the points where a possibly time-dependent scalar field given on a volumetric manifold has constant value. It consists of the following properties:

1. A sequence of *Grids*, one for each time step
2. Coordinates for each vertex, which is a field on the Skeleton of index depth 0 ("Vertices") on each grid, as represented in Cartesian coordinates.
3. Connectivity information for each triangle, which is a field on the Skeleton of dimension 2 and index depth 1 (faces) on each Grid, as represented in the Vertices.
4. Normal vectors define a vector field (precisely: a bi-vector or co-vector field) given on the Vertices Skeleton of the Grid.
5. Optional data fields on the vertices, such as another 3D field mapped on the surface.

An Isosurface is a sequence of Grid objects with two Skeletons defined on it, with the Skeleton for the Vertices carrying two or more fields.

2.5.2 Line Set

A set of lines, such as the result of the computation of stream lines of a vector field, is given by

1. A sequence of Grids, one for each time step, for the set of lines valid at each time step

2. Coordinates for each point along the lines, a field on the Skeleton of index depth 0

3. Connectivity information for the points building up a line, which is a field on the Skeleton of dimension 1 and index depth 1 (edges).

4. Tangential vectors define a vector field given on the Vertices Skeleton of the Grid

5. Optional data fields on the vertices may exist, as retrieved from a 3D field mapped on the surface.

A set of lines is very similar to an isosurface in this data model, though a different Skeleton on the Grid objects is employed.

2.5.3 Multiblock Data

The numerical data as provided by the computational fluid simulation are given as a collection of 2088 three-dimensional arrays describing coordinate location, pressure and fluid velocity for each grid point. These fields are all fibers on the vertices in the fiber bundle data model, where we also treat the coordinates as a field over the vertices. As the topological structure is regular the edges and faces are given implicitly. The decomposition of the data into blocks is represented as the internal memory layout structure of the fields, thus as field fragments visible in the 6th level of the data hierarchy. The topological structure of the blocks is thereby transparent to the user, but algorithms operate on collections of arrays instead of contiguous arrays, which is relatively straightforward extension to existing algorithms such as the computation of isosurfaces. To store information that is specific to each block, we consider each block as a collection of volume cells. Volume cells are topologically 3-Skeletons in a triangulation, and of index depth 1 in the fiber bundle data model. Collections of such elements are thus of index depth 2. Fibers on this Skeleton are thus a natural place to store e.g. the geometrical bounding box information for a block, or the min/max data range of a scalar field (which speeds up repeated isosurface computations) The layout of a multiblock data set thus consists of:

1. A sequence of Grid objects, one for each time step

2. A coordinate field on the Vertices Skeleton, which is shared among all time steps if the geometry remains constant over time

3. A scalar field on the Vertices Skeleton

4. A vector field on the Vertices Skeleton

5. A Skeleton of dimension 3 and index depth 2 describing a collection of volume elements, which are the respective blocks.

6. Optional scalar fields on the (3,2) Skeleton with min/max information of a scalar field per block, or coordinate fields for bounding box.

2.6 Data Operations

Given the certain components constituting the data model, we may formulate abstract operations among such components. These operate on abstract high-level objects without requirement to know the internal structure of the objects, though their concrete implementation will have to deal with them.

2.6.1 Isosurface Computation

The computation of an isosurface is an operation that takes a Field as input and yields a Grid object, and will be called for each instance of a time sequence. The operation is parameterized by some isolevel value. Certain conditions must be fulfilled by the input field for this operation to succeed, such as being a scalar-valued field residing on a regular grid. However, more advanced implementations rather than the standard marching cubes may well be formulated through the same interface, such as direct computation of magnitude isolevels of vector fields, or isosurface computation on tetrahedral grid. The high-level operation

<div align="center">

Grid = ComputeIsosurface(Field, float);

</div>

remains the same, and no changes in the user interface are required. For instance, more advanced operations could be invoked via some runtime plugin mechanism, transparent to the user, as it is supported by VISH.

2.6.2 Grid Evaluation

Given Field data on one Grid instance, like a 3D volume, they may be evaluated on another Grid, like a surface or a line set. Such is formulated as an EvalGrid operation which requires specification of a destination grid and a source field (implicitly given on another Grid object):

<div align="center">

Field = EvalGrid(Grid Dest, Field Source);

</div>

Certain constraints may apply to the input Grid, since the evaluation of a field is not possible on an arbitrary Grid. For instance, data given on a surface cannot be uniquely extrapolated into an entire 3D volume. However, a wide range of operations can be specified through this common API.

3   STREAMLINE STRATEGIES

Visualizing vector fields is a common need in CFD for investigating velocity fields. [LaH05] describes several tools for investigating CFD data. [PWS06] shows the combination of different geometry based and texture based techniques in a CFD application. More applications of texture based vector field visualization can be found in [LEG].

With more complicated and large data sets it is increasingly important to have a variety of tools for feature extraction and data exploration at hand. Thus, developing a flexible framework seems the right way to meet the increasing requirements effectively. We use streamlines in our approach because it is a well known standard technique and they can be described well within the context of fiber bundles.

The challenge here was to deal with the multi-block curvilinear data structure (as shown in Figure 3) and to verify the applicability of the *Fiber Bundle* data model. Using this model, our streamline visualization implementation modules separate into four groups: *Vector Field Data*, *Seed Point Data*, *Streamline Computation* and *Line Rendering*. The software modules are connected in a *directed graph* communicating data using *Grids* and *Fields*. This allows to exchange modules by other modules (high reusability) and to combine modules in different ways (high flexibility).The *Streamline Computation Module* takes a vector *Field* and an arbitrary *Grid* defining the seed points as inputs and outputs a *Line Set* as *Grid*.
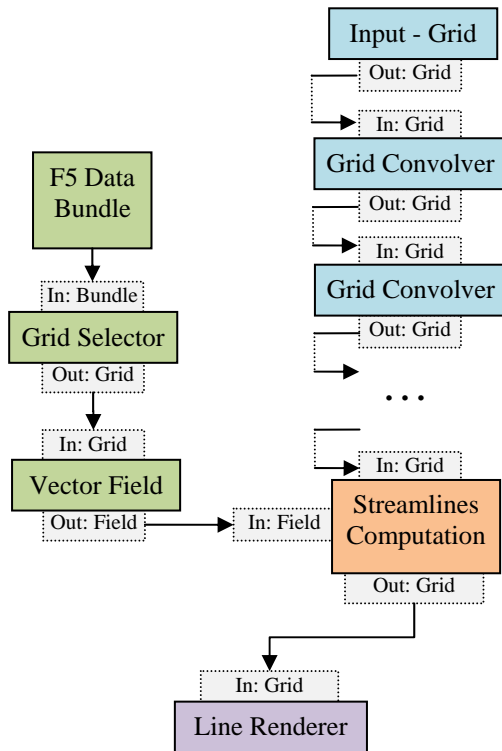

Figure 4: Streamline modules and dataflow

The computation of the streamline involves finding the location of a given *world position* in the dataset by identifying its block, cell and local curvilinear cell coordinates. These local coordinates are then used for linear interpolation of the vector field. The interpolated vector is used to advance to the next streamline point of the streamline, another world position.

Firstly, a kD-Tree is employed to find blocks, which might contain the *world position*, secondly a look up data structure called *UniGridMapper* that maps a uniform grid cell to curvilinear cells that might contain the world position is used and finally a Taylor approximation and Newton iteration, as described in [STA98], retrieves the local curvilinear cell coordinates.

Besides storing the calculated stream lines as *Line Set*s in the output *Grid* additional *Fields* are stored that carry the information necessary for interpolation. Block IDs and local cell coordinates are stored. This information can then be used by other independent modules to evaluate other data *Fields* on the streamline *Grid*, e.g. to evaluate a pressure *Field*. The final *Line Rendering* module employs *illuminated stream lines*, similar to those described in [SZH97]

3.1   Defining Seed Point for Streamlines

To visualize the characteristics of a certain vector field by streamlines it is important to find good starting points (seed points) for the streamline integration.

3.1.1   Grid Convolver

An operation called *Grid Convolver* allows the user to create sophisticated seed point geometries by 'convoluting' vertices of an input *Grid* with a set of parameters into an output *Grid*, similar to the mathematical convolution operation. Possible *Grid* convolution operations are *Point*, *Line*, *Rectangle*, *Circle*, *Ellipsoid* and *Uniform Grid*.

Figure 5 demonstrates a setup for creating seed points involving three *Grid Convolvers*. The first *Grid Convolver(a)* gets one point as input *Grid*. It 'convolutes' this input on a vertical *Line* with a subdivision of three points. This is then outputted into the second *Grid Convolver(b)* which 'convolutes' the three-point-*Line* on its horizontal two-point-*Line*. This results in the output *Grid* seen in (c). A final *Grid Convolver(d)* now 'convolutes' on its *Circle* geometry resulting in the final seed points in the output *Grid* of (d), shown in (e).

Since the module can be connected to any other modules that output *Grid* objects it is possible to create many different seed geometries. The module thus is highly reusable and flexible. Figure 4 illustrates a typical dataflow involving *Field* and

Grid objects. Figure 6, 7, and 8 demonstrate how this module was used to investigate the stir tank velocity field with different settings of position and rotation and different number of connected *Grid Convolvers.*

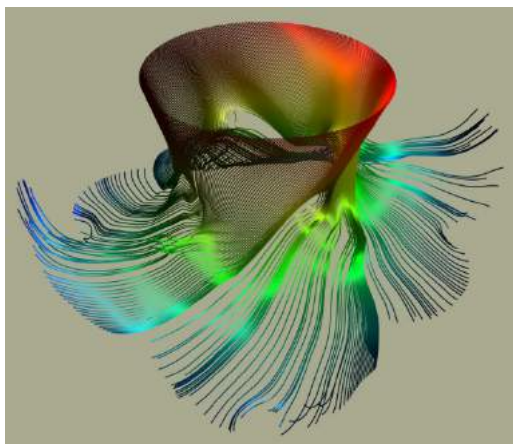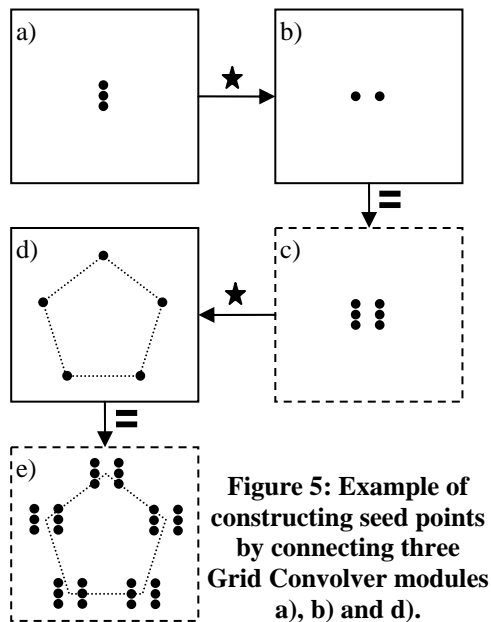**Figure 5: Example of constructing seed points by connecting three Grid Convolver modules a), b) and d).**

Figure 6: Streamlines with seed points on a circle.
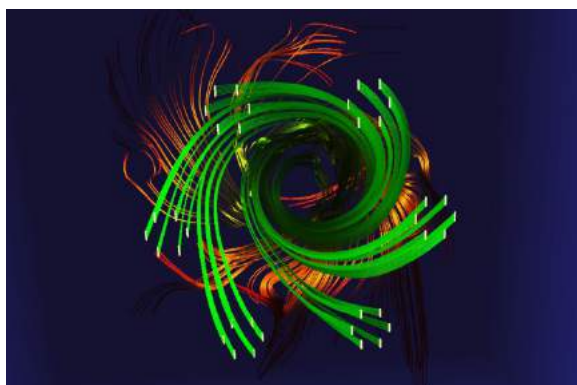
Figure 7: Streamlines emitted from seed points on a circle of circles of lines by using three Grid Convolvers.
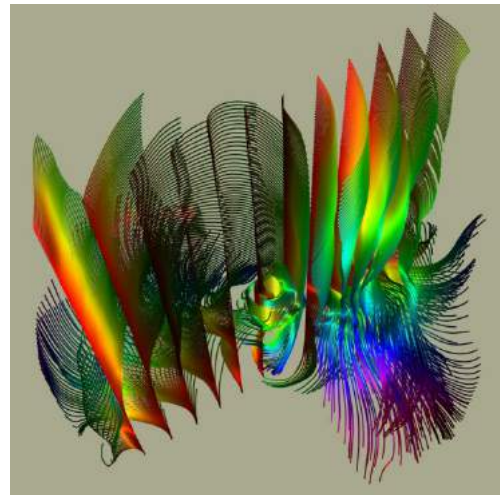
Figure 8: Streamlines emitted on an array of lines.
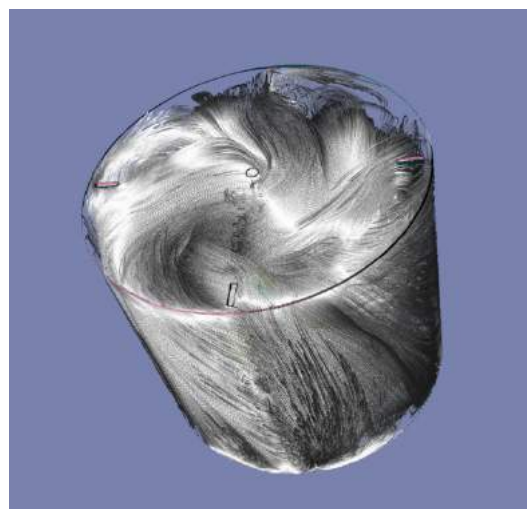
Figure 9: Space-filling streamlines in the STR

Besides interactively specifying seed points it is also possible to utilize other seeding methods such as those found in AMIRA [SWH05], where a threshold on a scalar field can specify seed points for a vector field, or similar to Weinkauf's method of computing the curvature measures of a vector field to find critical regions as indicators where streamlines were most interesting [WTH02]. Even a full coverage of the entire volume may provide worthwhile information (Figure 9).

3.1.2   Seed Points by Iso Surfaces

The usage of a *Grid* input at the *Streamline Computer* allows usage of arbitrary compatible *Grid* objects for defining seed points. For example, the *Grid* of an iso surface (the vertices of the triangular surface) computed on the pressure scalar field can be used as input, as shown in Figure 9 and Figure 10. With such streamline seeding the vector field close to the surface can be explored, similar to a texture based technique applied to the surface such as [LSH04]. This is a unique feature of VISH that can potentially be exploited to better understand the flow physics.

4 PERFORMANCE RESULTS

Computation of 100 streamlines in the given multiblock dataset of 2088 curvilinear blocks required about 7 seconds using an Euler integration scheme for 100 steps on a Intel Xeon CPU, 2.5GHz. Tecplot required 35 seconds using a comparable setup. We could not compare with Amira, since this data type is not supported there. The computation time of the isosurface crucially depends on the chosen level, and is below $1/30^{th}$ second for most values, but may require up to 2 seconds in particular cases. Computing streamlines from the isosurface vertex requires about 5-10 seconds for the setup as shown in Figure 11, but will linearly depend on the chosen length. Future improvements will utilize higher order integration schemes and parallelization, for which we expect to be able to reduce computation times under one second. The frame rates for rendering itself once the streamlines are computed are under $1/30^{th}$ of a second in all cases except Figure 9, where we got about 10 frames per second on an NVidia Quadro FX 5600 graphics board.
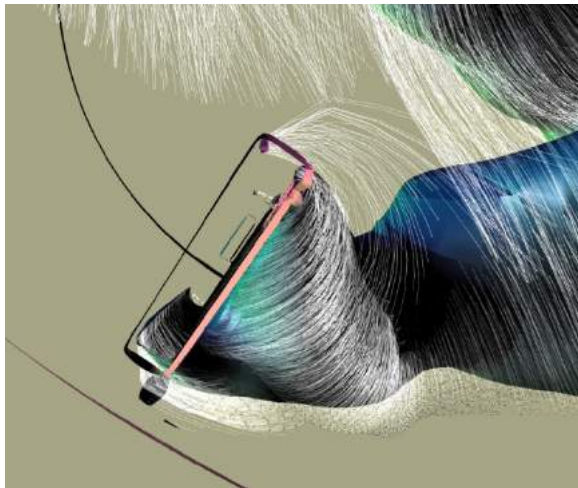
5 DISCUSSION

Figure 10: Detail of the emission of streamlines from the pressure isosurface.

Figure 10 shows velocity vectors adjacent to the impeller blades. It can be clearly seen that as the impeller rotates in the clockwise direction, the pitched blade surfaces force the fluid in its surface-normal direction imposing both radially outward and downward velocities on the flow along with an azimuthal velocity component due to its rotation. The pressure isosurface depicts the boundary layer formed over the impeller blade surface and its evolution downstream. Circular depressions (marked as P in Figure 11) can be observed in the pressure isosurface. These depressions point to local pressure minima suggesting formations of vortical structures in the direction opposite to the impeller rotation. These vortices (commonly termed as trailing-edge

vortices) convect fluid in the azimuthal direction and play a key role in mixing within the tank.
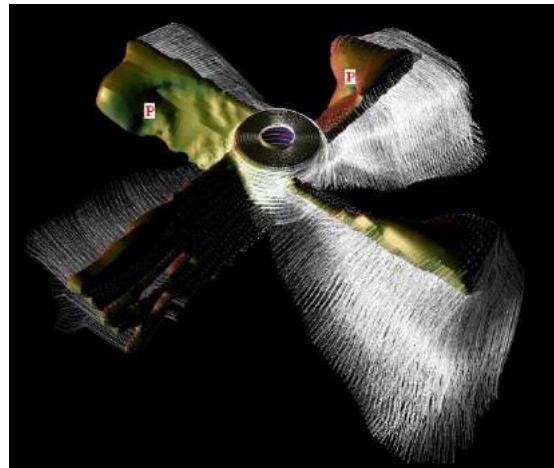
Figure 11: Velocity micro-streamlines and pressure isosurfaces over the impeller blade

Figure 12 shows the streamlines coming out of an impeller blade. The downward pumping from the impeller is clearly observable. Also, the streamlines are observed to bend near the hemispherical bottom of the tank and establish a circulating motion. The colors along the streamlines indicate the residence time of the fluid particles. The suction of the upper fluid is identified by the green color, the yellow colored impeller jet stream has both radial and axial component as the streamlines show almost a 45° bend. Later (orange color) these lines hit the bottom of the tank and bend upwards (red coloration) and the fluid is again convected to the upper part of the tank. This circulation is how the fluid mixing operation takes place in the stirred tank.
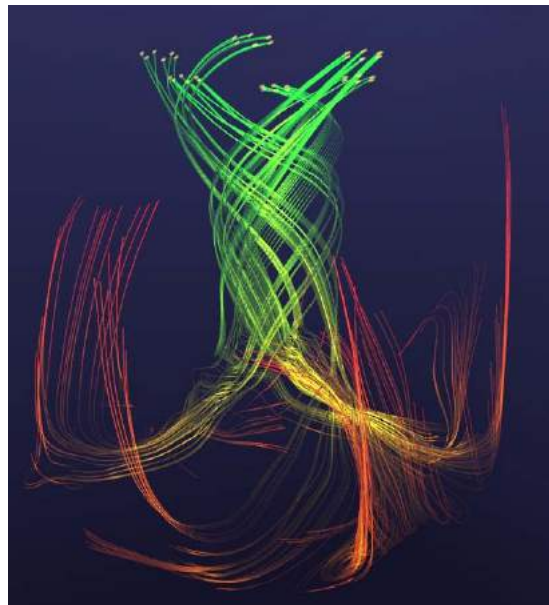
Figure 12: Streamlines over the tank volume

6   SUMMARY

We have described and applied the concept of "Grid objects" as elementary tools within a highly modular visualization environment to provide powerful seeding mechanisms for streamline computation to visualize flow in a stirred tank. Operations such as "Grid convolution" and seeding by pressure isosurfaces are natural consequences of utilizing the described data model. It is argued here that this approach offers specific capabilities that several other visualization platforms are unable to provide.

