DIPLOMARBEIT

Cash Flow
A Visualization Framework for 3D Flow Data

ausgeführt am

Institut für Softwaretechnik
der Technischen Universität Wien

unter Anleitung von

Univ.Prof. Dipl.-Ing. Dr.tech. Dieter Schmalstieg

durch

Ing. Michael Kalkusch
Wilbrandtgasse 45 / 4, 1180 Wien

10.Mai 2005
Datum

Unterschrift

Abstract

The main contribution of this work is the fusion of scientific visualization algorithms and a scene graph resulting in a dynamic creation of the data flow model being used. The most important flow visualization techniques will be introduced in this thesis. The current state of the art was analyzed by comparing the most important software packages with a special focus on data flow models being used.

CashFlow is based on an object–oriented, dynamic data flow model, which is configurable by an XML-similar file. This proposed dynamic data flow model is based on in direct and indirect linking on nodes inside a scene graph. Thus two loosely coupled corresponding data flow networks are available in addition. The conventional possibility of linking nodes inside the scene graph using their data–fields is also on–hand. In the context of this thesis the *CashFlow* prototype was implemented[1] demonstrating the proposed concepts. In order to be able to focus on essential aspects the implementation of *CashFlow* is based on Coin3D, an open source OpenInventor library published by *Systems–In–Motion*.

The number of numerical simulation evolved rapidly since the early 80^{th}, especially in the field of computer assisted engineering (CAE) sponsored by the automobile industry. The two separate worlds of numerical simulation and real–time graphics approach each other. *CashFlow* shall be one small contribution establishing ties between these two worlds.

Keywords: Open Inventor, scene graph, data flow network, data flow model, visualization, flow visualization, computational flow dynamics (CFD),

[1]CashFlow framework available at: http://www.kalkusch.at/CashFlow/

Kurzfassung

Die Fusionierung von Visualisierungsalgorithmen mit einem Szene Graphen und dem daraus resultierenden dynamischen Aufbau des Datenflussmodells stellt die Grundlage dieser Arbeit dar. Im Rahmen dieser Arbeit werden die wichtigsten Strömungsvisualisierungsmethoden vorgestellt. Es wurde der aktuelle Stand der Technik anhand der wichtigsten Softwarepakete analysiert. Dabei wurde besonders Augenmerk auf die dabei verwendeten Datenfluss Konzepte gelegt.

CashFlow baut auf einem objekt–orientierten dynamischen Datenflussmodell auf, welches über XML–ähnliche Skriptdateien konfigurierbar ist. Dieses dynamische Datenflussmodell basiert erstmalig auf direkter und indirekter Referenzierung von Knoten im Szene Graph. Dadurch können nun zwei lose gekoppelte, korrespondierende Datenfluss Netzwerke zusätzlich verwendet werden. Die herkömmliche Möglichkeit der direkte Verbindung der Datenfeldern von Knoten im Szene Graphen bleibt erhalten. Im Rahmen dieser Arbeit wurde ebenfalls ein Prototyp implementiert, der die vorgestellten Konzepte veranschaulicht. Um sich auf die wesentlichen Aspekte konzentrieren zu können, baut die Implementierung auf Coin3D, einer OpenInventor–Bibliothek von *Systems–In–Motion* auf.

Wie die Entwicklung im Bereich Computer Assisted Engineering (CAE) seit Begin der 80er Jahre gezeigt hat, nimmt der Anteil der numerischer Simulationsverfahren stetig zu, und die beiden Welten der Simulation und Echtzeit–Graphik nähern sich immer weiter an. In diesem Sinn soll auch *CashFlow* einen kleinen Beitrag zum Brückenschlag leisten.

Schlagworte: Open Inventor, Szene Graph, DatenFluss Netzwerk, Datenflussmodel, Visualisierung, Strömungsvisualisierung, CFD

Contents

$\Pi\acute{\alpha}\nu\tau\alpha \quad \acute{\rho}\epsilon\ddot{\iota}$
$\kappa\alpha\grave{\iota} \quad o\grave{\upsilon}\delta\acute{\epsilon}\nu \quad \mu\acute{\epsilon}\nu\epsilon\iota$

Pánta rhëi
kai udén menei

Alles fließt
nichts steht still

Everything flows
nothing stands still

Heraklit von Ephesus
550–480 before Christ

Chapter 1

Introduction

Scientific visualization is a useful tool in many areas, such as manufacturing, finite elements analysis, computational fluid dynamics simulations, telecommunications or geographic information systems. The key idea is to turn massive amounts of raw data into useful images for visual inspection, or visual data mining. Several software toolkits exists for this purpose such as VTK[VTK96], IBM OpenDX[DX91] or AVS Express[AVS92]. These toolkits generally follow a data flow paradigm, i. e. raw data is sent through a series of transformations until finally mapped to geometric primitives and turned into images or animations.