7   ACKNOWLEDGMENTS

8   REFERENCES

[Ben04] Benger, W. **Visualization of General Relativistic Tensor Fields via a Fiber** , PhD **-** thesis FU Berlin, 2004

[BRH07] W. Benger, G. Ritter and R. Heinzl, **The Concepts of VISH**, Proc. 4th High End Visualization Workshop Obergurgl, p. 26-39, 2007, Lehmann's Media

[Ben08] Benger, W. **Colliding galaxies, rotating neutron stars and merging black holes - visualizing high dimensional data sets on arbitrary meshes**, *New J. Phys. 10 (2008) 125004*, http://stacks.iop.org/1367-2630/10/125004.

[BP89] O. Butler, D. M. & Pendley, M. H. (1989). **A visualization model based on the mathematics of fiber bundles**. *Comp. in Physics*, 3(5), 45-51.

[HEI07] R. Heinzl: "*Concepts for Scientific Computing*"; PhD Thesis; Institut für Mikroelektronik, TU Wien, 2007;

[LaH05] Robert S. Laramee & Helwig Hauser, **Interactive 3D Flow Visualization Based on Textures and Geometric Primitives**, in NAFEMS World Congress Conference Proceedings, The International Association for the Engineering Analysis Community, May 17-20, 2005, St. Juliens Bay, Malta

[LEG] R. S. Laramee, G. Erlebacher, C. Garth, T. Schafhitzel, H. Theisel, X. Tricoche, T. Weinkauf, D. Weiskopf, **Applications of Texture-Based Flow Visualization**, Engineering Applications of Computational Fluid Mechanics, in publication

[LSH04] R. S. Laramee, J. Schneider & H. Hauser: **Texture-Based Flow Visualization on Isosurfaces from Computational Fluid Dynamics**, Proceedings of the 6th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2004), May 19-21, 2004, Konstanz, Germany

[MPS05] O. Mallo, R. Peikert, C. Sigg, F. Sadlo, **Illuminated Lines Revisited**, *Proceedings of IEEE Vis 2005, pp. 19-26 (Minneapolis, MN, USA, October 23-28, 2005)*

[PWS06] C. Petz, T. Weinkauf, H. Streckwall, F. Salvatore, B.R. Noack, H.-C. Hege, **Vortex Structures at a Rotating Ship Propeller**, Presented at the 24th Annual Gallery of Fluid Motion exhibit, held at the 59th Annual Meeting of the American Physical Society, Division of Fluid Dynamics, Tampa Bay, November 2006

[RA07] S. Roy, and S. Acharya, "Study on Flow and Turbulence Inside a Stirred Tank and Investigation on the Effects of Macroinstability on Trailing Vortex Structures", *ASME International Mechanical Engineering Congress and Exposition*, Seattle, November 2007

[STA98] D. Stalling. "Fast texture-based algorithms for vector field visualization." Dissertation, FU Berlin, Preprint SC-98-58, Konrad-Zuse-Zentrum Berlin (ZIB), December 1998.

[SWH05] D. Stalling and M. Westerhoff and H.-C. Hege, "Amira - an object oriented system for visual data analysis", Visualization Handbook, Christopher R. Johnson and Charles D. Hansen, Academic Press, 2005

[SZH97] D.Stalling, M.Zockler, and H.-C. Hege. **Fast display of illuminated field lines**. In *IEEE Transactions on Visualization and Computer Graphics*, vol.3, pages 118-128, 1997.

[SZH97] D.Stalling, M.Zockler, and H.-C. Hege. **Fast display of illuminated field lines**. In *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pages 118-128, 1997.

[TRHA07] M. Tyagi, S. Roy, S. Acharya, and A-D Harvey III, "Simulation of laminar and turbulent impeller stirred tanks using immersed boundary method and large eddy simulation technique in multi-block curvilinear geometries", *Chemical Engineering Sciences*, volume 63, issue 5, pages 1351-1363, 2007

[WTH02] T. Weinkauf, H. Theisel, **Curvature Measures of 3D Vector Fields and their Applications**, Journal of WSCG 10(2), WSCG 2002, Plzen, Czech Republic, February 4 - 8, 2002

Beyond the Visualization Pipeline:
The Visualization Cascade

Werner Benger[1] and Georg Ritter[2] and
Marcel Ritter[1,3] and Wolfram Schoor[3]

[1]Center for Computation and Technology, Louisiana State University, USA

[2]Institute for Astro- and Particle Physics, University of Innsbruck

[3]Institute of Computer Science, University of Innsbruck, Austria

[4]Fraunhofer Institute for Factory Operation and Automation

werner@cct.lsu.edu, georg.ritter@uibk.ac.at,
marcel@cct.lsu.edu, Wolfram.Schoor@iff.fraunhofer.de

Abstract

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

1 Introduction

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

has been a widely successful concept for the design of visualization software. Several well known software packages have been built upon this idea, examples include AVS [Upson et al., 1989], VTK [Schroeder et al., 1997], IRIS Explorer [Foulser, 1995], OpenDX [Treinish, 1997] (for a more complete list see [A.A. Ahmed, 2007]).
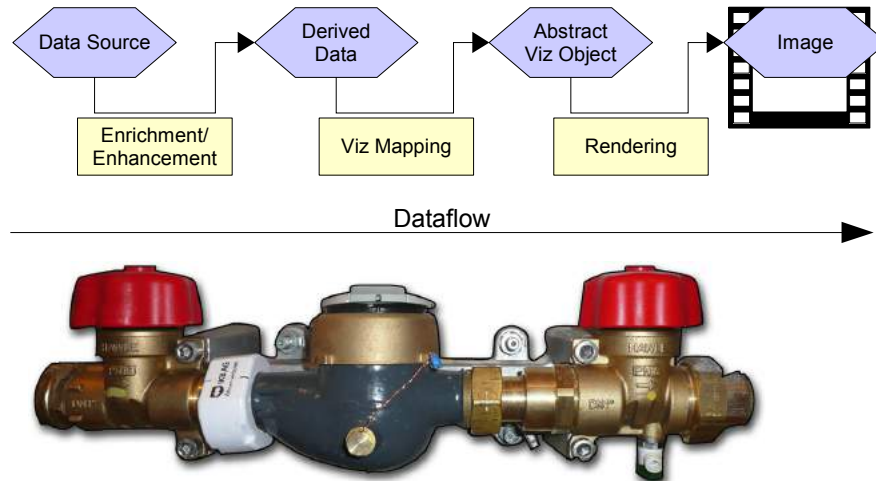
Figure 1: Visualization Pipeline: Data flows through the pipeline while operators modify the stream. They extract, filter, map and render data. Finally the pipeline outputs is an image or image stream.

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

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

media data is handled this way.

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

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

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

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

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

2   The Visualization Cascade

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

scenario is running an animation of time-dependent data. The "conventional way" is to load each time step, pump it through the visualization pipeline to create a pixel frame for each time step, and then eventually watch the evolution of the data as a movie. Many features of a dynamic data-set are only appreciated when viewed at interactive speeds of e.g. 30 frames per second, but usually the traversal of the entire visualization pipeline is much slower than 1/30th of second.
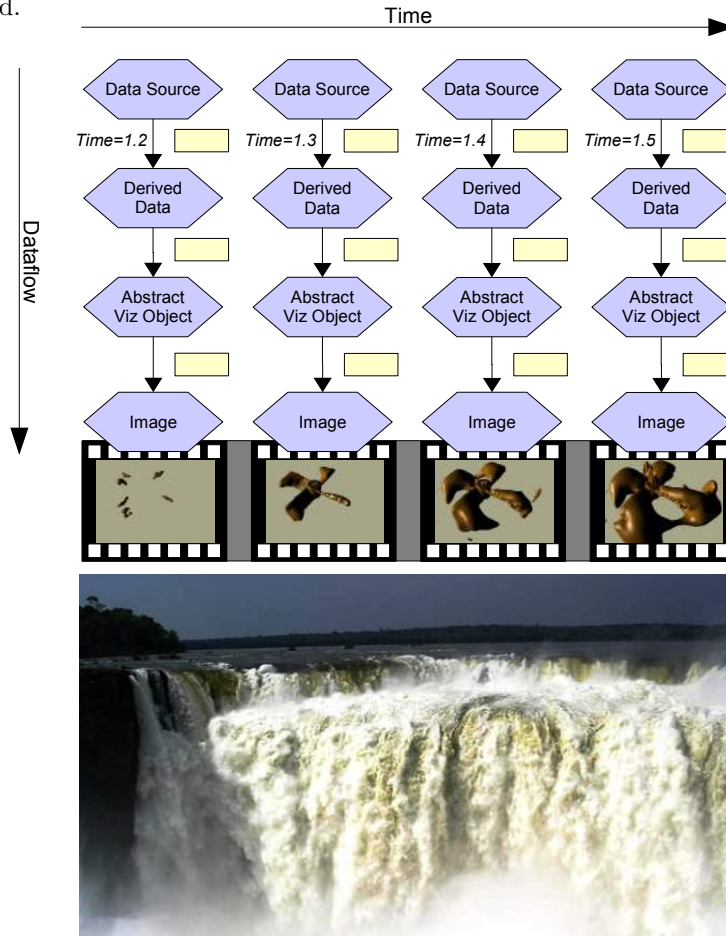
Figure 2: Exploring a full parameter space of some data-set requires parallel traversal of the data flow - resulting in a cascade rather than a single pipeline traversed repeatedly.

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

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

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

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

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

2.1 Data Result Caching

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

data in here. Using this associated node-local space as cache for intermediate results is an option, but not an optimal one.
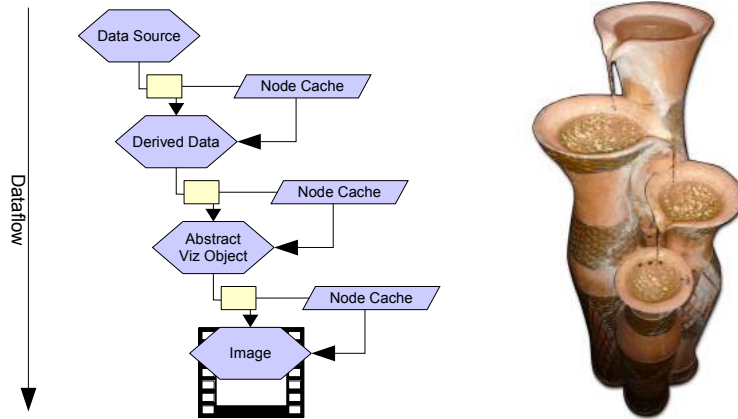
Figure 3: Caching of intermediate results of the data flow through a visualization pipeline allows to avoid repeating previously performed computations. Instead, final results may be retrieved from intermediate levels of the visualization *cascade*.

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

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

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

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

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

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

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

2.2 Operator Cache

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

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

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

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

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

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

The Operator Cache is furthermore related to a set of variable values, a "ValueSet". Its purpose is to associate the OperatorCache with such a set of values. If any of these values has changed upon a repeated call of the viz node's compute function on the *same* data-set then the Operator Cache needs to be equipped with data from a new computation. For instance we might consider an operator that computes some isosurface. If the isolevel value or some maximum number of allowed triangles is changed, then the operator would need to execute the numerical routine again, otherwise it could just return the data stored in the Operator Cache. An "OperatorCache::unchanged()" member function checks if
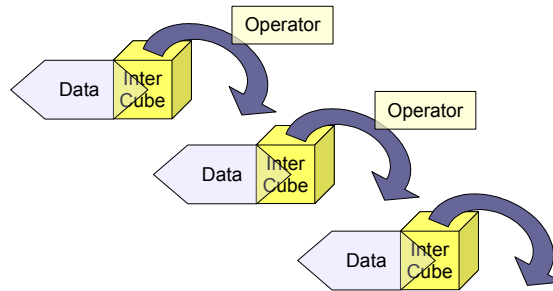
Figure 4: Data Storage in Intercubes

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

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

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

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

2.3 Caching in the Fiber Bundle

When data are available in the fiber bundle, and results are storable in the fiber bundle, one would not employ the OperatorCache. Rather, any results will be stored directly in the incoming data structures. To depict how it works, we do not need to know the entire complexity of the full model. It suffices to know that there are objects called `Bundle` and `Grid`. A `Bundle` may be accessed with a floating point value and a string to yield a `Grid` object. Such a `Grid` object may represent a 3D data volume with scalar fields (which is to be identified via some string), or a triangular surface such as the result of an isosurface computation. The actual numerical routine "`Isosurface`" will require a `Grid` object, a field name, and a floating point value specifying the isolevel. The schematic code will look similar to this:
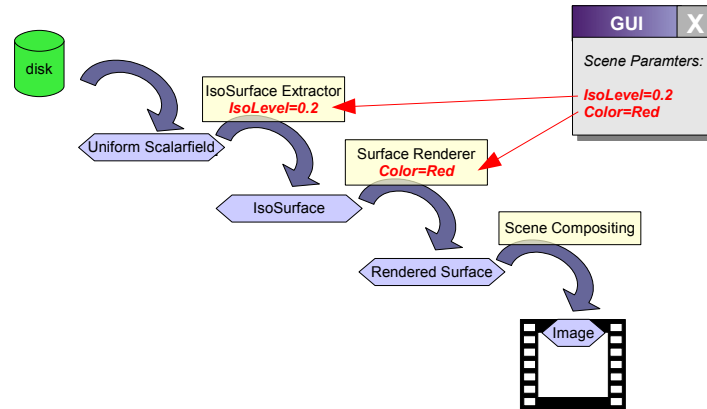
Figure 5: The execution flow: The first time, the complete cascade has to be executed. The operators read the data-set from disk, compute the isosurface, render the surface and composite it to the final image. The separation into operators is hidden and is not all seen by the user.

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

Note that in case an IsoSurface Grid already exists, there is no need to request a DataVolume object. The operation of requesting such might be effortsome, including slow disk access (on-demand loading), numerical computation of the source field, network access, etc. Never are any data actually stored in the `VizNode` object itself. In this version, a new geometry is stored for each time step and each isolevel value. Since these are floating point values, this may well need to an immense number of surfaces that are stored when exploring the parameter space of time and isolevel. Therefore, some appropriate memory
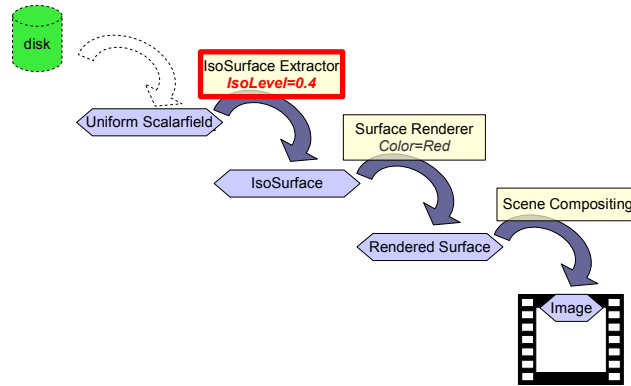
Figure 6: The execution flow: Resulting data flow when changing the isosurface level parameter. The data-set need not be read from disk again.
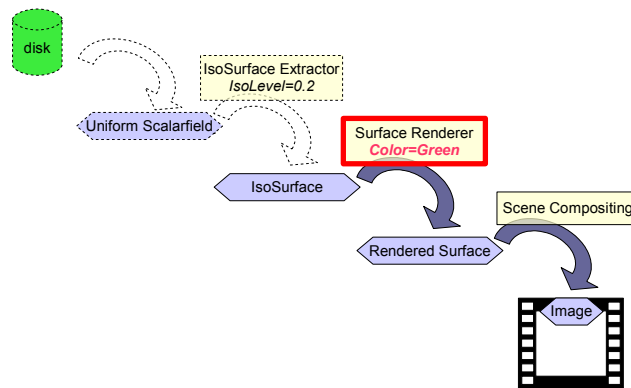
Figure 7: The execution flow: Resulting data flow when changing the color of the iso surface. The isosurface need not be recomputed again.

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

2.4 OpenGL Caching

The final stage of producing pixels using modern graphics hardware is loading data onto the memory of the graphics card (GPU). Once all data that are required for rendering are transferred to the GPU, pixel generation will be as fast as possible. Via means of OpenGL, large data at the GPU are modeled as *Display Lists, Textures* and *VertexBuffer Objects*. Framebuffer objects might fall into this category as well, but we did not consider them yet. While the graphics memory is limited, it may still provide enough space to also store objects that are not visible in the currently viewed frame but a previous one. Re-utilizing
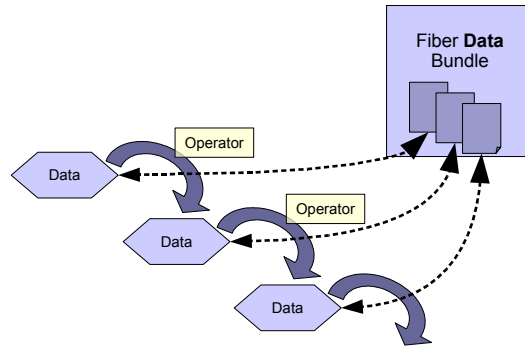
Figure 8: Data Storage in the Fiber Bundle

objects already stored in GPU memory is much faster than re-loading objects from RAM. We may expect so even in case the graphics driver is placing some object into RAM itself.
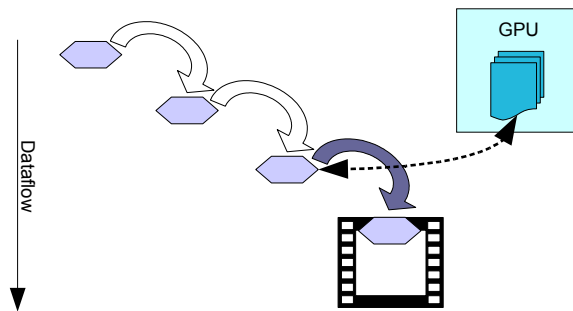
Figure 9: Caching on the GPU

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

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

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

1. an arbitrary Intercube object

2. an intrinsic C++ type ID

3. a set of values

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

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

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

3 Conclusion

An universal caching mechanism has been presented. It can be used to extend the visualization pipeline model (as defined by [Haber & McNabb, 1990]) to maximize the reuse of computed data and thereby minimizing the response time of an interactive visualization to parameter changes. The mechanism can relate computed data for all parameters of the visualization, which make fast and easy navigation in the whole parameter space possible. Special emphasis
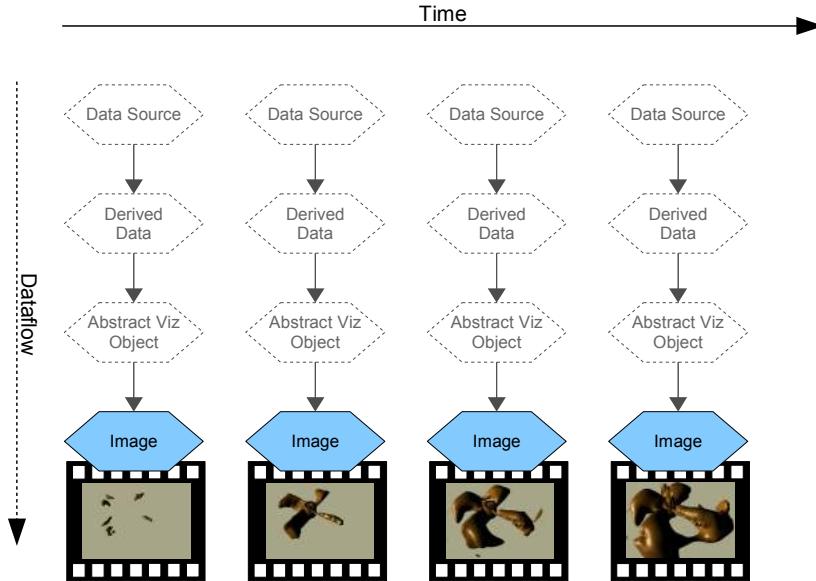
Figure 10: A GPU cached visualization cascade provides the animation without expensive data flow.

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

4 Acknowledgements

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using Geometric Algebra for Navigation in Riemannian and Hard Disc Space

Werner Benger
Center for Computation &
Technology
Louisiana State University
239 Johnston Hall
Baton Rouge, LA 70803, USA
werner@cct.lsu.edu

Andrew Hamilton
Center for Astrophysics and
Space Astronomy
JILA, University of Colorado
Boulder, CO 80309, USA
Andrew.Hamilton
@colorado.edu

Mike Folk/Quincey Koziol
The HDF Group
1901 So. First St. Suite C-2
Champaign, IL 61820. USA
mfolk@hdfgroup.org

Simon Su
Princeton Institute for
Computational Science and
Engineering
345 Peter B. Lewis Library
Princeton, NJ 08544, USA
simonsu@princeton.edu

Erik Schnetter
Center for Computation &
Technology
Dept. of Physics & Astronomy
Louisiana State University
Baton Rouge, LA 70803, USA
schnetter@cct.lsu.edu

Marcel Ritter/Georg Ritter
Department of Computer
Science
University of Innsbruck
Technikerstrasse 21a
A-6020 Innsbruck, Austria
csab7885@uibk.ac.at

ABSTRACT
A "vector" in 3D computer graphics is commonly understood as a triplet of three floating point numbers, eventually equipped with a set of functions operating on them. This hides the fact that there are actually different kinds of vectors, each of them with different algebraic properties and consequently different sets of functions. Differential Geometry (DG) and Geometric Algebra (GA) are the appropriate mathematical theories to describe these different types of "vectors". They consistently define the proper set of operations attached to each class of "floating point triplet" and allow to derive what meta-information is required to uniquely identify a specific type of vector in addition to its purely numerical values. We shortly review the various types of "vectors" in 3D computer graphics, their relations to rotations and quaternions, and connect these to the terminology of co-vectors and bi-vectors in DG and GA. Not only in 3D, but also in 4D, the elegant formulations of GA yield to more clarity, which will be demonstrated on behalf of the use of bi-quaternions in relativity, allowing for instance a more insightful formulation to determine the Newman-Penrose pseudo scalars from the Weyl tensor.

1. INTRODUCTION
Geometric Algebra and the sometimes mystified concept of spinors eases implementation and intuition significantly, both in computer graphics and in physics. We demonstrate the concrete application of these concepts in two independently developed computer graphic software packages, where Geometric Algebra is used for navigating the camera position in space and time. Another application example is given by a simulation code solving Einstein's equation in general relativity numerically on supercomputers, outputting the Newman-Penrose pseudo scalars as primary quantities of interest to study gravitational waves, both for visualization and observational verification.

Geometric Algebra moreover provides means to describe how the metadata information required per "vector" can be provided in persistent storage. Given large datasets that are expensively collected or generated by simulations requiring millions of CPU hours, it is increasingly important and difficult to be able to share and correctly interpret such datasets years after their generation, across different research groups from different fields of science. A unique, standardized, extensible identification of the geometric properties of the dataset elements is a necessary pre-requisite for this. similar to the way in which the IEEE standard for floating point values enables sharing floating point values. We utilize the mechanisms as provided by the HDF5 library here, a generic self-describing file format developed for large datasets as used in high performance computing. It allows specifying metadata in addition to the purely numerical data, providing an abstraction layer for specifying the mathematical properties on top of the lower-level binary layout. It is therefore desirable to us the functionality of this powerful I/O library to express the semantics of vector quantities as they arise in Geometric Algebra. This will be discussed in section 5.

2. VECTOR SPACES
A vector space over a field F (such as $\mathbb{R}$) is a set V together with two binary operations *vector addition* $+ : V \times V \to V$ and *scalar multiplication* $\circ : F \times V \to V$. The elements of V are called *vectors*. A vector space is closed under the operations $+$ and $\circ$, i.e., for all elements $u, v \in V$ and all elements $\lambda \in F$ there is $u + v \in V$ and $\lambda \circ u \in V$ (vector space axioms). The vector space axioms allow computing the dif-

ferences of vectors and therefore defining the derivative of a vector-valued function $v(s) : \mathbb{R} \to V$ as

$$\frac{d}{ds} v(s) := \lim_{ds \to 0} \frac{v(s + ds) - v(s)}{ds} \quad . \tag{1}$$

2.1 Tangential Vectors

In differential geometry, a tangential vector on a manifold M is the operator $\frac{d}{ds}$ that computes the derivative along a curve $q(s) : \mathbb{R} \to M$ for an arbitrary scalar-valued function $f : M \to \mathbb{R}$:

$$\left. \frac{d}{ds} f \right|_{q(s)} := \frac{df(q(s))}{ds} \quad . \tag{2}$$

Tangential vectors fulfill the vector space axioms and can therefore be expressed as linear combinations of derivatives along the n coordinate functions $x^\mu : M \to \mathbb{R}$ with $\mu = 0 \ldots n - 1$, which define a basis of the tangential space $T_{q(s)}(M)$ on the n-dimensional manifold M at each point $q(s) \in M$:

$$\frac{d}{ds} f = \sum_{\mu=1}^{n-1} \frac{dx^\mu(q(s))}{ds} \frac{\partial}{\partial x^\mu} f =: \sum_{\mu=1}^{n-1} \dot{q}^\mu \partial_\mu f \tag{3}$$

where $\dot{q}^\mu$ are the components of the tangential vector $\frac{d}{ds}$ in the chart $\{x^\mu\}$ and $\{\partial_\mu\}$ are the basis vectors of the tangential space in this chart. We will use the Einstein sum convention in the following text, which assumes implicit summation over indices occurring on the same side of an equation. Often tangential vectors are used synonymous with the term "vectors" in computer graphics when a direction vector from point A to point B is meant. A tangential vector on an n-dimensional manifold is represented by n numbers in a chart.

2.2 Co-Vectors

The set of operations $df : T(M) \to \mathbb{R}$ that map tangential vectors $v \in T(M)$ to a scalar value $v(f)$ for any function $f : M \to \mathbb{R}$ defines another vector space which is dual to the tangential vectors. Its elements are called *co-vectors*.

$$< df, v >= df(v) := v(f) = v^\mu \partial_\mu f = v^\mu \frac{\partial f}{\partial x^\mu} \tag{4}$$

Co-vectors fulfill the vector space axioms and can be written as linear combination of co-vector basis functions dx^μ:

$$df =: \frac{\partial f}{\partial x^\mu} dx^\mu \tag{5}$$

with the dual basis vectors fulfilling the duality relation

$$< dx^\nu, \partial_\mu >= \begin{cases} \mu = \nu : & 1 \\ \mu \neq \nu : & 0 \end{cases} \tag{6}$$

The space of co-vectors is called the co-tangential space $T_p^*(M)$. A co-vector on an n-dimensional manifold is represented by n numbers in a chart, same as a tangential vector. However, co-vector transforms inverse to tangential vectors when changing coordinate systems, as is directly obvious from eq. (6) in the one-dimensional case: As $< dx^0, \partial_0 >= 1$ must be sustained under coordinate transformation, dx^0 must shrink by the same amount as ∂_0 grows when another coordinate scale is used to represent these vectors. In higher dimensions this is expressed by an inverse transformation matrix, as demonstrated in Fig. 1. In Euclidean
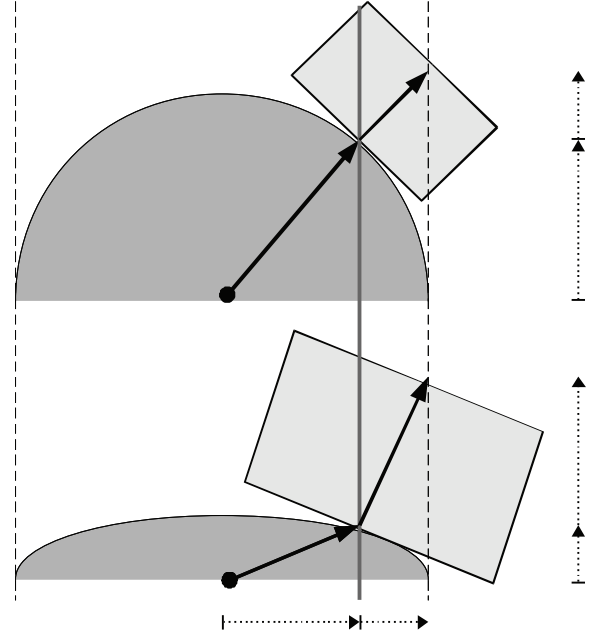
Figure 1: Vector transformation under shrinking the height coordinate by a factor of two: tangential vectors (differences between two points) shrink in their height component by a factor two as well, whereas surface normal vectors (co-vectors) grow by a factor two in height, see the vertical components of the vector and co-vector shown on the right hand side in the figure.

three-dimensional space, a plane is equivalently described by a "normal vector", which is orthogonal to the plane. While "normal vectors" are frequently symbolized via a vector arrow, like tangential vectors, they are not the same, rather they are dual to tangential vectors. It is more appropriate to visually symbolize them as a plane. This visual is also supported by (5), which can be interpreted as the total differential of a function f: a co-vector describes how a scalar function advances in space, which can be visualized as surfaces of constant function value ("isosurface"). On an n-dimensional manifold a co-vector is correspondingly symbolized by an $(n - 1)$-dimensional subspace.

2.3 Tensors

A *tensor* T_m^n of rank $n \times m$ is a multi-linear map of n vectors and m co-vectors to a scalar

$$T_m^n : \underbrace{T(M) \times \ldots T(M)}_{n} \times \underbrace{T^*(M) \times \ldots T^*(M)}_{m} \to \mathbb{R} \quad . \tag{7}$$

Tensors are elements of a vector space themselves and form the tensor algebra. They are represented relative to a coordinate system by a set of k^{n+m} numbers for a k-dimensional manifold. The construction of an tensor of higher rank from lower rank is called the *outer product* (or tensor product), denoted by $\otimes$:

$$T \equiv T^{ab} \partial_a \otimes \partial_b = v^a u^b \partial_a \otimes \partial_b = v^a \partial_a \otimes u^b \partial_b = v \otimes u \tag{8}$$

Tensors of rank 2 may be represented using matrix notation. Tensors of type T_1^0 are equivalent to co-vectors and called co-variant, in matrix notation (relative to a chart) they correspond to rows. Tensors of type T_0^1 are equivalent to a tangential vector and are called contra-variant, corresponding to columns in matrix notation. The duality relationship between vectors and co-vectors then corresponds to the matrix multiplication of a $1 \times n$ row with a $n \times 1$ column, yielding a single number

$$< a, b > = < a^\mu \partial_\mu, b_\mu dx^\mu > \equiv (a^0 a^1 \dots a^n) \begin{pmatrix} b^0 \\ b^1 \\ \dots \\ b^n \end{pmatrix} . \quad (9)$$

By virtue of the duality relationship (6) the contraction of lower and upper indices is defined as the *interior product* ι of tensors, which reduces the dimensionality of the tensor:

$$\iota : T_n^m \times T_k^l \to T_{n-l}^{m-k} : u, v \mapsto \iota_u v \quad (10)$$

The interior product can be understood (visually) as a generalization of some "projection" of a tensor onto another one.

Of special importance are symmetric tensors of rank two $g \in T_2^0$ with $g : T(M) \times T(M) \to \mathbb{R} : u, v \mapsto g(u, v)$, $g(u, v) = g(v, u)$, as they can be used to define a *metric* or *inner product* on the tangential vectors. Its inverse, defined by operating on the co-vectors, is called the co-metric. A metric, same as the co-metric, is represented as a symmetric $n \times n$ matrix in a chart for a n-dimensional manifold.

Given a metric tensor, one can define equivalence relationships between tangential vectors and co-vectors, which allow to map one into each other. These maps are called the "musical isomorphisms", $\flat$ and $\sharp$, as they raise or lower an index in the coordinate representation:

$$\flat : \quad T(M) \to T^*(M) : \quad v^\mu \partial_\mu \mapsto v^\mu g_{\mu\nu} dx^\nu \quad (11)$$
$$\sharp : \quad T^*(M) \to T(M) : \quad V_\mu dx^\mu \mapsto V_\mu g^{\mu\nu} \partial_\nu \quad (12)$$

As an example application, the "gradient" of a scalar function is given by $\nabla f = \sharp df$ using this notation. In Euclidean space, the metric is represented by the identity matrix and the components of vectors are identical to the components of co-vectors. As computer graphics usually is considered in Euclidean space, this justifies the usual negligence of distinction among vectors and co-vectors; consequently graphics software only knows about one type of vectors which is uniquely identified by its number of components. However, when dealing with coordinate transformations or curvilinear mesh types then distinguishing between tangential vectors and co-vectors is unavoidable. Treating them both as the same type within a computer program leads to confusions and is not safe. Section 4 will address this issue.

2.4 Exterior Product

The *exterior product* $\wedge : V \times V \to \Lambda(V)$ (also known as wedge product, Grassmann product, or alternating product) generates vector space elements of higher dimensions from elements of a vector space V by taking the antisymmetric part of the outer product (eq. 8) as

$$u \wedge v = \frac{1}{2} (u \otimes v - v \otimes u) \quad (13)$$

The new vector space is denoted $\Lambda(V)$. With the exterior product, $v \wedge u = -u \wedge v \quad \forall u, v \in V$, which consequently results in $v \wedge v = 0 \ \forall \ v \in V$. The exterior product defines an algebra on its elements, the exterior algebra (or Grassman algebra) [9, 5]. It is a sub-algebra of the Tensor algebra consisting on the anti-symmetric tensors. The exterior algebra is defined intrinsically by the vector space and does not require a metric. For a given n-dimensional vector space V, there can at most be n-th power of an exterior product, consisting of n different basis vectors. The $n + 1$-th power must vanish, because at least one basis vector would occur twice, and there is exactly one basis vector for $\Lambda^n(V)$.

Elements $v \in \Lambda^k(V)$ are called k-vectors, whereby 2-vectors are also called bi-vectors and 3-vectors trivectors. The number of components of an k-vector of an n-dimensional vector space is given by the binomial coefficient $\binom{n}{k}$. For $n = 2$ there are two 1-vectors and one bi-vector, for $n = 3$ there are three 1-vectors, three bi-vectors and one tri-vector. These relationships are depicted by the Pascal's triangle, with the row representing the dimensionality of the underlying base space and the column the vector type:

$$\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array} \quad (14)$$

As can be easily read off, for a four-dimensional vector space there will be four 1-vectors, six bi-vectors, four tri-vectors and one 4-vector. The n-vector of a n-dimensional vector space is also called a *pseudo-scalar*, the $(n-1)$ vector a *pseudo-vector*.

2.5 Visualizing Exterior Products

An exterior algebra is defined on both the tangential vectors and co-vectors on a manifold. A bi-vector v formed from tangential vectors is written in chart as

$$v = v^{\mu\nu} \partial_\mu \wedge \partial_\nu \quad , \quad (15)$$

a bi-covector U formed from co-vectors is written in chart as

$$U = U_{\mu\nu} dx^\mu \wedge dx^\nu \quad . \quad (16)$$

They both have $\binom{n}{2}$ independent components, due to $v^{\mu\nu} = -v^{\nu\mu}$ and $U_{\mu\nu} = -U_{\nu\mu}$ (three components in 3D, six components in 4D). A bi-tangential vector (15) can be understood visually as an (oriented, i.e., signed) plane that is spun by the two defining tangential vectors, independently of the dimensionality of the underlying base space. A bi-co-vector (16) corresponds to the subspace of an n-dimensional hyperspace where a plane is "cut out". In three dimensions these visualizations overlap: both a bi-tangential vector and a co-vector correspond to a plane, and both a tangential vector and a bi-co-vector correspond to one-dimensional direction ("arrow"). In four dimensions, these visuals are more distinct but still overlap: a co-vector corresponds to a three-dimensional volume, but a bi-tangential vector is represented by a plane same as a bi-co-vector, since cutting out a 2D plane from four-dimensional space yields a 2D plane again. Only in higher dimensions these symbolic representations become unique.
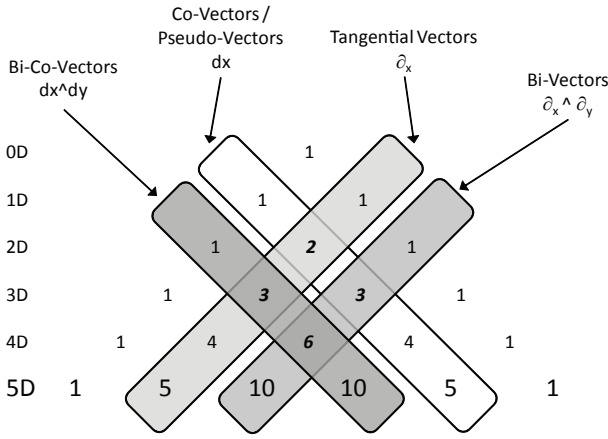
Figure 2: Pascal's triangle showing the location of tangential vectors, bi-vectors, co-vectors and bi-covectors in the various subspaces in different dimensions. Especially in three dimensions there are many overlaps, indicating ambiguities where different quantities are all represented by "just three numbers". Similar situations occur in 4D, only in 5D all vector types become unambiguous.