The new approach in this thesis[1] is the link between OpenInventor, a scene graph library and a variety of scientific visualization algorithms. Although scientific visualization toolkits like AVS and VTK already addressed rapid prototyping, until now it was not possible to write an application by scripting an OpenInventor file only. Not a single line of code is necessary to create visualizations from generated data files. Our approach is a fusion of the visual programming paradigm, promoted by AVS and OpenDX, and the concept VTK is based on. In VTK software modules are linked together using various programming languages.

To master the complexity of general purpose visualization, an object–oriented approach is advisable. For example, the white paper of one of the most recognized toolkits, VTK, mentions a number of design requirements such as modularity, extensibility, portability and simple interfaces. VTK's implementation uses concepts such as reference counting and separation of data set and action objects.

We observe that a scene graph library such as *Coin3D*[Coi00] satisfies all of the above requirements. *Coin3D* is an OpenInventor clone by *System In Motion*[Motay] based on [Aki92b][Aki92a] It provides a complete runtime system for managing collections of visualization objects as well as a general environment for graphical output. In fact, many trivial "visualization techniques", such as rendering of textures or, colored polygon meshes, as well as combining several images are readily supported. *Coin3D* also has mechanisms well suited to implement a general data flow paradigm. In a nutshell, the scene graph structure can be used to construct the data flow network, which

[1]this document is available at: http://www.kalkusch.at/CashFlow/kalkusch_thesis_CashFlow.pdf

is then executed using *Coin3D's* native traversal mechanism. These traversal mechanism provides a scene graph traversal as a primary data flow network and the ability to build a secondary data flow network using *field connections*, that create a network of linked nodes. Finally, the Studierstube extensions[Sch96b][Sch97a] to *Coin3D* make any software based on *Coin3D*, including the proposed visualization toolkit readily suitable for immersive virtual reality scenarios, in case such a solution is desired.

In summary, implementing a scientific visualization toolkit on top of *Coin3D* can be done by concentrating purely on the visualization aspects. The result is a set of complementary toolkits that together can address more complex applications than a scene graph library or visualization toolkit alone.

To understand the proposed design called "*CashFlow*", we will introduce several visualization algorithms suitable for flow visualization. The most important visualization toolkits will be compared with respect to the used data flow networks. The software design of *CashFlow* is explained in detail. The necessary components for a flow visualization system using a flexible data flow framework are introduced. Especially the linking between the scene graph concept and the data flow model is addressed in this thesis. Our objective is to provide raw data to the system, then let it flow through a series of transformations, and finally pass the data to a rendering method. Of course we want the maximum flexibility and extensibility for all these components.

Chapter 2

Related Work

In this chapter various data flow models of several visualization frameworks are introduced and analyses in section 2.1. The second half of this chapter gives an introduction to scientific visualization algorithms suitable for 2D and 3D flow visualization (section 2.2, page 19). The first thing to start with when designing a new framework is to take a close look at the existing available applications. Scientific visualization is covered by several Open Source[GPL85] frameworks like:

- VTK, the Visualization ToolKit [VTK96] ,
- OpenDX, Open Visualization Data Explorer [DX91] © IBM
- SCIRun [SCI02]
- Scalable Visualization [Too00]
- and others

A large number of commercial products for scientific visualization and flow visualization are available. Some of them are also of scientific interest, because a lot of research was done by some companies. Important commercial visualization systems are:

- AVS Express [AVS92] , the application visualization system
- TechPlot [Tec81], CFD post–processing software
- MatLab ®[Mat05]
- EnSight by Computational Engineering International, Inc.[CEI94]
- FieldView by Intelligent Light [Fie97]
- IRIX Explorer [Exp71] ® SGI
- and others

All of these frameworks and systems suite very special needs. VTK for instance has a large set of volume rendering algorithms and can be used and extended using programming languages like C++, JAVA, Python, PERL and TCL.