However, in any case a co-vector and a pseudo-vector will have the same appearance as an $n-1$ dimensional hyperspace, same as a tangential vector corresponds to an pseudo-co-vector:

$$V_\mu dx^\mu \quad \Leftrightarrow \quad v_{\alpha_0\alpha_1\ldots\alpha_{n-1}} \; \partial_{\alpha_0} \wedge \partial_{\alpha_1} \wedge \ldots \partial_{\alpha_{n-1}} \quad (17)$$

$$v^\mu\partial_\mu \quad \Leftrightarrow \quad V_{\alpha_0\alpha_1\ldots\alpha_{n-1}} \; dx^{\alpha_0} \wedge dx^{\alpha_1} \wedge \ldots dx^{\alpha_{n-1}} \quad (18)$$

A tangential vector – lhs of (18) – can be understood as one specific direction, but equivalently as well as "cutting off" all but one $n-1$-dimensional hyperspaces from an n-dimensional hyperspace – rhs of (18). This equivalence is expressed via the interior product of a tangential vector v with an pseudo-co-scalar Ω yielding a pseudo-co-vector V (19), similarly the interior product of a pseudo-vector with an pseudo-co-scalar yielding a tangential vector (19):

$$\iota_\Omega : T(M) \quad \rightarrow \quad (T^*)^{(n-1)}(M) : V \mapsto \iota_\Omega v \quad (19)$$

$$\iota_\Omega : T^{(n-1)}(M) \quad \rightarrow \quad T^*(M) : V \mapsto \iota_\Omega v \quad (20)$$

Pseudo-scalars and pseudo-co-scalars will always be scalar multiples of the basis vectors $\partial_{\alpha_0} \wedge \partial_{\alpha_1} \wedge \ldots \partial_{\alpha_n}$ and $dx^{\alpha_0} \wedge dx^{\alpha_1} \wedge \ldots dx^{\alpha_n}$. However, under when inversing a coordinate $x^\mu \rightarrow -x^\mu$ they flip sign, whereas a "true" scalar does not. An example known from Euclidean vector algebra is the allegedly scalar value constructed from the dot and cross product of three vectors $V(u,v,w) = u\cdot(v \times w)$ which is the negative of when its arguments are flipped:

$$V(u,v,w) = -V(-u,-v,-w) = -u\cdot(-v \times -w) \quad . \quad (21)$$

which is actually more obvious when (21) is written as exterior product:

$$V(u,v,w) = u \wedge v \wedge w = V\partial_0 \wedge \partial_1 \wedge \partial_2 \quad (22)$$

The result (22) actually describes a multiple of a volume element span by the basis tangential vectors ∂_μ - any volume must be a scalar multiple of this basis volume element,

but can flip sign if another convention on the basis vectors is used. This convention depends on the choice of a right-handed versus left-handed coordinate system, and is expressed by the orientation tensor $\Omega = \pm\partial_0 \wedge \partial_1 \wedge \partial_2$. In computer graphics, both left-handed and right-handed coordinate systems occur, which often causes confusion.

By combining (20) and (12) – requiring a metric – we get a map from pseudo-vectors to vectors and reverse. This map is known as the *Hodge star operator* "$*$":

$$* : T^{(n-1)}(M) \rightarrow T(M) : V \longmapsto \sharp\iota_\Omega V \quad (23)$$

The same operation can be applied to the co-vectors accordingly, and generalized to all vector elements of the exterior algebra on a vector space, establishing a correspondence between $k-vectors$ and $n-k$-vectors. The Hodge star operator allows to identify vectors and pseudo-vectors, similarly to how a metric allows to identify vectors and co-vectors. The Hodge star operator requires a metric and an orientation Ω.

A prominent application in physics using the hodge star operator are the Maxwell equations, which, when written based on the four-dimensional potential $A = V_0 dx^0 + A_k dx^k$ (V_0 the electrostatic, A_k the magnetic vector potential) take the form

$$d * dA = J \quad (24)$$

with J the electric current and magnetic flow, which is zero in vacuum. The combination $d * d$ is equivalent to the Laplace operator "$\Box$", which indicates that (24) describes electromagnetic waves in vacuum.

2.6 Geometric Algebra

Geometric Algebra is motivated by the intention to find a closed algebra on a vector space with respect to multiplication, which includes existence of an inverse operation. There is no concept of dividing vectors in "standard" vector algebra. Because the result of the inner and exterior product is of different dimensionality than their operands, they are not suited to define a closed GA on the vector space.

Geometric algebra postulates a product on elements of a vector space $u,v,w \in \mathcal{V}$ that is associative, $(uv)w = u(vw)$, left-distributive $u(v+w) = uv + uw$, right-distributive $(u+v)w = uw + vw$, and reduces to the inner product as defined by the metric $v^2 = g(v,v)$. It can be shown that sum of the exterior product (which, within Geometric Algebra, is also called outer product, but should not be confused with the outer product $\otimes$ on tensors from eq. 8) and the inner product fulfill these requirements; this defines the *geometric product* as the sum of both:

$$uv := u \wedge v + u \cdot v \quad . \quad (25)$$

Since $u \wedge v$ and $u \cdot v$ are of different dimensionality ($\binom{n}{2}$ and $\binom{n}{0}$, respectively), the result must be in a higher dimensional vector space of dimensionality $\binom{n}{2} + \binom{n}{0}$. This space is formed by the linear combination of k-vectors, its elements are called *multivectors*. Its dimensionality is $\sum_{k=0}^{n-1} \binom{n}{k} \equiv 2^n$.

For instance, in two dimensions the dimension of the space of multivectors is $2^2 = 4$. A multivector V, constructed from tangential-vectors on a two-dimensional manifold, is written
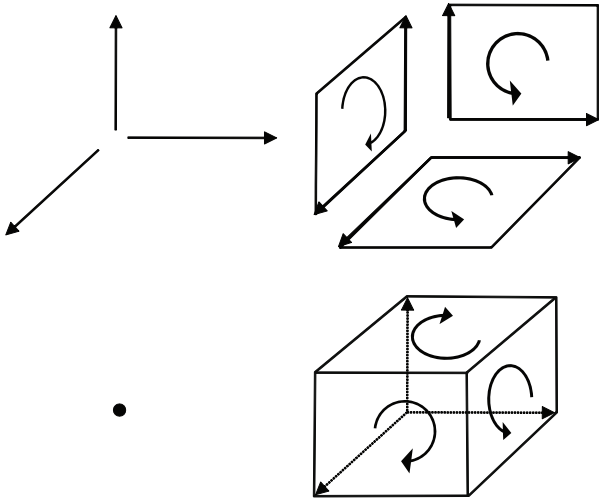
Figure 3: Graphical representation of the $1+3+3+1$ **structure of components that build a 3D multivector: three tangential vectors, three oriented planes, one scalar and one (oriented) volume element.**

as

$$V = V^0 + V^1 \partial_0 + V^2 \partial_1 + V^3 \partial_0 \wedge \partial_1 \qquad (26)$$

with V^μ the four components of the multivector in a chart. For a three-dimensional manifold a multivector on its tangential space has $2^3 = 8$ components and is written as

$$\begin{aligned} V = V^0 +& \\ V^1 \partial_0 &+ V^2 \partial_1 + V^2 \partial_2 + \\ V^4 \partial_0 &\wedge \partial_1 + V^5 \partial_1 \wedge \partial_2 + V^6 \partial_2 \wedge \partial_0 + \\ V^7 \partial_0 &\wedge \partial_1 \wedge \partial_2 \end{aligned} \qquad (27)$$

with V^μ the eight components of the multivector in a chart. The components of a multivector have a direct visual interpretation, which is one of the key features of geometric algebra. In 3D, a multivector is the sum of a scalar value, three directions, three planes and one volume. These basis elements span the entire space of multivectors.

2.7 Spinors and Quaternions

Given a bi-vector $U = u \wedge v$ built from two orthonormal unit vectors u, v (which fulfill $|u| = 1$, $|v| = 1$, $u \cdot v = 0$ under a given metric such that $U = uv$), we find that it provides the same algebraic properties as the imaginary unit $\sqrt{-1}$:

$$U^2 = UU = (uv)(uv) = (uv)(-vu) = -u(vv)u = -1 \quad (28)$$

This is a well known aspect of Geometric Algebra, which leads to n distinct imaginary units on an n-dimensional vector space. For $n = 3$ we have three imaginary units (usually denoted as i, j, k), which relate to the three bi-vectors along the three coordinate axis. These three basis vectors $i = \partial_x \wedge \partial y$, $j = \partial_y \wedge \partial z$, $k = \partial_z \wedge \partial x$ (equivalently to the co-vectors) fulfill $ijk = -1$, which is identical to the definitions used in Quaternion algebra [7]. A quaternion consists of four components, a scalar and "vectorial" part. They represent

the even parts of a multivector in 3D (27):

$$Q = Q^0 + Q^2 \partial_0 \wedge \partial_1 + Q^0 \partial_1 \wedge \partial_2 + Q^1 \partial_2 \wedge \partial_0 \qquad (29)$$

It can be shown that the even multivectors form a closed sub-algebra itself. GA provides a direct geometric insight for quaternions via (29), with the hard-to-memorable quaternion product being immersed within the easily rememberable geometric product. Given an even multivector (29), its dual as provided by the Hodge star operator (23) yields an odd multivector, consisting out of a tangential vector and a pseudo-scalar (i.e., a volume element).

Quaternions are known in computer graphics for implementing rotations (for instance, the `SbRotation` class in OpenInventor), alternatively to rotation matrices (such as used in OpenGL). Same functionality is provided by *rotors* in GA. In 2D the right-multiplication of a vector $v = v^x \partial_x + v^y \partial_y$ with the bi-vector $\partial_x \wedge \partial_y = \partial_x \partial_y$ corresponds to a counter-clockwise rotation:

$$v(\partial_x \wedge \partial_y) = v^x \partial_x (\partial_x \partial_y) + v^y \partial_y (\partial_x \partial_y) = v^x \partial_y - v^y \partial_x \quad (30)$$

Therefore a rotation around an arbitrary angle φ is written as a linear combination of a scalar component and a bi-vector, which is called a *rotor* (or *spinor*):

$$R = \cos \varphi + i \sin \varphi =: e^i \qquad (31)$$

where i is an arbitrary unit bi-vector fulfilling $i^2 = -1$. e is the Euler number used here for defining the exponential function of a bi-vector, in style of the Euler equation. The inverse rotor (implementing clockwise rotation on right-multiplication, or counter-clockwise when applied from the left) is given by inversing the rotation angle

$$R^{-1} = e^{-i} = \cos \varphi - i \sin \varphi \quad . \qquad (32)$$

In two dimensions it is equivalent if some vector v is left-multiplied or right-multiplied with a rotor

$$vR^{-2} \equiv R^2 v \equiv RvR^{-1} \quad , \qquad (33)$$

however in more than two dimensions the symmetric variant RvR^{-1} with multiplying from the left and from the right has to be used to cancel out a tri-vector component that would otherwise occur (from multiplying the vector with a the bi-vector part of the rotor). While Quaternion Algebra is specific to three dimensions, the concept of a rotor in GA is independent from the dimensions and directly applicable to the 4D case, as will be reviewed in the next section.

2.8 Multilinear Multivector Maps

Same as a tensor is a multilinear map of vectors and co-vectors, we may represent multilinear maps of multivectors as a set of numbers given a specific chart. The Riemann tensor R, as described in 3.3, is such a case, is it can be seen as a map from bivectors to bivectors:

$$R : T_p(M) \wedge T_p(M) \to T_p(M) \wedge T_p(M) \qquad (34)$$

The Riemann tensor can then be interpreted as argument of the Lorentz boost $e^{R(U)}$ resulting from a tiny circuit within a plane defined by the bivector U.

3. NEWMAN-PENROSE FORMALISM

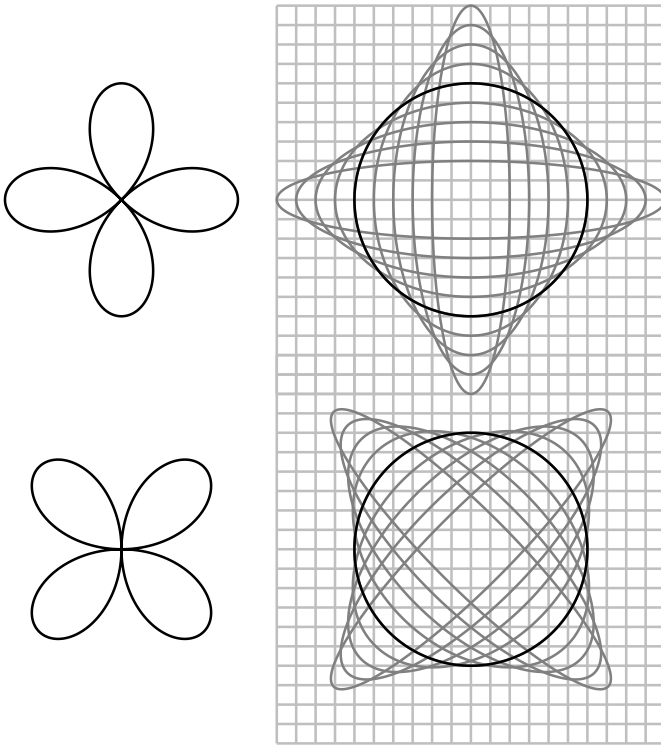General relativity predicts the existence of gravitational waves. There is a huge effort to detect gravitational waves

Figure 4: **The two linear polarizations of gravitational waves. The $+$ polarization (top) has a $\cos 2\chi$ shape about the direction of propagation (into the paper), while the $\times$ polarization (bottom) has a $\sin 2\chi$ shape. A gravitational wave causes a system of freely falling test masses to oscillate relative to a grid of points a fixed proper distance apart.**

expected for example from merging pairs of black holes [1]. To date no gravitational waves have been detected directly. There is however indirect evidence for their existence from the gradual decrease in orbital period of the binary pulsar, which is quantitatively consistent with the general relativistic prediction of energy loss by quadrupole emission of gravitational waves [4, 6].

It is conventional to characterize gravitational waves in terms of their Newman-Penrose (1962) (NP) components [14, 2, 16]. The purpose of this section is to give an idea of how this works, and how the geometric algebra offers insight into the NP formalism. The traditional derivation of the NP components of gravitational waves is magical, and shrouded in unnecessary and misleading notation. As Held (1974) [13] politely puts it, the NP formalism presents "a formidable notational barrier to the uninitiate".

The notion of a gravitational wave can be perplexing. A passing gravitational wave causes the distance between two freeling-falling masses to oscillate. But if gravity affects the very measurement of length itself, how can the distance between the masses be measured? The answer is that, despite the fact that in general relativity spacetime has no absolute existence, in the sense that the choice of coordinate system

is arbitrary, nevertheless the metric asserts that there is a unique proper distance along a given path (or affine distance, along a null path) between any two points in spacetime (such as the path followed by a beam of laser light). The presence of gravity, or curvature, is expressed by the presence of a gravitational force between two points a fixed proper distance apart. A gravitational wave causes an oscillation in the differential gravitational force, or tidal force, between two points a fixed distance apart.

Figure 4 illustrates gravitational waves, in their two possible linear polarizations, $+$ and $\times$. The grid represents a locally inertial system of points a fixed proper distance apart. The superposed ellipses represent a system of freely-falling test masses whose positions, initially on a circle, are being perturbed by a gravitational wave moving in a direction perpendicular to the paper. The proper distance between freely-falling test masses oscillates. That oscillation can be measured for example by the change in the number of wavelengths along a laser beam between the masses.

Sometimes one sees depictions of gravitational waves similar to Figure 4, but with the grid oscillating along with the ellipses. Such depictions are intended to convey the idea that gravitational waves are waves of spacetime (of the metric), but they are misleading, since they suggest that rulers oscillate along with the test masses, which is false.

3.1 Newman-Penrose tetrad

The Newman-Penrose (NP) formalism is particularly well adapted to treating waves that travel at the speed of light, which includes electromagnetic and gravitational waves. The NP formalism starts with the rest frame of an observer, and applies two tricks to it. The axes, or tetrad, of the observer's locally inertial frame form an orthonormal basis of vectors in the geometric algebra

$$\{\boldsymbol{\gamma}_t, \boldsymbol{\gamma}_x, \boldsymbol{\gamma}_y, \boldsymbol{\gamma}_z\} \ , \tag{35}$$

with the metric in Minkowski signature of the form

$$\boldsymbol{\gamma}_m \cdot \boldsymbol{\gamma}_n = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{36}$$

with indices m, n running over t, x, y, z. The NP formalism chooses one axis, typically the z-axis, to be the direction of propagation of the wave.

The first NP trick is to replace the transverse axes $\boldsymbol{\gamma}_x$ and $\boldsymbol{\gamma}_y$ by spinor axes $\boldsymbol{\gamma}_+$ and $\boldsymbol{\gamma}_-$ defined by

$$\boldsymbol{\gamma}_+ \equiv \frac{1}{\sqrt{2}} \left(\boldsymbol{\gamma}_x + I\boldsymbol{\gamma}_y \right) \ , \quad \boldsymbol{\gamma}_- \equiv \frac{1}{\sqrt{2}} \left(\boldsymbol{\gamma}_x - I\boldsymbol{\gamma}_y \right) \ . \tag{37}$$

This is the same trick used to define the spinor components $L_\pm$ of the angular momentum operator $\boldsymbol{L}$ in quantum mechanics.

The second NP trick is to replace the time t and propagation z axes with outgoing and ingoing null axes $\boldsymbol{\gamma}_v$ and $\boldsymbol{\gamma}_u$, defined by

$$\boldsymbol{\gamma}_v \equiv \frac{1}{\sqrt{2}} \left(\boldsymbol{\gamma}_t + \boldsymbol{\gamma}_z \right) \ , \quad \boldsymbol{\gamma}_u \equiv \frac{1}{\sqrt{2}} \left(\boldsymbol{\gamma}_t - \boldsymbol{\gamma}_z \right) \ . \tag{38}$$

The resulting outgoing, ingoing, and spinor axes form a NP null tetrad

$$\{\boldsymbol{\gamma}_v, \boldsymbol{\gamma}_u, \boldsymbol{\gamma}_+, \boldsymbol{\gamma}_-\} \ , \tag{39}$$

with NP metric

$$\boldsymbol{\gamma}_m \cdot \boldsymbol{\gamma}_n = \begin{pmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{40}$$

with indices m, n running over $v, u, +, -$. The NP metric (40) has zeros down the diagonal. This means that each of the four NP axes $\boldsymbol{\gamma}_m$ is null: the scalar product of each axis with itself is zero. In a profound sense, the null, or light-like, character of each the four NP axes explains why the NP formalism is well adapted to treating fields that propagate at the speed of light.

Three kinds of transformation, considered further below, take a particularly simple form in the NP tetrad:

  I Reflections through the transverse axis y;
 II Rotations about the propagation axis z;
III Boosts along the propagation axis z.

3.1.1 Reflections
Under transformation I, a reflection through the y-axis, the spinor axes swap:

$$\boldsymbol{\gamma}_+ \leftrightarrow \boldsymbol{\gamma}_- \ , \tag{41}$$

which may also be accomplished by complex conjugation. Reflection through the y-axis, or equivalently complex conjugation, changes the sign of all spinor indices of a tensor component

$$+ \leftrightarrow - \ . \tag{42}$$

In short, complex conjugation flips spin, a pretty feature of the NP formalism.

3.1.2 Rotations
Under transformation II, a right-handed rotation by angle χ about the direction z of propagation, the transverse axes $\boldsymbol{\gamma}_x$ and $\boldsymbol{\gamma}_y$ transform as

$$\begin{aligned} \boldsymbol{\gamma}_x &\rightarrow \cos\chi \, \boldsymbol{\gamma}_x - \sin\chi \, \boldsymbol{\gamma}_y \ , \\ \boldsymbol{\gamma}_y &\rightarrow \sin\chi \, \boldsymbol{\gamma}_x + \cos\chi \, \boldsymbol{\gamma}_y \ . \end{aligned} \tag{43}$$

It follows that the spinor axes $\boldsymbol{\gamma}_+$ and $\boldsymbol{\gamma}_-$ transform under a right-handed rotation by angle χ as

$$\boldsymbol{\gamma}_\pm \rightarrow e^{\pm I\chi} \, \boldsymbol{\gamma}_\pm \ . \tag{44}$$

The transformation (44) identifies the spinor axes $\boldsymbol{\gamma}_+$ and $\boldsymbol{\gamma}_-$ as having spin $+1$ and -1 respectively. More generally, an object can be defined as having spin s if it varies by

$$e^{sI\chi} \tag{45}$$

under a rotation by angle χ about the direction of propagation. The NP components of a tensor inherit spin properties from that of the spinor basis. The general rule is that the spin s of any tensor component is equal to the number of $+$ covariant indices minus the number of $-$ covariant indices:

  spin s = number of $+$ minus $-$ covariant indices . (46)

3.1.3 Boosts
The final transformation III, a boost along the z-axis, multiplies the outgoing and ingoing axes $\boldsymbol{\gamma}_v$ and $\boldsymbol{\gamma}_u$ by a blueshift factor ϵ and its reciprocal

$$\begin{aligned} \boldsymbol{\gamma}_v &\rightarrow \epsilon \, \boldsymbol{\gamma}_v \ , \\ \boldsymbol{\gamma}_u &\rightarrow (1/\epsilon) \, \boldsymbol{\gamma}_u \ . \end{aligned} \tag{47}$$

If the observer boosts by velocity v in the z-direction away from the source, then the blueshift factor is the special relativistic Doppler shift factor

$$\epsilon = \left(\frac{1-v}{1+v} \right)^{1/2} \ . \tag{48}$$

The exponent n of the power ϵ^n by which an object changes under a boost along the z-axis is called its boost weight. Thus $\boldsymbol{\gamma}_v$ has boost weight $+1$, and $\boldsymbol{\gamma}_u$ has boost weight -1. The NP components of a tensor inherit their boost weight properties from those of the NP basis. The general rule is that the boost weight n of any tensor component is equal to the number of v covariant indices minus the number of u covariant indices:

  boost weight n = number of v minus u covariant indices .
$$\tag{49}$$

3.2 Electromagnetic waves
The properties of gravitational waves are in many ways similar to those of electromagnetic waves. Both kinds of waves are massless, traveling at the speed of light. A crucial difference is that gravitational waves are spin-2 (tensor) waves, whereas electromagnetic waves are spin-1 (vector) waves.

Recall the nature of electromagnetic waves. Electromagnetic waves are characterized by the electromagnetic field F_{ij}, which is an antisymmetric tensor, or bivector, with 6 distinct components. The 6 components are commonly collected into two 3-dimensional vectors, the electric and magnetic fields E and B. The geometric algebra gives the insight that the electromagnetic field tensor, being a bivector, has a natural complex structure, in which the electric and magnetic fields together form a complex 3-vector $E + IB$.

With respect to a NP null tetrad (39), the electromagnetic bivector has 3 complex components, of spin respectively -1, 0, and $+1$, in accordance with the rule (46):

$$\begin{aligned} -1 \ &: \ F_{u-} \\ 0 \ &: \ \tfrac{1}{2}\left(F_{uv} + F_{+-} \right) \\ +1 \ &: \ F_{v+} \ . \end{aligned} \tag{50}$$

The complex conjugates of the 3 components are:

$$\begin{aligned} -1^* \ &: \ F_{u+} \\ 0^* \ &: \ \tfrac{1}{2}\left(F_{uv} - F_{+-} \right) \\ +1^* \ &: \ F_{v-} \ , \end{aligned} \tag{51}$$

whose spins have the opposite sign. Conventionally (Chandrasekhar 1983), the 3 complex spin components of the elec-

tromagnetic field bivector in the NP formalism are denoted

$$
\begin{aligned}
-1 \;&:\; \phi_2 \,, \\
0 \;&:\; \phi_1 \,, \\
+1 \;&:\; \phi_0 \,.
\end{aligned}
\tag{52}
$$

The notation, like much of the rest of conventional NP notation, is truly awful.

For outgoing electromagnetic waves, only the spin -1 component propagates, carrying electromagnetic energy far away from a source:

$$
-1 \;:\; \text{propagating, outgoing} \,.
\tag{53}
$$

This propagating, outgoing -1 component has spin -1, but its complex conjugate has spin $+1$, so effectively both spin components, or helicities, of an outgoing wave are embodied in the single complex component. The remaining 2 complex NP components (spins 0 and 1) of an outgoing wave are short range, describing the electromagnetic field near the source.

Similarly, for ingoing waves, only the spin $+1$ component propagates.

The isolation of each propagating mode into a single complex NP mode, incorporating both helicities, is simpler than the standard picture of oscillating orthogonal electric and magnetic fields.

3.3 Gravitational waves

In electromagnetism, the electromagnetic field tensor is defined by the commutator of the gauge-covariant derivative. In general relativity, the analogous commutator of the covariant derivative is the Riemann curvature tensor R_{klmn}. The Riemann curvature tensor has symmetries which can be designated shorthandly

$$
R_{([kl][mn])} \,.
\tag{54}
$$

Here $[]$ denotes antisymmetry, and $()$ symmetry. The designation (54) thus signifies that the Riemann curvature tensor R_{klmn} is antisymmetric in its first two indices kl, antisymmetric in its last two indices mn, and symmetric under exchange of the first and last pairs of indices, $kl \leftrightarrow mn$. In addition to the symmetries (54), the Riemann curvature tensor has the totally antisymmetric symmetry

$$
R_{klmn} + R_{kmnl} + R_{knlm} = 0 \,.
\tag{55}
$$

The symmetries (54) imply that that the Riemann curvature tensor is a symmetric matrix of antisymmetric tensors, which is to say, a 6×6 symmetric matrix of bivectors. A 6×6 symmetric matrix has 21 independent components. The additional condition (55) eliminates one degree of freedom, leaving the Riemann curvature tensor with 20 independent components.

In spacetime algebra any bivector U (6 component) can be written as complex sum $U = (E + IB)\gamma_t$ of two spatial 3-vectors $E = E^x\gamma_x + E^y\gamma_y + B^z\gamma_z$ and $B = B^x\gamma_x + B^y\gamma_y + B^z\gamma_z$, due to the identity $I\gamma_x\gamma_t \equiv \gamma_y\gamma_z$ etc. In analogy to electromagnetism, $E\gamma_t$ is called the electric bivector, $B\gamma_t$ the magnetic bivector. The Riemann tensor, a multilinear

multimap on bivectors eq. (34), can then be organized into a 2×2 matrix of 3×3 blocks with bivector indices, yielding the structure

$$
\left(
\begin{array}{cc}
R_{EE} & R_{EB} \\
R_{BE} & R_{BB}
\end{array}
\right) \,.
\tag{56}
$$

The condition of being symmetric implies that R_{EE} and R_{BB} are symmetric, while $R_{BE} = (R_{EB})^\top$. The condition (55) states that the 3×3 block R_{EB} (and likewise R_{BE}) is traceless.

The natural complex structure of bivectors in the geometric algebra suggests recasting the 6×6 Riemann curvature matrix (56) into a 3×3 complex matrix, which would have the structure $(R_E + IR_B)(R_E + IR_B)$, or equivalently

$$
R_{EE} - R_{BB} + I(R_{EB} + R_{BE}) \,,
\tag{57}
$$

which is a complex linear combination of the four 3×3 blocks of the Riemann matrix (56). However, it turns out that the complex symmetric 3×3 matrix (57) encodes only part of the Riemann curvature tensor, namely the Weyl tensor. More specifically, the Riemann curvature tensor decomposes into a trace part, the Ricci tensor R_{km}, and a totally traceless part, the Weyl tensor C_{klmn}. The Ricci tensor, which is symmetric, has 10 independent components. The Weyl tensor, which inherits the symmetries (54) and (55) of the Rieman tensor, and in addition vanishes on contraction of any pair of indices, also has 10 independent components. Together, the Ricci and Weyl tensors account for the 20 components of the Riemann tensor. The components of the Ricci and Weyl tensors, though algebraically independent, are related by the differential Bianchi identities.

The end result is that the Weyl tensor, the traceless part of the Riemann curvature tensor, can be written as a 3×3 complex traceless symmetric matrix (57). Such a matrix has 5 distinct complex components.

In empty space (vanishing energy-momentum tensor), the Ricci tensor vanishes identically. Thus the properties of the gravitational field in empty space are specified entirely by the Weyl tensor. In particular, gravitational waves are specified entirely by the Weyl tensor.

When the 5 complex components of the Weyl tensor are expressed in a NP null tetrad (39), the result is 5 complex components, of spins respectively -2, -1, 0, $+1$, and $+2$:

$$
\begin{aligned}
-2 \;&:\; C_{u-u-} \\
-1 \;&:\; C_{uvu-} \\
0 \;&:\; \tfrac{1}{2}\left(C_{vuvu} + C_{vu-+}\right) \\
+1 \;&:\; C_{vuv+} \\
+2 \;&:\; C_{v+v+} \,.
\end{aligned}
\tag{58}
\tag{59}
$$

It can be shown that these 5 complex components exhaust the degrees of freedom of the Weyl tensor.

For outgoing gravitational waves, only the spin -2 component propagates, carrying gravitational waves to far distances:

$$
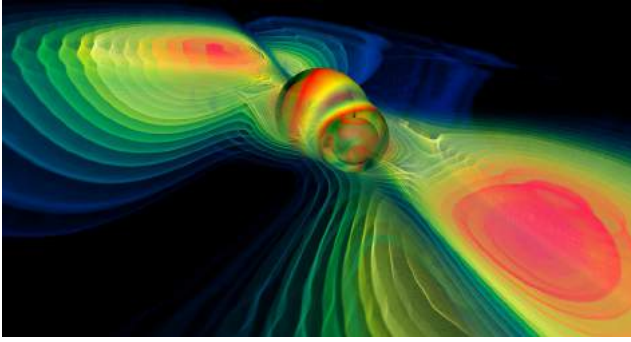-2 \;:\; \text{propagating, outgoing} \,.
\tag{60}
$$

Figure 5: Volume rendering of the gravitational radiation during a binary black hole merger, represented by the real part of Weyl scalar $r \cdot \psi_4$.

This propagating, outgoing -2 component has spin -2, but its complex conjugate has spin $+2$, so effectively both spin components, or helicities, or polarizations, of an outgoing wave gravitational wave are embodied in the single complex component. The remaining 4 complex NP components (spins -1 to 2) of an outgoing gravitational waves are short range, describing the gravitational field near the source.

Conventionally (Chandrasekhar 1983), the 5 complex spin components of the Weyl tensor in the NP formalism are impenetrably denoted

$$
\begin{aligned}
-2 &: \quad \psi_4 , \\
-1 &: \quad \psi_3 , \\
0 &: \quad \psi_2 , \\
+1 &: \quad \psi_1 \\
+2 &: \quad \psi_0 .
\end{aligned}
\tag{61}
$$

Thus the component ψ_4 represents propagating, outgoing gravitational waves. The real part of ψ_4 represents the $\cos(2\chi)$, or $+$, polarization of the propagating gravitational wave, while (minus) its imaginary part represents the $\sin(2\chi)$, or $\times$, polarization, Figure 4. Next time you see an illustration of gravitational waves where the caption says that ψ_4 is plotted, that's what it is (see figure 5). We consider the formulation of the NP scalars as presented here much easier to understand than the usual approach, such as e.g. [16].

4. IMPLEMENTING VECTORS IN C++

As demonstrated in section 2, denoting a vector by just its dimensionality n is insufficient to completely identify its algebraic properties including coordinate transformation rules. Additional information is needed, such as the number of covariant and contra-variance indices.

4.1 Class Hierarchy

Let us denote an array of fixed size N over some type T as `FixedArray<T,N>`, using C++ template notation. No algebraic operation shall be defined on this type, it just serves as a container for numbers, forming an N-tupel of T's. This definition serves as a base class for a type `Vector<T,N>`, which does not add new data members but only adds operators for addition of `Vector<T,N>`'s and multiplication with scalar values, yielding objects of type `Vector<T,N>` again.

$$\texttt{FixedArray<T,N>} \rightarrow \texttt{Vector<T,N>} \tag{62}$$

The resulting class `Vector<T,N>` is a vector in the algebraic sense. It is convenient to make use of matrix algebra in many cases, and since matrices have vector space properties, to express such by deriving the `Matrix` class from the general `Vector` class:

$$\texttt{Vector<T,N*M>} \rightarrow \texttt{Matrix<T,N,M>} \tag{63}$$

The matrix class will add the concept of a matrix product to the general vector space elements. A convenient, though not required, intermediate definition is to define rows and columns – they are rather type definitions than derived classes:

$$\texttt{Matrix<T,1,M>} \quad \rightarrow \quad \texttt{Row<T,M>} \tag{64}$$
$$\texttt{Matrix<T,N,1>} \quad \rightarrow \quad \texttt{Column<T,N>} \tag{65}$$

These definitions of provide a the basis of vector types to be used on the tangential space of a manifold. For a given N,T the following classes are derived:

$$\texttt{FixedArray<T,N>} \quad \rightarrow \quad \texttt{point} \tag{66}$$
$$\texttt{Row<T,N>} \quad \rightarrow \quad \texttt{covector} \tag{67}$$
$$\texttt{Column<T,N>} \quad \rightarrow \quad \texttt{tvector} \tag{68}$$
$$\texttt{Vector} < \texttt{T}, N^2 - N(N+1)/2 > \quad \rightarrow \quad \texttt{bivector} \tag{69}$$
$$\texttt{Vector} < \texttt{T}, 1 + N^2 - N(N+1)/2 > \quad \rightarrow \quad \texttt{rotor} \tag{70}$$
$$\texttt{Vector} < \texttt{T}, 2^N > \quad \rightarrow \quad \texttt{mulvector} \tag{71}$$

The definition of (67) and (68) directly implements the duality relationship (6) in a type-safe way. Tangential vectors and co-vectors both have vector space properties by virtue of (63), but are different types, yet with the property that their product (inherited from the matrix product) yields a scalar. A `point` (66) by itself has no algebraic properties, it only provides coordinates. However, the difference between two points is to be defined to yield a tangential vector (68). On `tvector`s and `covector`s usual matrix operations are inherently defined, so existing algorithms – that are usually provided using matrix algebra – can still be applied to them. However, objects that directly implement operations from Geometric Algebra such as `bivector`, `rotor` and `multivector` are safe from being used as parameters to matrix algebra, yet they inherit vector space properties. We can not show the actual implementation of the operations here due to space limitations; it is sufficient to emphasize that, by using C++ operator overloading, the API can be made very close to the mathematical notation. In addition it is convenient to overload the function call operator "`()`" for `rotor` objects to denote them to be applied to a vector object, meaning "$R(v)$" $:= R v R^{-1}$. This operator will be used in the following code excerpts.

4.2 Camera Navigation using GA

A "camera" in the Vish [8] visualization framework is defined through an observer's location P, a point that is looked at L, and an horizontal view plane, which is given as a bi-vector U corresponding to the "upwards" direction. The difference $t = L - P$ gives the view direction, a tangential vector.

One algorithm for camera navigation is to rotate the camera by an angle φ horizontally around the point of interest L

and by an angle ϑ "upwards" along the line of sight. This algorithm is easily expressed in terms of geometric algebra. First we define the view plane V as

$$V := t \wedge *U \tag{72}$$

and then construct two rotors, a horizontal one and a vertical one

$$R_H := e^{U/|U|\ \varphi} \tag{73}$$
$$R_V := e^{V/|V|\ \vartheta} \tag{74}$$

Now the camera motion is achieved by computing the new observer location by adding the rotated view direction to the point of interest:

$$P_{new} = L + (R_H R_V)(t) \tag{75}$$

Finally, the horizontal view plane needs to be adjusted as well by the vertical rotation

$$U_{new} = R_V U \tag{76}$$

This algorithm can directly be implemented in six C++ source code statements:

```
void Rotate(Camera&TheCamera,
            double phi, double theta)
{
tvector t = TheCamera.Observer - TheCamera.LookAt;

bivector VerticalPlane = (t ^ *TheCamera.Up).unit();

rotor HorizontalRotation = exp(TheCamera.Up , phi),
      VerticalRotation   = exp(VerticalPlane, theta);

      TheCamera.Up *= VerticalRotation;

      TheCamera.Observer = TheCamera.LookAt +
          (VerticalRotation*HorizontalRotation)( t );
}
```

Another algorithm will rotate the camera around the view direction. This is trivial to implement, since we just need the rotor R_t that corresponds to the view direction, which is given by the exponential of from the dual of the sight vector (a bi-vector),

$$R_t = e^{\varphi*(P-L)/|P-L|} \quad , \tag{77}$$

and apply this to the camera's Up-bivector to rotate it. The corresponding C++ source code is accordingly simple:

```
double RotateAroundViewdir(Camera&theCamera, double phi)
{
tvector t = (Camera.Observer - Camera.LookAt).unit();
rotor ViewRotor = exp(*t, phi);
      Camera.Up = ViewRotor( Camera.Up );
}
```

This formulation is considered to be much simpler than an equivalent formulation using matrices and objects like "axial vectors". Using the operations and involved objects is very intuitive once their meaning in the Geometric Algebra has become clear.

4.3 Relativistic observers in the BHFS

4.3.1 The BHFS

The Black Hole Flight Simulator (BHFS) is general relativistic software that can be used to visualize black holes. The BHFS remains work in progress, but has already been used in a number of productions, including the large-format high-resolution dome show "Black Holes: The Other Side of Infinity" (2006, Denver Museum of Nature and Science), and the TV documentaries "Monster of the Milky Way" (2006, NOVA-PBS), and "Monster Black Hole" (2008, Naked Science series, National Geographic). Figure 6 illustrates three frames from a sequence rendered for the National Geographic documentary.

The BHFS provides a complete implementation of the Reissner-Nordström geometry of a charged black hole, including its analytic connections inside the horizon to wormholes, white holes, and other universes. Real astronomical black holes probably have little charge, but they probably do rotate rapidly. A charged black hole is often taken as a surrogate for a rotating black hole, since the interior structure of a spherical charged black hole resembles that of a rotating black hole, but is much easier to model.

The Reissner-Nordström geometry, like its rotating counterpart the Kerr-Newman geometry, is subject to the relativistic counter-streaming instability at the inner horizon first pointed out by Poisson & Israel (1990) [15], and called by them "mass inflation" (see Hamilton & Avelino 2009 [12] for a review). The inflationary instability is expected to eliminate the wormhole and white hole connections inside realistic (astronomical) black holes.

4.3.2 Lorentz rotors in the BHFS

In addition to volume-rendering, the BHFS implements quasi-rigid objects, called "Ships", which by default move along geodesics in the black hole geometry. The camera (observer) is attached to one of the Ships. The orientation and motion of the camera are defined by a Lorentz transformation (which includes both a spatial rotation and a Lorentz boost), or equivalently, by a Lorentz rotor.

A Lorentz rotor R is a unimodular member of the even elements of the spacetime algebra. A Lorentz rotor can be written

$$R = e^{\theta} \tag{78}$$

where θ is a bivector in the spacetime algebra. The corresponding inverse Lorentz rotor is the reverse $\bar{R}$

$$\bar{R} = e^{-\theta} \ . \tag{79}$$

The condition of being unimodular means $\bar{R}R = 1$.