Evaluating these software packages lets one question arise. How can these software frameworks be compared in a suitable way. Since all visualization frameworks use a data flow model, it was the most obvious thing to compare. Figure 2.1 shows a general process flow diagram used in scientific visualization.

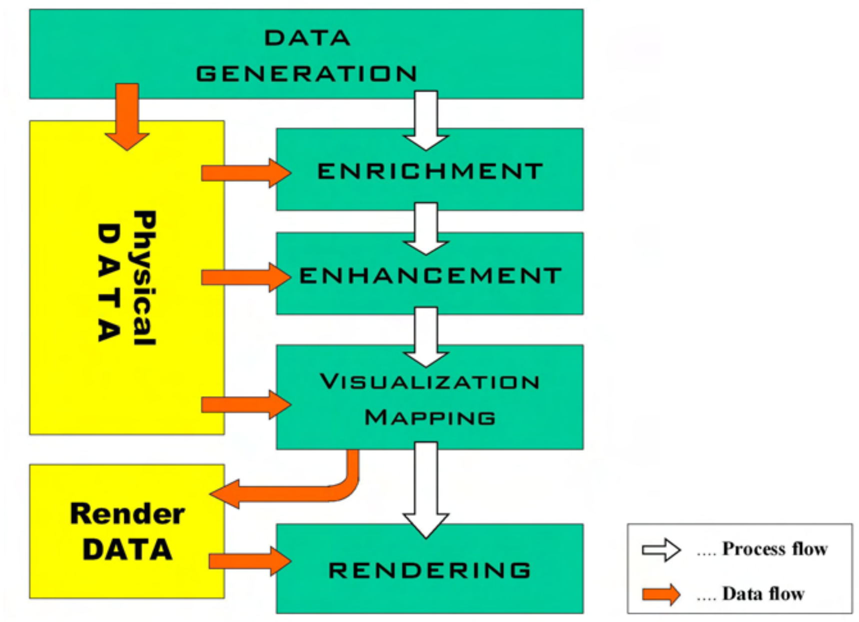

Figure 2.1: Visualization Pipeline

Scientific visualization receives input from a wide variety of data sources, both from sensors, such as medical scanners (CT, MRI, UltraSound, PET, SPECT) as well as numerical simulation (FEM, CFD).

- **Data Generation**
  Data can be obtained from real fluid flows or via numerical simulation. For computation of simulation data the two most important methods are:
  - **Finite Element Method (FEM)**
    to simulate a propagation of forces.
  - **Computational Fluid Dynamic (CFD)** systems
    that solve differential equations like the navier–stokes equation.
- **Data Enrichment / Enhancement**
  are techniques where parts of the data are selected or filtered. Due to the large

amount of data, especially when dealing with unsteady flow, the selection and filtering of data is very important. Other possibilities of generating data are the resampling of grids or fusion of different grids to name some.

- **Visualization Mapping**
  These are the visualization algorithms generating geometric primitives or new derived data. To enable an effective way of rendering, additional spatial data like for instance stream lines or iso–contours are created.

  Ordered by complexity there are three kinds of visualizations:

  - **Direct Visualization**
    The mapping is done without the creation of temporal objects. Examples are color maps (see section 2.2.1) on any surface and direct volume rendering (see section 2.2.8).
  - **Visualization based on Interpolation**
    has become a larger group of algorithms. These techniques generate new data based on the raw data like particle traces (see section 2.2.2) and 2D/3D contour lines (see section 2.2.3).
  - **High Level Visualization**
    These techniques are often based on particle traces generated either at the interpolation stage or directly for the high level visualization. They can be divided into Texture Advection methods (see section 2.2.4) and Flow Field Topology (see section 2.2.7) algorithms.

- **Rendering**
  Rendering is often executed via OpenGL,in our case through OpenInventor.

2.1 Data Flow Models

Using a scientific visualization toolkit always arises the question how to combine components of the system. The *data flow model* is well known in the field of mechanical engineering as well as electrical engineering and was adapted to computer science. Data flow networks normally are built from nodes and directed edges. Most networks are either required to be acyclic or use token. In general, edges indicate transport of data and nodes process that data. A wide spread technique to provide a high level of flexibility is to relay on visual programming as introduced by Hils[Hil91]. A visual editor is used for combining several modules or components of the scientific visualization toolkit. The following chapter compares several visualization toolkits, their data flow models and if available their visual programming concepts. Several books lead into the area of scientific visualization taking data flow models into account [HP97] [SM00] [AH02b].

2.1.1 Visualization Pipeline

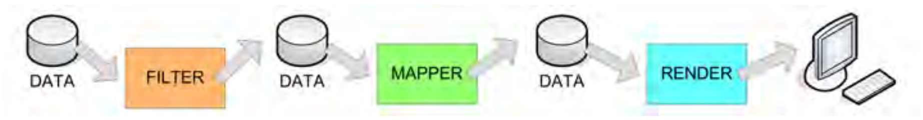

Figure 2.2: Traditional visualization pipeline: Raw data is processed by a Filter generating derived data that is mapped to geometric primitives by the Mapper. The Render unit generates the final image.

One important concept of scientific visualization is the traditional visualization pipeline[1] shown in figure 2.2 proposed in many papers like [IWC$^+$88][CUL89][WJSL96]. The stages and elements of the visualization pipeline are:

- **DATA**
  Each stage reads data, processes it and generates new data. Also a stage may create intermediate data like gradient information per vertex as part of the algorithm, that could also be used by other algorithms as well. Using the visualization pipeline allows to standardize this intermediate data which either can be computed on–the–fly or be part of a preprocessing step.

  This concept also allows optimizing only the parts of the pipeline without affecting other algorithms. When it comes to real–time computation some sections of the pipeline may be skipped or the algorithm feeds only parts of the pipeline. An example for skipping part's of the pipeline is direct volume rendering, were the hole data set is processed and rendered directly using a transfer–function.

  Most rendering algorithms can be mapped to this visualization pipeline in a proper way.

- **FILTER**
  The raw–data is processed by the *Filter* object either selecting parts of the data or resampling the input data to another grid. The *Filter* object generates intermediate data which is processed by *Mapper* or *Render* objects.

- **MAPPER**
  The *Mapper* object creates new data by applying a certain visualization algorithm to the data. Examples for mapping algorithms are applying color to geometric primitives, creation of textures based on values as well as generation of Glyphs.

- **RENDER**
  The *Render* object finally creates the geometric representation and combines all rendered images. *Filter* and *Mapper* objects feed the *Render* object. The *Render* object produces OpenGL calls only and does not produce other output data.

[1]details on *CashFlow* visualization pipeline in section 3.1, figure 3.1 on page 38.

- **Final Image**
  It is either created on the fly using OpenGL–commands or the *Render* object creates a texture. In figure 2.2 the screen icon represents the final image.

◇ **User Interaction**
  One important aspect is missing in the traditional visualization pipeline which is *the user* and his needs to interact with the system. Due to that lack Jock MacKinley[SKCM99] proposed a user centered visualization pipeline. This is addressed in section 3.12 on page 64.

The visualization pipeline is widely used in visualization systems like VTK[VTK96], Open Visualization Data Explorer[DX91] (see figure 2.11 page 15) and other packages. In the field of information visualization extension to the traditional pipeline are very popular like the one proposed by Jock MacKinlay[SKCM99]. In 1996 the *data flow model* was adapted to computer science first in SketchPad by Sutherland[Sut63] and ThingPad in 1979 by Bornig [Bor79][Bor81][AHB87]. In 1989 Upson et al [CUL89] published their work on AVS, the Application Visualization System. Since they intended to include several different visualization algorithms in one framework they analyzed the structure and needs of different systems and algorithms. They came up with an *analysis cycle* shown in figure 2.3, that is similar to the visualization pipeline shown in figure 2.2 page 6.