The even spacetime algebra is isomorphic to the algebra of complex quaternions, also called biquaternions. A complex quaternion can be written

$$q = q_R + Iq_I \tag{80}$$

where q_R and q_I are two real quaternions comprising the real and imaginary parts of the complex quaternion q

$$q_R = ix_R + jy_R + kz_R + w_R \ , \quad q_I = ix_I + jy_I + kz_I + w_I \ . \tag{81}$$
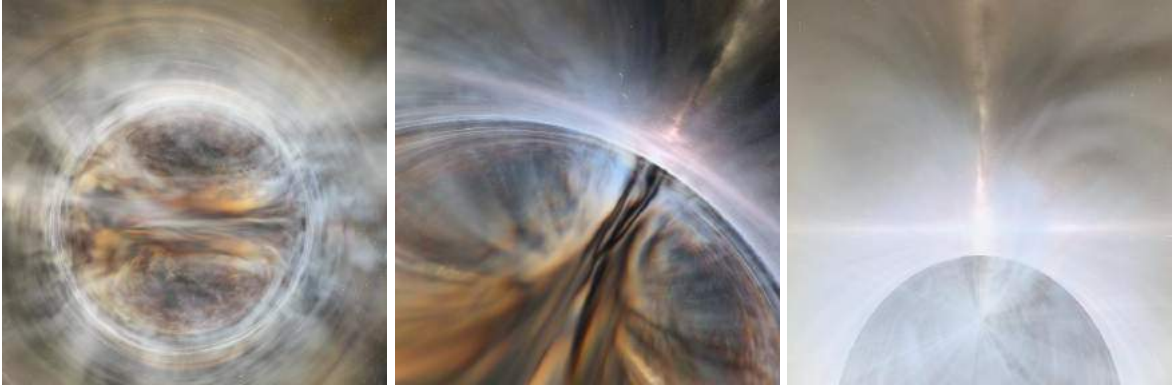
Figure 6: Three frames from a 3000-frame general relativistic volume-rendering with the BHFS of a general relativistic magnetohydrodynamic supercomputer simulation of a disk and jet around a black hole (John Hawley, 2007, private communication). The three frames show, from left to right, (a) outside the black hole, (b) passing through the black hole's outer horizon, (c) hitting the black hole's inner horizon, where the infinite blueshift and energy density triggers the mass inflation instability (Poisson & Israel 1990). The background texture was created from a 3D model of the Milky Way by Donna Cox's team at NCSA. The sequence was prepared for "Monster Black Hole", an episode of National Geographic's Naked Science series.

The imaginary I is the pseudoscalar of the spacetime algebra. It commutes with the quaternionic imaginaries i, j, k. The quaternionic imaginaries themselves satisfy

$$i^2 = j^2 = k^2 = -1 \ , \quad ijk = 1 \ , \qquad (82)$$

from which it follows that the quaternionic imaginaries anticommute between each other, for example $ij = -k = ji$. The convention $ijk = 1$, equation (82), agrees with the convention for quaternions in OpenGL, but is opposite to William Rowan Hamilton's carved-in-stone convention $ijk = -1$. In OpenGL, rotations accumulate to the right: a rotation $R = R_1 R_2$ means rotation R_1 followed by rotation R_2.

The BHFS stores a complex quaternion q as an 8-component object

$$q = \left(\begin{array}{cccc} x_R & y_R & z_R & w_R \\ x_I & y_I & z_I & w_I \end{array} \right) \ . \qquad (83)$$

The reverse $\bar{q}$ of the complex quaternion q is its quaternionic conjugate

$$\bar{q} = \left(\begin{array}{cccc} -x_R & -y_R & -z_R & w_R \\ -x_I & -y_I & -z_I & w_I \end{array} \right) \ . \qquad (84)$$

The group of Lorentz transformations, or Lorentz rotors, corresponds to complex quaternions of unit modulus. The unimodular condition $\bar{R}R = 1$, a complex condition, removes 2 degrees of freedom from the 8 degrees of freedom of complex quaternions, leaving the Lorentz group with 6 degrees of freedom, which is as it should be.

Spatial rotations correspond to real unimodular quaternions, and account for 3 of the 6 degrees of freedom of Lorentz transformations. A spatial rotation by angle θ right-handedly about the x-axis is the real Lorentz rotor

$$R = \cos(\theta/2) + i \sin(\theta/2) \ , \qquad (85)$$

or, stored as a complex quaternion,

$$R = \left(\begin{array}{cccc} \sin(\theta/2) & 0 & 0 & \cos(\theta/2) \\ 0 & 0 & 0 & 0 \end{array} \right) \ . \qquad (86)$$

Lorentz boosts account for the remaining 3 of the 6 degrees of freedom of Lorentz transformations. A Lorentz boost by velocity v, or equivalently by boost angle $\theta = \operatorname{atanh}(v)$, along the x-axis is the complex Lorentz rotor

$$R = \cosh(\theta/2) + Ii \sinh(\theta/2) \ , \qquad (87)$$

or, stored as a complex quaternion,

$$R = \left(\begin{array}{cccc} 0 & 0 & 0 & \cosh(\theta/2) \\ \sinh(\theta/2) & 0 & 0 & 0 \end{array} \right) \ . \qquad (88)$$

4.3.3 Simplicity of Lorentz rotors

The advantages of quaternions for implementing spatial rotations are well-known to 3D game programmers. Compared to standard rotation matrices, quaternions offer increased speed and require less storage, and their algebraic properties simplify interpolation and splining.

Complex quaternions retain similar advantages for implementing Lorentz transformations. They are fast, compact, and straightforward to interpolate or spline.

Under a spacetime rotation by Lorentz rotor R, a general multivector a in the spacetime algebra transforms as

$$a \rightarrow \bar{R}aR \ . \qquad (89)$$

A general such multivector in the spacetime algebra is a 16-component object, with 8 even components, and 8 odd components.

As remarked earlier, the 8-component even spacetime subalgebra is isormorphic to the algebra of complex quaternions. As an example, the electromagnetic field constitutes a 6-component bivector, an even element of the spacetime algebra. The electric and magnetic fields E and B can be

encoded as the complex quaternion

$$F = \begin{pmatrix} E_x & E_y & E_z & 0 \\ B_x & B_y & B_z & 0 \end{pmatrix} . \qquad (90)$$

The transformation (89) then becomes

$$F \to \bar{R}FR , \qquad (91)$$

which is a powerful and elegant way to Lorentz transform the electromagnetic field. The electromagnetic field F in the transformation (91) is the complex quaternion (90), and the rotor R is another complex quaternion, so the Lorentz transformation (91) amounts to multiplying 3 complex quaternions, a one-line expression in a c++ program.

The most common need in the BHFS is to Lorentz transform odd multivectors, not even multivectors. For example, every point on a scene that an observer sees is represented by the energy-momentum 4-vector of a photon emitted by the point and observed by the observer. Each such 4-vector $a = a^m \gamma_m$ is an odd multivector in the spacetime algebra. A general odd multivector is a sum of a vector part a and a pseudovector part Ib. The odd multivector can be written as a product of γ_t (the time basis element of the spacetime algebra) and an even multivector q

$$a + Ib = \gamma_t q \qquad (92)$$

where q is the even multivector, or complex quaternion,

$$q = \begin{pmatrix} -b^x & -b^y & -b^z & a^t \\ a^x & a^y & a^z & b^t \end{pmatrix} . \qquad (93)$$

The Lorentz transformation (89) implies $\gamma_t q \to \bar{R}\gamma_t qR = \gamma_t \bar{R}^* qR$, where $*$ denotes complex conjugation with respect to the peudoscalar imaginary I. It follows that the complex quaternion q, equation (93), transforms as

$$q \to \bar{R}^* qR . \qquad (94)$$

The transformation (94) of the complex quaternion (93) provides a simple and elegant way to Lorentz transform a 4-vector a^m and 4-pseudovector Ib^m. Since b^m (without the I factor) is just another 4-vector, the transformation (94) effectively transforms two 4-vectors, a^m and b^m, simultaneously. The transformation (94) amounts to multiplying 3 complex quaternions, a one-line expression in a c++ program.

5. VECTORS ON THE HARD DISK
5.1 Meta-Data on Vector Types

Storing a specific vector on hard disk, entails storing its numerical representation in a chosen coordinate system. However, when reading an unkown object from disk, solely the information on its numerical representation is insufficient to know what kind of vector it might be. We need some metadata, additional information about the data itself, that tells what properties the object on disk has.

Within a C++ program, this meta-information is available via the `typeid` function of a type. For instance, it allows to distinguish between a `FixedArray<3,double>` and a `Vector<3,double>`, because `typeid(FixedArray<3,double>) != Vector<3,double>`, even though the memory layout of both types is exactly the same. However, the function value of `typeid` cannot be stored to disk – it is a compiler-internal property that makes only sense at runtime for this specific compiler.

We therefore need to assign certain properties to a type that are associated with its algebraic properties. These properties must not be stored with the vector type itself for performance reasons. They could be stored within a class as `enum`s, `typedef`s or static member functions, thereby not requiring memory for the actual numerical type. An alternative technique is to associate information to a type via C++ type trait templates. This technique, common in C++ template meta-programming [19], allows to specify information about a type independently from this type, thereby achieving some encapsulation between the original type and the meta-information about it. Type traits are templates that are specialized for known types and provide information on these types without the need to modify the type itself. They can be applied to native types as well as user-defined types, and including to types that are defined externally, for example by a library. They can be added independently to an existing type. An example of a type trait definition is given in the following code excerpt:

```
template <class Type> struct MetaInfo;

template <>
struct MetaInfo<double>
{   enum   { SIZE = 1 } };

template <int N>
struct MetaInfo<FixedArray<N, double> >
{   enum   { SIZE = N }   };
```

The type trait `MetaInfo` associates an integer value `SIZE` with an arbitrary type `Type`. This information is available at compile-time, and can be reduced to an usual integer in a template class at any time, such as in:

```
template <class Type>
int NumberOfElements(const Type&T)
{
    return MetaInfo<T>::SIZE;
}
```

Note that a type trait class may also specify default values (by specifying a non-specialized definition) and can be functions on template types itself (as demonstrated in the second specialization). This mechanism allows to equip existing types, e.g. as provided by external libraries, with meta-information as required for our framework.

The objective is to specify complete meta-information about a "vector space element" as required to uniquely identify it. As introduced in section 2, such information includes a reference to the metric (or metric field) and the orientation form ι, to know perform the correct algebraic operations. This information can be provided via a "coordinate system", which can be a global type definition – not more than providing the implicit knowledge on how to perform these operations, such as in Euclidean space. In such a case, no memory or computational resources are implied, but another type definition could require explicit formulae for expressions that

are implicit in Euclidean space. Such a chart object may be expressed via a convention on how the coordinate functions are named, for instance $\{x, y, z\}$ for Cartesian coordinates versus $\{r, \vartheta, \varphi\}$ for polar coordinates. While this is yet work in progress, the following quantities have been found to be required for at least basic distinction and identification of vector types:

I   *multiplicity*: an integer value expressing the number of components of this type.

II *rank*: the power $k = a + b$ of the vector space in terms of the tangential space $T^a(M) \times (T^*)^b(M)$; it is the dimensionality of the index space when considering the vector type as an array: zero indicates a scalar type, one is a one-dimensional vectorial type (tangential vector, co-vector, pseudo-vector, pseudo-covector), two are objects representable as matrix, etc.

III *grade*: for quantities from geometric algebra, specifies the grade k of the k-vector; the default is zero, for instance for symmetric tensor fields. For example, a bivector in 3D will have a grade of 2 whereas its rank is 1.

IV *dimensions*: the dimensionality n of the n-dimensional manifold on which this vector type is attached.

V   *coordinatename(i)*: textual functions specifying the naming convention for each of the n coordinate functions.

VI *covariance(i)*: for each index, a flag specifying whether the index is an upper index or lower index. It can be implemented via some function that returns true or false for each index; this function may be evaluated fully at compile-time (a template function that is known) or via lookup into some static array.

VII *symmetries(n)*: often, tensors have symmetric or antisymmetric index pairs. For efficiency reasons it is then important to calculate and store only a minimum subset of the components. This can be implemented via two lookup tables: one table lists those components which are actually stored, the other table contains the prescription for obtaining each tensor component. In a simple scheme, each tensor component is either stored, or is the negative of a stored component, or is zero. (See tables 1 and 2 for examples.) More complex schemes also allow cyclic symmetries, where tensor components can be linear combinations of stored components.

VIII *coordinate systems(i)*: tensor components are only defined with respect to a particular coordinate system. It is necessary to store (for each index) the name of the associated coordinate system. There are objects, such as basis systems or operators that transform between different coordinate systems, where different tensor indices correspond to different coordinate systems.

These properties have been chosen such that some operations on the given types can also succeed with partial knowledge, since certain algorithms do not require full knowledge of the entire algebraic operations of all types.

List of stored components mapping the component name to each storage index:

[0]	[1]	[2]	[3]	[4]	[5]
g_{xx}	g_{xy}	g_{xz}	g_{yy}	g_{yz}	g_{zz}

Obtaining tensor components from stored components via prescription for each entry:

g_{xx}	g_{xy}	g_{xz}	g_{yx}	g_{yy}	g_{yz}	g_{zx}	g_{zy}	g_{zz}
+[0]	+[1]	+[2]	+[1]	+[3]	+[4]	+[2]	+[4]	+[5]

Table 1: Storing a symmetric 3×3 tensor: The component table works like a pointer to the stored components.

List of stored components, mapping the component name to each storage index:

[0]	[1]	[2]
B_{xy}	B_{xz}	B_{yz}

Obtaining tensor components from stored components via prescription for each entry:

B_{xx}	B_{xy}	B_{xz}	B_{yx}	B_{yy}	B_{yz}	B_{zx}	B_{zy}	B_{zz}
0	+[0]	+[1]	−[0]	0	+[2]	−[1]	−[2]	0

Table 2: Storing an antisymmetric 3×3 tensor: The component table defines also signs during dereferencing, or in general, a polynomial expression of components.

This list of "vector properties" is not claimed to be complete; it is an early attempt to find a comprehensive scheme to cover all geometric and algebraic quantities that occur when performing numerical computations on manifolds. Special attention must also be given to the case of non-tensorial quantities such as Christoffel symbols, which do not yet fit into this ontology.

The Cactus framework [11, 3] currently uses a scheme that is simpler than the above; it is based on tensor algebra only and does not support *grades*. However, it does offer support for tensor densities (by associating a *weight* with each quantity), and it handles also certain special non-tensorial objects, such as logarithms of scalar densities and Christoffel symbols. These special cases are handled as exceptions; there is no generic scheme for them. This scheme is mostly used for symmetry conditions, which require either reflecting (mirroring) or rotating tensors. These operations require only the *symmetry* information above.

What is left is a sufficiently powerful I/O layer that allows to store and retrieve this meta-information persistently on disk, such that a set of pure numbers can be identified for their algebraic properties.

5.2 Storing Vector Types in HDF5

HDF5[17] is a generic scientific data format with supporting software, primarily an API provided in C. An HDF5

file can be viewed as a container, in which data objects are organized in ways that are meaningful and convenient to an application. HDF5 can be seen as a framework, rather than a specific format itself, allowing adaption to the various needs of diverse scientific domains [10]. The basic HDF5 object model is relatively simple, yet extremely versatile in terms of the types of data that it can store. The model contains two primary objects: groups, and datasets. Groups provide the organizing structures, and datasets are the basic storage structures. HDF5 groups and datasets may also have associated attributes, which are small data objects for storing metadata defined by applications.

HDF5 allows the specification of user-defined types that shall be stored in a file via its H5T API [18]. For instance, a struct in C/C++ of the form

```
struct CartesianVector
{
    double x,y,z;
};
```

can be expressed in the H5T API as *compound type*:

```
hid_t id = H5Tcreate(H5T_COMPOUND,
                     sizeof(CartesianVector) );
H5Tinsert( id, "x", 0, H5T_DOUBLE);
H5Tinsert( id, "y",   sizeof(double), H5T_DOUBLE);
H5Tinsert( id, "z", 2*sizeof(double), H5T_DOUBLE);
```

This code fragment creates an HDF5 identifier `id` that represents a type of the memory layout as in the aforementioned structure definition. This functionality provides an implementation of the component storage indices as used in table 1 and 2. More details can be found in the HDF5 reference manual.

When writing or reading a dataset to disk, the HDF5 API requires a type identifier to be specified with a `void*`. This tells the HDF5 library how to interpret some chunk of memory. Various generic tools exist to investigate the contents of an HDF5 file, which has a structure of a file system itself. "Datasets" play the role of a file, "Groups" the role of a directory. The tool `h5ls` – part of the HDF5 distribution – lists the contents of an HDF5 file in the fashion of the Unix tool `ls`, enhanced with additional information about the *type* of a dataset. The following example shows how a three-dimensional dataset `CartesianVector data[5][13][9];` appears in this file listing (shortened as compared with actual output):

```
/Block00001 Dataset {5/5, 13/13, 9/9}
    Location:  1:15768
    Links:     1
    Storage:   7020 allocated bytes
    Type:      struct {
                   "x"      +0      native float
                   "y"      +4      native float
                   "z"      +8      native float
               } 12 bytes
    Data:
       (0,0,0) {0.210951, -0.0406732, 0.0611351},
```

```
              {0.210204, -0.0443333, 0.0611199},
              {0.209324, -0.0483009, 0.0611070},
              {0.208286, -0.0525892, 0.0610958},
      (0,0,4) {0.207065, -0.0571980, 0.0610863},
              {0.205640, -0.0621138, 0.0610815},
```

By virtue of HDF5, we can easily attach *names* to the purely numerical values in the data field. Hereby the HDF5 library offers various features that are very useful in practice, such as not only taking care of conversions between big-endian and little-endian platforms, but also conversions from double to float component types as well as transformations between different layouts such as $\{x, y, z\} \Leftrightarrow \{z, x, y\}$.

The availability of a naming scheme attached to numerical values is already sufficient to identify a coordinate system that is supposed to be "attached" to these numbers, in spirit of 5.1, V. Knowing the coordinate system relative to which the numbers are stored, in addition we need to specify the various attributes defining the algebraic properties of this vector type HDF5 allows to attach attributes with a dataset, group or "named data type". A named data type is a type id that was created by the `H5Tcreate()` call but saved to disk. It needs to be associated with a group in the file. Attributes attached to such a named data type are shared among all data sets of this type – the data type acts like a pointer to a common location of a set of attributes. We now need to define an HDF5 type for each of the vector types as defined from the meta-information about a specific data type. The following HDF5 listing shows the created named type, stored in a group `/Charts/Cartesian3D`, as it is named "Point" and equipped with an integer telling this data type refers to a manifold of dimension three. This data type "Point" is then later used to declare a dataset of points (shown with two attributes denoting the name of the associated chart and the dimension of the related manifold):

```
/Charts/Cartesian3D/Point Type
    Attribute: ChartDomain scalar
        Type:  null-terminated ASCII string
        Data:  "Cartesian3D"

    Attribute: Dimensions scalar
        Type:      native int
        Data:  3
    Type:      shared-1:13328 struct {
                   "x"      +0      native float
                   "y"      +4      native float
                   "z"      +8      native float
               } 12 bytes

/Block00001 Dataset {5/5, 13/13, 9/9}
    Location:  1:15768
    Links:     1
    Storage:   7020 allocated bytes
    Type:      { shared-1:13328} struct {
                   "x"      +0      native float
                   "y"      +4      native float
                   "z"      +8      native float
               } 12 bytes
    Data:
```

This scheme allows to identify the dataset named "Blocks" as representing Cartesian coordinates of point locations. Accessing the dataset "Blocks" during reading, the software

application can easily check for the attributes of the dataset to retrieve its algebraic properties. However, doing so is *optional*. Many applications might not implement the full set of tensor algebra, but might still provide a set of useful operations – such as displaying a dataset numerical as a spreadsheet etc. The information that the dataset consists of three floating point numbers, the only information required for a generic operation such as displaying as spreadsheet, is immediately available, more complex properties require further lookup.

This naming scheme is work in progress and not yet implemented or available in its full generality. Various questions have yet to be addressed, such as a generic naming scheme for types or the specification of multivectors. For the latter, one might utilize the HDF5 feature that a compound type may contain other compound types as well. If such is the appropriate solution here, will be subject of further investigation.

5.3 Storing Multi-Vector Types in HDF5

Multivectors are linear combinations of vectors of different basis elements, thereby forming an higher-dimensional space. A similar functionality is achieved using HDF5 by creating compound types from the basic vector types. For instance, given a bivector type in 3D, created by HDF5 API calls of the form

```
hid_t   bivector3D_id  =
        H5Tcreate( H5T_COMPOUND, 3*sizeof(double) );

H5Tinsert( bivector3D_id, "yz",  0, H5T_NATIVE_DOUBLE);
H5Tinsert( bivector3D_id, "zx",  8, H5T_NATIVE_DOUBLE);
H5Tinsert( bivector3D_id, "xy", 16, H5T_NATIVE_DOUBLE);
```

we may create a rotor in the following as compound containing the bivector, and adding a scalar:

```
hid_t   rotor3D_id   = H5Tcreate( H5T_COMPOUND, 32 );

H5Tinsert( rotor3D_id, "cos", 0, H5T_NATIVE_DOUBLE);
H5Tinsert( rotor3D_id, "sin", 8, bivector3D_id);
```

We name the scalar and bivector component "cos" and "sin" here, inspired by the construction of a rotor. What naming scheme to use here in general, will yet need to be explored. It is now a nice feature of HDF5 that different storage schemes are automatically mapped, i.e. datasets stored as the following type

```
hid_t antirotor3D_id =
H5Tcreate( H5T_COMPOUND, 4*sizeof(double) );
H5Tinsert( antirotor3D_id, "sin", 0 , bivector3D_id);
H5Tinsert( antirotor3D_id, "cos", 24, H5T_NATIVE_DOUBLE);
```

can be directly read without further specific treatment as a `rotor3D_id` dataset. This way HDF5 easily provides the notion of $a + c \wedge b \equiv c \wedge b + a$, i.e., commutativity of the "+" operator. One can also define a type which only retrieves the bivector component of a dataset of rotors, or the scalar component. This functionality is already provided by HDF5.

The specification of maps on multivectors, section 2.8, appears non-trivial, due to the many symmetry conditions that occur in these cases. For instance, if the Riemann tensor as in 3.3 would be stored by each of its tensor components, this results in 256 values (at each point). However, only 20 need to be stored, and under certain conditions (such as matter-free spacetime) that may be known in advance, only 10 . A smart type definition system that is able to express such properties yet has to be developed. Symmetry tables such as discussed in 5.1 might be a way to go, and a formulation of those as attributes on HDF5 types will be developed.

6. CONCLUSION

In this article we have reviewed the various types of what is usually called a "vector" in the context of differential geometry and geometric algebra. Various algebraic types have been identified, which are all represented numerically by three floating point numbers in three dimensions: tangential vectors, co-vectors, bi-vectors and bi-co-vectors. Yet these four different types have distinct algebraic properties and should be distinguished. We demonstrated the application of diverse vector types in four dimensions, leading to an easier formulation of the Newmann-Penrose formalism by virtue of Geometric Algebra. The clarity of the diverse algebraic types as achieved via GA thereby eases "navigation" in Riemann space, computer graphic applications (where two examples are given), and identification of quantities stored in files.

7. REFERENCES

[1] LIGO: Laser Interferometer Gravitational Wave Observatory, URL http://www.ligo.caltech.edu/.

[2] GRwiki: a repository of basic definitions and formulas for gravitational physics, URL http://grwiki.physics.ncsu.edu/.

[3] Cactus Computational Toolkit home page, URL http://www.cactuscode.org/.

[4] Binary pulsar. Wikipedia, 2009. URL http://en.wikipedia.org/wiki/Binary_pulsar.

[5] Exterior algebra. Wikipedia, 2009. URL http://en.wikipedia.org/wiki/Exterior_algebra.

[6] The Nobel Prize in physics 1993. Wikipedia, 2009. URL http://nobelprize.org/nobel_prizes/physics/laureates/1993/illpres/discovery.html.

[7] Quaternion. Wikipedia, 2009. URL http://en.wikipedia.org/wiki/Quaternion.

[8] W. Benger, G. Ritter, and R. Heinzl. The Concepts of VISH. In 4th *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007*, pages 26–39. Berlin, Lehmanns Media-LOB.de, 2007.

[9] A. Bossavit. Differential geometry for the student of numerical methods in electromagnetism. Technical report, Tampere University of Technology, 1991. URL http://butler.cc.tut.fi/~bossavit/.

[10] M. T. Dougherty, M. J. Folk, E. Zadok, H. J. Bernstein, F. C. Bernstein, K. W. Eliceiri, W. Benger, and C. Best. Unifying biological image formats with hdf5. *Communications of the ACM (CACM)*, 52(10):42–47, October 2009.

[11] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf. The Cactus framework and toolkit: Design and applications. In

Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science, Berlin, 2003. Springer.

[12] A. J. S. Hamilton and P. P. Avelino. The physics of the relativistic counter-streaming instability that drives mass inflation inside black holes. *Physics Reports*, accepted, 2009. gr-qc/0811.1926.

[13] A. Held. A formalism for the investigation of algebraically special metrics. i. *Commun. Math. Phys.*, 37:311–26, 1974.

[14] E. T. Newman and R. Penrose. An approach to gravitational radiation by a method of spin coefficients. *J. Math. Phys.*, 3:566–79, 1962.

[15] E. Poisson and W. Israel. Inner-horizon instability and mass inflation in black holes. *Phys. Rev.*, D41:1796–1809, 1990.

[16] E. (Ted) Newman and R. Penrose. Spin-coefficient formalism. *Scholarpedia*, 4(6):7445, 2009. URL `http://www.scholarpedia.org/article/ Newman-Penrose_formalism`.

[17] The HDF Group. Hierarchical data format version 5. `http://www.hdfgroup.org/HDF5`, 2000-2009.

[18] The HDF Group. HDF5 H5T API, 2009. URL `http://www.hdfgroup.org/HDF5/doc/RM/RM_H5T.html`.

[19] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

Evolving Time Surfaces in a Virtual Stirred Tank

Bidur Bohara
Farid Harhad
Department of Computer Science
Louisiana State University
Baton Rouge, LA-70803
bbohar1@tigers.lsu.edu,
fharhad@cct.lsu.edu

Werner Benger
Center for Computation &
Technology
Louisiana State University
Baton Rouge, LA-70803
werner@cct.lsu.edu

Nathan Brener
S. Sitharama Iyengar
Bijaya B. Karki
Department of Computer Science
Louisiana State University
Baton Rouge, LA-70803
brener@csc.lsu.edu
iyengar@csc.lsu.edu,karki@csc.lsu.edu

Marcel Ritter
Department of Computer Science
University of Innsbruck
Technikerstrasse 21a
A-6020 Innsbruck, Austria
marcel.ritter@student.uibk.ac.at

Kexi Liu, Brygg Ullmer
Center for Computation &
Technology
Department of Computer Science
Louisiana State University
Baton Rouge, LA-70803
kliu9@lsu.edu, ullmer@lsu.edu

Nikhil Shetty, Vignesh Natesan,
Carolina Cruz-Neira
Center for Adv. Comp. Studies
University of Louisiana at
Lafayette
Lafayette, LA 70504
nikhil.j.shetty@gmail.com

ABSTRACT
The complexity of large scale computational fluid dynamic simulations demand powerful tools to investigate the numerical results. Time surfaces are the natural higher-dimensional extension of time lines, the evolution of a seed line of particles in the flow of a vector field. Adaptive refinement of the evolving surface is mandatory for high quality under reasonable computation times. In contrast to the lower-dimensional time line, there is a new set of refinement criteria that may trigger the refinement of a triangular initial surface, such as based on triangle degeneracy, triangle area, surface curvature etc. In this article we describe the computation of time surfaces for initially spherical surfaces. The evolution of such virtual "bubbles" supports analysis of the mixing quality in a stirred tank CFD simulation. We discuss the performance of various possible refinement algorithms, how to interface alternative software solutions and how to effectively deliver the research to the end-users, involving specially designed hardware representing the algorithmic parameters.

Keywords
visualization, CFD, large data, pathlines, timelines, surface refinement

1. INTRODUCTION
1.1 Motivation
Computational Fluid Dynamics (CFD) is a computationally-based design and analysis technique for the study of fluid flow. CFD can provide high fidelity temporally and spatially resolved numerical data, which can be based on meshes that range from a few million cells to tens of millions of cells. The data from CFD can range to several hundred thousand time steps and be of sizes in order of terabytes.

Therefore, a key challenge here is the ability to easily mine the time dependent CFD data; extract key features of the flow field; display these spatially evolving features in the space-time domain of interest. In this work, we present an interdisciplinary effort to generate and visualize time surfaces of the fluid flow from the time dependent CFD data. The implementation of time surfaces, such as an evolving surface of a sphere, for analyzing the flow field is more relevant in context of a stirred tank system. The integration of surfaces over time generates an evolving surface that can illustrate key flow characteristics such as how matter injected in a stirred tank disperses, and in what regions of the tank is the turbulence high. Such observations are crucial to identifying the best conditions for optimal mixing.
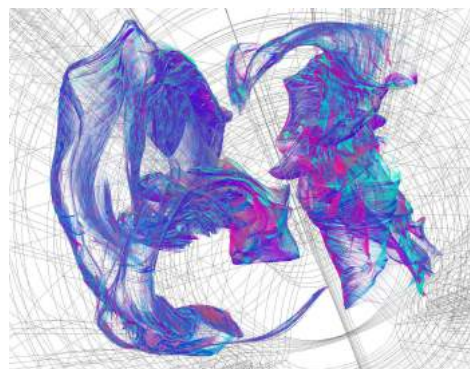

Figure 1: Two evolving spheres visualized just before their mixing in the Stirred Tank simulation system.

The CFD dataset was obtained from a large eddy simulation (LES) of flow inside a stirred tank reactor (STR). The

simulation is performed on 200 processors (64 bit 2.33 G Hz Xeon quadcore) where each time-step is calculated in approximately 36 seconds. Stirred tanks are the most commonly used mixing device in chemical and processing industries. Improvements in the design of stirred tanks can translate into several billion dollar annual profit. However, better designs of stirred tanks require detailed understanding of flow and mixing behavior inside the tank. The present study focuses on analyzing the dynamics of mixing inside the tank. Turbulent flow inside the stirred tank was solved numerically using LES to resolve small-scale turbulent fluctuations and the immersed boundary method (IBM) in order to model the rotating impeller blade in the framework of a fixed curvilinear grid representing the tank geometry. The grid is distributed over 2088 blocks and comprised of 3.1 million cells in total. Flow variables like velocity and pressure are defined at the center of each cell and computed for each time step over a total of 5700 time steps representing 25 complete rotations of the impeller. The handling and processing of these voluminous, multi-block, non-uniform curvilinear datasets to generate time surfaces and track set of particles in the fluid flow is the main challenge addressed in this paper.

1.2 Related Work

One of the earliest works related to this problem is the generation of stream surfaces, in particular Hultquist's attempt to generate a triangular mesh representation of streamsurfaces. Hultquist introduced an algorithm that constructs stream surfaces by generating triangular tiles of adjacent streamlines or stream ribbons. In Hultquist's algorithm, tiling is done in a greedy fashion. When forming the next triangle, the shortest leading edge is selected out of the two possible trailing triangles and appended to the ribbon. Each ribbon forming the stream surface is advanced until it is of equivalent length to its neighboring ribbon along the curve they share [13]. Particles are added to the trail of the stream surface by splitting wide ribbons, and particles are removed from the stream surface by merging two narrow (and adjacent) ribbons into one. Note that Hultquist's algorithm was developed for steady flows. Also, advancing the front of the stream surface requires examining all the trailing ribbons.

Along the same lines, Schafhitzel et al. [15] adopted the Hultquist criteria to define when particles are removed or added, but they derived a point-based algorithm that is designed for GPU implementation. In addition to rendering a stream surface, they applied line integral convolution to show the flow field patterns along the surface.

Rather than remeshing a stream surface when the surface becomes highly distorted, von Funck et al [23] introduced a new representation of smoke in a flow as a semi transparent surface by adjusting opacity of triangles that get highly distorted and making them fade. Throughout the evolution of the smoke surface, they do not change the mesh, but rather use the optical model of smoke as smoke tends to fade in high divergent areas [23]. However, the authors report that this method does not work well if the seeding structure is a volume structure instead of a line structure.

Core tangibles [21] we use in this paper are physical interaction elements such as Cartouche menus and interaction trays, which serve common roles across a variety of tangible and embedded interfaces. These elements can be integrated to dynamically bind discrete and continuous interactors to various digital behaviors. Many toolkits support low-level tangible user interface design, allowing designers to assemble physical components into hardware prototypes which can be interfaced to software applications using event-based communication. Notable examples include PHidgets [10], Arduino [2], iStuff [1], SmartIts [3] etc. Core tangibles focus on tangible interfaces for visualization, simulation, presentation, and education, often toward collaborative use by scientist end-users [21].

2. MATHEMATICAL BACKGROUND

In the domain of computer graphics one distinguishes four categories of integration lines $q \subset M$ that can be computed from a time-dependent vector field $v \in \mathcal{T}(M)$, mathematically a section of the tangent bundle $T(M)$ on a manifold M describing spacetime: *path lines*, *stream lines*, *streak lines* and *material line*s. Each category represents a different aspect of the vector field:

path lines (also called *trajectories*) follow the evolution of a test particle as it is dragged around by the vector field over time.

stream lines (also called *field lines*) represent the instantaneous direction of the vector field; they are identical to path lines if the vector field is constant over time.

streak lines represent the trace of repeatedly emitted particles from the same location, such as a trail of smoke.

material lines (also called time lines) depict the location of a set of particles, initially positioned along a seed line, under the flow of the vector field.

Each of these lines comes with different characteristics: stream lines and path lines are integration lines that are tangential to the vector field at each point

$$\dot{q} \equiv \frac{d}{ds} q(s) = v(q(s)) \qquad (1)$$

Since the underlying differential equation is of first order, the solution is uniquely determined by specifying the initial condition $q(0) = q_0$ by a seed point $q_0 \in M$ in spacetime. Neither stream lines nor path lines can self-intersect (in contrast to e.g. geodesics, which are solutions of a second order differential equation). However, a path line may cross the same spatial location at different times, so the spatial projection of a path line may self-intersect.

In contrast to stream and path lines, streak and material lines are one-dimensional cuts of two-dimensional integration surfaces $S \subset M$, $dim(S) = 2$. This surface is constructed from all integral lines that pass through an event on this initial seed line $q_0(\tau)$:

$$S = \{q : \mathbb{R} \to M, \dot{q}(s) = v(q(s)), q(0) = q_0(\tau)\}$$

The resulting surface contains a natural parametrization $S(s, \tau)$ by the initial seed parameter τ and the integration parameter s. It carries an induced natural coordinate basis of tangential vectors $\{\vec{\partial}_\tau, \vec{\partial}_s\}$, with $\vec{\partial}_s \equiv \dot{q} = v$.

For a streak line, the initial seed line $q_0(\tau)$ is timelike as new particles are emitted from the same location over time, $dq_0(\tau)/dt \neq 0$, for a material line the seed line is spacelike $dq_0(\tau)/dt = 0$, a set of points at the same instant of time. The respective streak/time line is the set of points of the surface $q(t) = S_{t=const.}$ for a constant time. If the integration parameter is chosen to be proportional to the time $s \propto t$, which e.g. is the case when performing Euler steps, then the original seed line parameter τ provides a natural parameter for the resulting lines, i.e. each point along a time line is advanced by the same time difference dt at each integration step.

Refinement of lines by introducing new integration points is mandatory to sustain numerical accuracy of the results. The ideas of the Hultquist algorithm [12] and its improvements by Stalling [17] could be applied also to the spatio-temporal case, however such would result in the requirement to perform timelike interpolation of the vector field. For data sets that are non-equidistant in time such as adaptive mesh refinement data generated from Berger-Oliger schemes [6] finding the right time interval for a given spatial location this becomes non-trivial. For now we refrain from non-equidistant refinement in the temporal direction (such as done in [14]), though this is an option – if not requirement – for future work.

A time surface is the two-dimensional generalization of a time line, a volumetric object in spacetime. The Hultquist algorithm, if applied to a spatio-temporal surface, discusses criteria on refining one edge, whereas here we have a much richer set of possible surface characteristics that may trigger creation or deletion of integration points. Some options are to refine a surface at locations where

- a triangle's edge
- a triangle's area
- a triangle's curvature
- a triangle degeneration ("stretching")

becomes larger than a certain threshold. Section 5.1 reviews our results experimenting with different such criteria.

3. SOLUTION
3.1 Data Model
We use the VISH [4] visualization shell as our implementation platform. It supports the concept of fiber bundles [8] for the data model. The data model consists of seven levels, each of which is comprised of compatible arrays that represent a certain property of the dataset [5]. These levels, which constitute a Bundle, are Slice, Grid, Skeleton, Representation, Field, Fragment and Compound. The Field represents arrays of primitive data types, such as int, double, bool, etc., and the collection of Fields describes the entire Grid. The Grid objects for different time slices are bundled together and are represented as a Bundle. As an example of our implementation, each Field contains values of a property such as coordinates, connectivity information, velocity, etc. The collection of these Fields is a Grid object, and the collection

of Grid objects for all time slices is the Bundle of the entire dataset.

The dataset used for visualizing the features of fluid flow contains numerical data for 2088 curvilinear blocks constituting the virtual stirred tank. The input vector field is fragmented and these fragments are the blocks of the Grid. The input dataset for each time slice consists of coordinate location, pressure and fluid velocity for each grid point in the entire 2088 blocks. These properties are stored as Fields in the Grid object for each time slice, and these Grid objects are then combined into a Bundle.

When a multi-block is accessed for the first time, a Uniform-Grid-Mapper is created which is a uniform grid having the same size as a world coordinate aligned bounding box of the multi-block. For each cell of the Uniform-Grid-Mapper a list of curvi linear block cells (indices) is stored which intersect the Uni-Grid-Mapper cell by doing one iteration over all curvilinear grid cells and a fast min/max test. When computing the local multi-block coordinates the corresponding Uni-Grid-Mapper cell is identified first which then selects a small number of curvilinear cells for the Newton iteration. Uni-Grid-Mapper objects are stored in the Grid object of the vector field and can be reused when accessing the same multi-block again later.

3.2 Out of Core Memory Management
The original approach taken while visualizing the features of fluid flow is to keep the entire vector field data in the main memory and integrate over the vector field to extract the features. However, with the necessity of visualizing the time-dependent 3D vector field, the original approach has restrictions, such as the size of the time-dependent data can easily exceed the capacity of main memory of even state of the art workstations. In [24], the authors present the concept of an out-of-core data handling strategy to process the large time dependent dataset by only loading parts of the data at a time and processing it. Two major strategies presented for out-of-core data handling are Block-wise random access and Slice-wise sequential access. The authors emphasize the Slice-wise sequential access strategy for handling the data given in time slices, however, we have implemented both Block-wise access and Slice-wise access of time-dependent data while generating the time surfaces for visualizing the fluid flow.
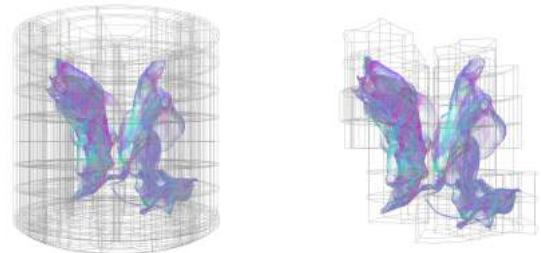

Figure 2: Time surface computed from a vector field given in 2088 fragments (curvilinear blocks) covering the Stirred Tank Grid (left). Only those fragments that affect the evolution of the time surface (right) are actually loaded into memory.
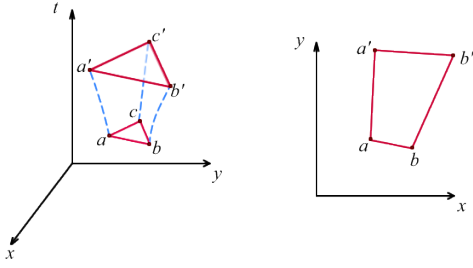
Figure 3: Particle advection of a 2-dimensional element vs. a 1-dimensional element. In our case, our surface element is in 3-dimensional space spanned over time.