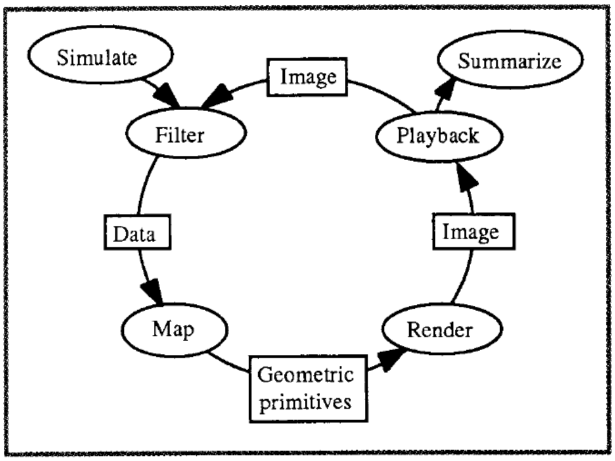

Figure 2.3: AVS analysis cycle[CUL89]. For details on the advanced visualization system (AVS) see section 2.1.4 on page 14.

The data flow model for scientific visualization by Dyer[Dye90] was extended for regular and irregular grids by Haber et al [RBHC91]. The software architecture for a scientific visualization system was analyzed and improved by Lucas et al. [BL92]. Visualization of multi–variable data using object–oriented design was published by [FH94][RAEM94]. The data flow model was also applied to the *multi–media component kit* by Demay et al [dMG93] and extensions to the data flow architecture were published by Abram [AT95] and Wright [HWB96]. The application of visual programming via a visual editor is also used in the field of multimedia like for instance in DirectShow 9.0 GraphEdit by Microsoft.

The Visualization Toolkit (VTK) system paper by Schröder, Martin and Lorensen [WJSL96] published in 1996 absorbed these concepts. VTK is open source project[VTK96] and has evolved until now with a focus on volumetric rendering (see section 2.2.8 page 36).

The following list shows a collection of visualization systems whose data flow models will be observed in detail:

1. **SketchPad** [1963]
   The first object–oriented visualization system with light pen interaction by Ivan Sutherland [Sut63].

2. **ThingLab** [1979]
   Editor suitable of constraints[Bor79][Bor81]. Successor of SketchPad.

3. **AVS** [1989]
   The Application Visualization System [CUL89][AVS92]. Its successor, AVS Express evolved and is a very successful commercial product.

4. **Open DX** [1991] ©by IBM
   Visualization system using visual programming for rapid prototyping [DX91]. *OpenDX* is the short form for Open Visualization Data Explorer initially The initial project was named IRIX Explorer and it was renamed to Open Visualization Data Explorer (OpenDX) when IBM decided to release it as Open Source project.

5. **OPEN INVENTOR** [1993]
   An object–oriented scene graph system [Inv92]

6. **VTK** [1996]
   The Visualization Toolkit [VTK96] [Sch97b] [WJSL97]

7. **VISAGE** [1999]
   An object–oriented scientific visualization system [WJSV92]

8. **VISSION** [1999]
   An object–oriented data flow system for simulation and visualization [Tv99][TvW99]

Since our implementation of the CashFlow framework is based on a scene graph, it is important to introduce the concept of a scene graph. Scene graphs are object–orientated library and applications. In May 1994 Mark Pesce and Tony Parisi proposed VRML (Virtual Reality Markup Language) as a description language for static virtual environments. The name was soon altered to "Virtual Reality Modeling Language". On 24^{th} October 1994 the first draft on VRML 1.0 was released based on SGI[2] Open

[2]SGI, http://www.sgi.com/ [SGI]

Inventor 3D[3] metafile format. VRML was rarely used since Flash[Gay96] developed by Jonathan Gay was published in December 1996 by Macromedia[Mac]. One of the main problems of VRML was, that the content creation was rather difficult at the time it was released. Nowadays 3D content creation is much easier since common tools are wide spread in the community. Since Alias[Ali] reduced the price for MAYA[MAY] and 3DStudio Max[dM] from AutoDesk[Aut] is comparable inexpensive also a large group of artists have access to these tool.

When JAVA3D[3D] was published in May 27^{th}, 1997 by Sun[4] this was another introduction of scene graphs to a wide audience. Sun held a course on Java3D at Siggraph1997[5]. Recently NVidea presented the *NVidea scene graph* in August 9^{th}, 2004 at Siggraph2004. One advantage of scene graphs is, that they are easy to script.

In the following section these visualization systems will be introduced. Some of these systems evolved and are still used (3)(4)(5)(6) while others are of scientific importance (7)(8) only or have historical importance (1)(2). Also the concept of *visual programming* and the *scene graph* concept are introduced in the following section. A general overview on data flow models was published by Ed H. Chi [Chi02a][Chi02b].