The virtual stirred tank system has 2088 blocks, and each block has vector field data for every time slice. The data for each time slice is accessed only once as a Grid object from the input Bundle and processed to generate the time surface at that particular time. The integration of the time surface does not process all the blocks, instead only the blocks that are touched at the given time slice are loaded and processed.

At every time slice two Grid Objects are handled, one containing the input data of the vector field and the other consisting of seed points and connectivity information among the seed points. The connectivity information is used to generate the triangle mesh for surface generation. In the case of no surface refinement, the connectivity information is constant throughout the time slices and is stored once and used multiple times. This conserves the memory and also reduces the memory access. However, with surface refinement the number of points and their connectivity changes over time resulting in an increase in memory usage.

3.3 Particle Seeding and Advection

Our set of particle seeds q_{i,t_0} for $i = 0, ..., n-1$, lie on a sphere. At any given time $t > t_0$, the time surface is represented as a triangular mesh formed by the particles $q_{i,t}$ that have been advected using equation 1. Figure 3 illustrates the difference between our seeding approach versus Hultquist's where we are evolving a surface element (a triangle) over time as opposed to spanning a surface out of a line segment element.

3.4 Triangular Mesh Refinements

As time elapses, the triangular mesh of particles enlarges and twists according to the flow field. To preserve the quality of the mesh, we refine it by adding new particles and advecting them while updating the mesh connectivity. Of the possible refinements criteria mentioned above, we have implemented the following:

Edge length: If the distance between pairwise particles of a triangle is larger than a threshold edge length, we insert a new midpoint and subdivide the triangle accordingly.

Triangle area: If the area of the triangle formed by the new positions of the particle triplet is larger than a threshold area, we insert three midpoints and subdivide the triangle to a new set of four adjacent triangles.

4. ALTERNATIVE APPROACHES

In order to verify and compare our results with other implementations, we also investigate alternative implementations. Paraview [11] is one of the well known and widely used visualization tools in the scientific community. It addresses issues pertaining to the visualization of large scale data-sets using high-performance computing environments. It can be perceived as a framework around the well known Visualization Toolkit (VTK) [16] library. It not only provides a GUI to VTK, but also provides a convenient environment for intuitive visual programming of the visualization pipeline.

Paraview has implicit mechanisms for handling scale, both in terms of data and computation [7]. It achieves this by providing generalized abstractions for parallelization and distribution. Therefore a scientist using Paraview can switch from visualizing smaller data-sets on a desktop computer to a much larger data-set utilizing a large HPC infrastructure, with minimum effort.

We describe ongoing work and approaches to porting and visualizing the given F5 (fiber-bundle) data-set, as described in 3.1, in Paraview.

4.1 Porting the fiber-bundle (F5) to Paraview

The 500GB fiber-bundle data-set is provided in the F5 format. This format has no native support in Paraview and some form of conversion would be required to utilize the data.

One approach to solve this problem is to use a format converter and separately convert the entire file to a natively supported format. However, this approach causes redundant data and can waste considerable amount of space on the storage disk. An alternative solution is to write a custom reader into Paraview such that the data is read and mapped into internal VTK data-structures. This approach adds an additional computation time into the visualization pipeline and can cause unnecessary slowdown of the visualization process.

An ideal solution would be a combination of the above mentioned approaches such that both space and time optimization can be achieved. Such a solution is possible in our case due to a certain characteristic of the F5 format (explained shortly) and the use of XDMF (eXtensible Data Model and Format) [9] which is supported in Paraview.

An F5 format is characteristically a specific description or organization of the HDF5 data format. All H5 readers and commands which typically work on HDF5 formats also work on F5.

The XDMF data format is an XML format for data generally known as a "light data". It provides light weight descriptions of the "heavy data" which is typically a HDF5 file containing the actual data. A XDMF file can thus be seen as an index into the HDF5 file and is usually much smaller in size, taking very less time to get generated.

Paraview is supplied with the generated XDMF file through which it can access the data in the corresponding HDF5 (or F5) file. No other reader or converter is necessary. An

added advantage of this approach is that parallel file readers (if supported) and other parallel algorithms can be used to quickly access and process very large data-sets. We thus leverage on the parallel and distributed framework already provided in Paraview.

4.2 Details of XDMF for F5 fiber-bundle

An XDMF description of the F5 fiber-bundle is shown below.

```xml
<?xml version="1.0" ?>
<!DOCTYPE Xdmf SYSTEM "Xdmf.dtd" [
<!ENTITY HeavyData "50New.f5">
]>

<Xdmf Version="2.0">
 <Domain>
  <Grid Name="TimeSeries" Type="Temporal">
   <Grid Name="Multiblock" Type="Spatial">
   <Time Value="000000000.0000000000"/>
    <Grid Name="Block00001">
     <Topology TopologyType="3DSMesh"/>
     <Geometry>
       <DataItem Format="HDF">
       &HeavyData;:/f5/path/to/Points/Block00001
       </DataItem>
     </Geometry>
     <Attribute Name="Pressure">
       <DataItem Format="HDF">
       &HeavyData;:/f5/path/to/Pressure/Block00001
       </DataItem>
     </Attribute>
     <Attribute Name="Velocity">
       <DataItem Format="HDF">
       &HeavyData;:/f5/path/to/Velocity/Block00001
       </DataItem>
     </Attribute>
    </Grid>
    <Grid Name="Block00002">
     ...
    </Grid>
   </Grid>
  </Grid>
  <Grid Name="TimeSeries" Type="Temporal">
   <Grid Name="Multiblock" Type="Spatial">
   <Time Value="000000001.0000000000"/>
    ....
    ....
   </Grid>
  </Grid>
 </Domain>
</Xdmf>
```

As seen in the description above, the XDMF consists of a collection o f Temporal-Grids which represents each time step. Each Temporal-Grid contains a collection of Spatial-Grids which is a representation of multiblock data. Each multiblock data consists of curvilinear blocks. The data for these blocks are in the HDF5 file specified within DataItems.

As of now, results from the Paraview approach are still pending and subject of further investigation.

5. RESULTS

5.1 Surface Refinement

For the different refinement criteria test cases, we benchmarked our implementation with a 30-timestep subset (85 MB per timestep) of the stirred tank data and on a 64-bit dual core (2GHz each) pentium laptop machine with 4GB of RAM. We advected one sphere for the first 30 timesteps of the simulation. Due to the small size of our test data, we could not notice a difference in time surface meshing quality from the visualization itself, but from the data in tables 1 and 2, we notice a slight performance improvement of the area criteria over the edge length criteria. Though the number of particles is slightly higher in the second case, this suggests that the quality of the surface with the area criterion is better.

threshold	tot points	avg time/slice(sec)	tot time(sec)
0.005	4269	6.480	200.868
0.01	822	1.519	47.1
0.02	258	1.165	36.101

Table 1: Timing Analysis for Edge Length Criteria

threshold	tot points	avg time/slice(sec)	tot time(sec)
0.005	4269	6.864	212.785
0.01	837	1.454	45.08
0.02	258	1.150	35.646

Table 2: Timing Analysis for Triangle Area Criteria

From either tables 1 or 2, picking a threshold too small compared to the characteristic of the triangle being examined, results in maximum refinement, while a large enough threshold leads to no refinement at all.

5.2 Timing Analysis

For the overall integration and refinement of the time surface, we used a larger dataset of size 12GB with 150 timesteps. We ran the implementation on a 64bit quadcore workstation with 64 GB of RAM. We used the edge length criterion with a threshold of 0.01.

time	no. of points	time for slice(sec)	time/point(ms)
0	516	0.4	7.0
50	3468	2.0	5.9
100	15822	7.4	4.8
125	41574	18.8	4.7
150	129939	49.7	4.0

Table 3: Timing Analysis for Threshold=0.01

The listing in the above table is for 12 GB of input data from an initial time of 0 to a final time of 150. Initially the number of points is 516, which increases over time as more points are generated for surface refinement. As the number of points increases, the computation time for the next time slice increases. However, the time per point seems to be slowly decreasing. This may be because more and more points tend to locate in the same block and the data of one block is shared by many points, resulting in less memory access per point.
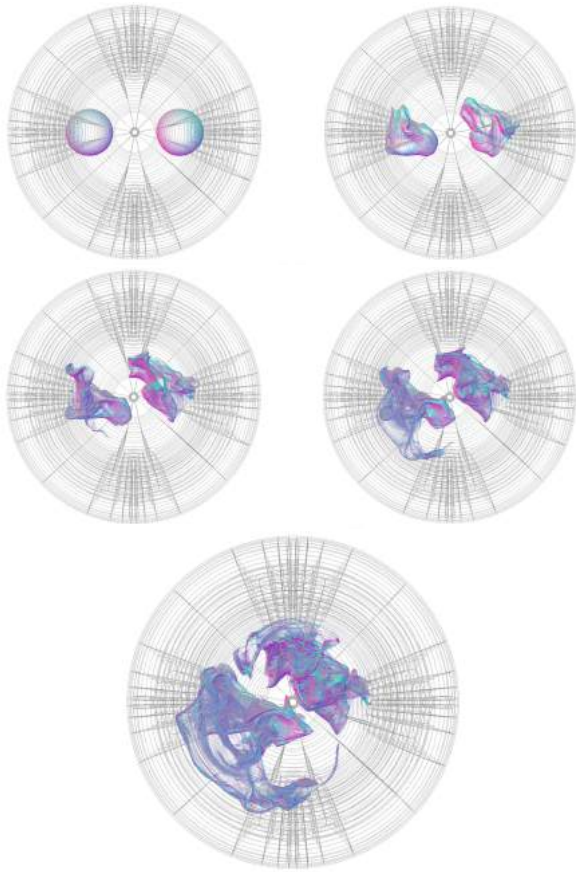
Figure 4: **Images showing evolution of two spheres at time slices 0, 50, 100, 125 and 150, respectively from left-top to bottom, as seen top-view of the stirred tank. First image shows the seed spheres, and the last image shows two sphere just before the surfaces are about to mix.**

6. DEPLOYMENT TO END USERS

Results of the algorithm can be investigated better if we explore the entire time evolution of the surface interactively, by navigating through space and time. In most visualization environments, the graphical user interface is tightly coupled with the underlying visualization functionality. One feature of VISH is that it decouples the interface from the underlying visualization application. At least in principle, this makes it as easy to couple VISH to a CAVE immersive environment, a web based distributed interface, or physical interaction devices as to the provided traditional 2D graphical user interface. As an example of this, we have based a significant portion of our interaction with the present large dataset from stirred tank with "viz tangible" interaction devices. An example of this is pictured in Figure 5. Earlier stages of this work have been described in [22, 20, 19, 18].

An application programming interface (API) is under development which supports coupling tangibles to VISH and other visualization environments. In this API, when interaction control messages are sent (triggered by physical events, such as RFID entrance/exit or the turning of a knob), they trigger corresponding methods in VISH. We use cartouches –
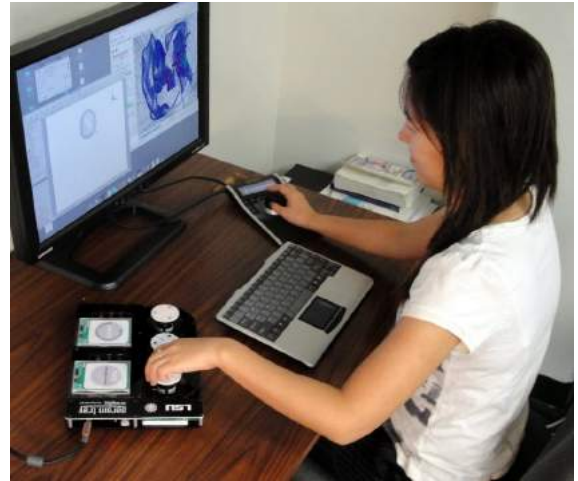

Figure 5: **User physically manipulating VISH application through "viz tangibles" interaction devices**

RFID-tagged interaction cards [19, 20] – as physical interactors which describe data and operations within the VISH environment. Users can access, explore and manipulate datasets by placing appropriate cartouches on an interaction tray (Figures 5, 6), and making appropriate button presses, wheel rotations, etc.
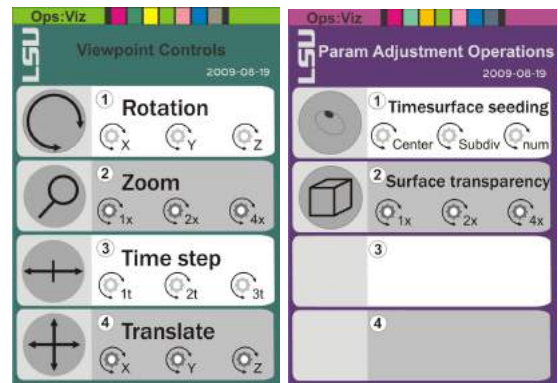

Figure 6: **Cartouche cards for viewpoint control and parameter adjustment operations**

In our present implementation, we have used two classes of cartouche objects. These are summarized below:

1. *Viewpoint operations:* Specific supported view point controls include rotation, zooming, and translation. In the case of rotation and translation, individual wheels are bounds to the (e.g.) x, y, z axis. In the context of zooming or time step navigation, wheels represent different scales of space and time navigation.

2. *Parameter Adjustment operations:* Our current implementation includes time surface seedings and surface transparency adjustment. For time surface seedings, we steer center of seeds, number of subdivisions, etc.

to parameter wheels. Within surface transparency adjustment, wheels are bounded to different scales of surface transparency.

In future, we hope quantities in high dimensional parameter space such as curvature and torsion of the surface can also be explored effectively with the integration of "viz tangibles" and the API.

7. CONCLUSION

While most of the previous visualization techniques for fluid flow have concentrated on flow streamlines and pathlines, our approach has been directed towards generating the time surfaces of the flow. The interdependencies of integration over a vector field require random access to amounts of data beyond a single workstation's capabilities, while at the same time requiring shared memory for required refinements. This limits available hardware and impacts parallelization efforts. The evolution of a seed surface required refinement of its corresponding triangular mesh to preserve the quality of the time surface over time. From the results we noticed a slight superior quality of the area refinement criterion over the edge length criterion.

8. ACKNOWLEDGMENTS

9. ADDITIONAL AUTHORS

Additional authors: Sumanta Acharya and Somnath Roy, Department of Mechanical Engineering, at Louisiana State University; acharya@me.lsu.edu,sroy13@tigers.lsu.edu.

10. REFERENCES

[1] R. Ballagas, M. Ringel, M. Stone, and J. Borchers. iStuff: a physical user interface toolkit for ubiquitous computing environments.

[2] M. Banzi. *Getting Started with Arduino*. Make Books - Imprint of: O'Reilly Media, Sebastopol, CA, 2008.

[3] M. Beigl and H. Gellersen. Smart-its: An embedded platform for smart objects. In *Smart Objects Conference (sOc)*, volume 2003. Citeseer, 2003.

[4] W. Benger, G. Ritter, and R. Heinzl. The Concepts of VISH. In 4th *High-End Visualization Workshop, Obergurgl, Tyrol, Austria, June 18-21, 2007*, pages 26–39. Berlin, Lehmanns Media-LOB.de, 2007.

[5] W. Benger, M. Ritter, S. Acharya, S. Roy, and F. Jijao. Fiberbundle-based visualization of a stir tank fluid. In *WSCG 2009, Plzen*, 2009.

[6] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53:484–512, 1984.

[7] J. Biddiscombe, B. Geveci, K. Martin, K. Morel, and D. Thompson. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13:2007.

[8] D. M. Butler and M. H. Pendley. A visualization model based on the mathematics of fiber bundles. *Computers in Physics*, 3(5):45–51, sep/oct 1989.

[9] J. A. Clarke and R. R. Namburu. A distributed computing environment for interdisciplinary applicationsâĂİ, concurrency and computation: Practice and experience vol. 14, grid computing environments special issue. *Currency and Computation: Practice and Experience*, (14):13–15, 2002.

[10] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218. ACM New York, NY, USA, 2001.

[11] A. Henderson. Paraview guide, a parallel visualization application, 2005.

[12] J. P. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *Visualization '92*, pages 171–178. IEEE Computer Society, 1992.

[13] J. P. M. Hultquist. Constructing stream surfaces in steady 3d vector fields. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 171–178, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[14] H. Krishnan, C. Garth, and K. I. Joy. Time and streak surfaces for flow visualization in large time-varying data sets. *Proceedings of IEEE Visualization '09*, Oct. 2009.

[15] T. Schafhitzel, E. Tejada, D. Weiskopf, and T. Ertl. Point-based stream surfaces and path surfaces. In *GI '07: Proceedings of Graphics Interface 2007*, pages 289–296, New York, NY, USA, 2007. ACM.

[16] W. Schroeder, K. Martin, and W. Lorensen. The visualization toolkit: An object oriented approach to 3d graphics, 1996.

[17] D. Stalling. *Fast Texture-Based Algorithms for Vector Field Visualization*. PhD thesis, Free University Berlin, 1998.

[18] C. Toole, B. Ullmer, R. Sankaran, K. Liu, S. Jandhyala, C. W. Branton, and A. Hutanu. Tangible interfaces for manipulating distributed scientific visualization applications. *Submitted to Proc. of TEI'10*, 2010.

[19] B. Ullmer, Z. Dever, R. Sankaran, C. Toole, C. Freeman, B. Casady, C. Wiley, M. Diabi, A. J. Wallace, M. Delatin, B. Tregre, K. Liu, S. Jandhyala, R. Kooima, C. W. Branton, and R. Parker. Cartouche: conventions for tangibles bridging diverse interactive systems. *Submitted to Proc. of TEI'10*, 2010.

[20] B. Ullmer, A. Hutanu, W. Benger, and H.-C. Hege. Emerging tangible interfaces for facilitating collaborative immersive visualizations. *NSF Lake Tahoe Workshop on Collaborative Virtual Reality and Visualization*, 2003.

[21] B. Ullmer, R. Sankaran, S. Jandhyala, B. Tregre, C. Toole, K. Kallakuri, C. Laan, M. Hess, F. Harhad, U. Wiggins, et al. Tangible menus and interaction trays: core tangibles for common physical/digital activities. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages 209–212. ACM New York, NY, USA, 2008.

[22] B. Ullmer, R. Sankaran, S. Jandhyala, B. Tregre, C. Toole, K. Kallakuri, C. Laan, M. Hess, F. Harhad, U. Wiggins, and S. Sun. Tangible menus and interaction trays: core tangibles for common physical/digital activities. In *Proc. of TEI '08*, pages 209–212, 2008.

[23] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2008)*, 14(6):1396–1403, November - December 2008.

[24] T. Weinkauf, H. Theisel, H.-C. Hege, and H.-P. Seidel. Feature flow fields in out-of-core settings. In H. Hauser, H. Hagen, and H. Theisel, editors, *Topology-based Methods in Visualization*, Mathematics and Visualization, pages 51–64. Springer, 2007. Topo-In-Vis 2005, Budmerice, Slovakia, September 29 - 30.