Notions

In the following sections several visualization systems and their data flow models will be introduced in detail. Regrettably each system names its components and modules similar even though the functionality may differ. To avoid misunderstanding as far as possible Table 2.1 summarize the notions used in the following.

signal processing	source	filter	sink	
AVS	source	filter & map	render & output	1989
OpenDX	bottom of node	special node	top of node	1991
OpenInventor	node field	engine	node field	1993
VTK	source	filter	mapper	1996
CashFlow	data node	mapper	render	2005

Table 2.1: Comparison of notations from data flow models used in visualization systems.

[3]OpenInventor [Inv92]

[4]Sun Java3D version 0.95 specification released

[5]SIGGRAPH97 course 35: "Introduction to Java3D"

2.1.2 Constraint Based Visual Programming [1979]

SketchPad[Sut63] and ThingPad[Bor79][Bor81][AHB87] were programmed using Smalltalk[Ing78] and are examples for object–oriented design in computer graphics. SketchPad[Sut63] was implemented on a TX–2 mainframe at MIT's Lincoln Labs and is also an early example for advanced user interaction. A light pen was used as input device to create drawings by pointing to the monitor.

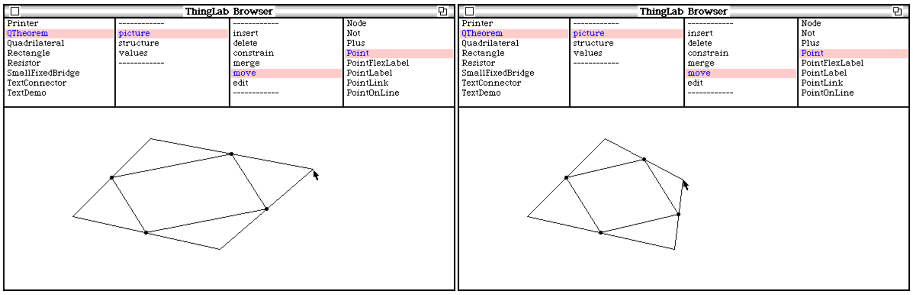

Figure 2.4: ThingLab constraint & object–oriented design[Bor79]: The right vertex of the outer rectangle is selected. The constraint of the inner rectangle is, that its vertices are in the middle of the outer rectangle. While moving the outer vertex the constraint is met and the inner rectangle is updated [Bor79][Bor81].

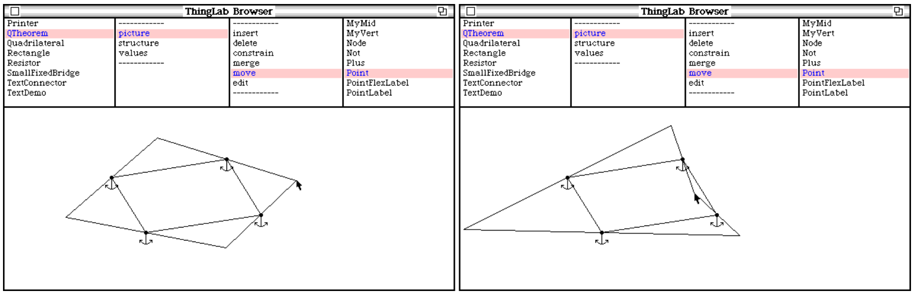

Figure 2.5: ThingLab anchor constraint[Bor79]: The right vertex of the outer rectangle is selected. The constraint of the inner rectangle is, that its vertices are at a fixed position symbolized by the anchor icon. While moving the outer vertex the constraint is met and the outer rectangle is distorted [Bor79][Bor81].

Alan Borning wrote this PhD–thesis[Bor79] on ThingPad which was the successor of SketchPad. Figure 2.4 shows a screen shot of the ThingPad application and the object hierarchy used. To move a point the following objects are selected:

$$QTheorem \rightarrow picture \rightarrow move \rightarrow Point$$

Constraints could be defined for the objects. For example in figure 2.4 the vertices of the inner square are on the midpoints of the lines of the outer square. If a point of the outer square is moved the constraint that the inner square is inside the outer square touching its edges in the middle is met. This results in a relocation of the inner square once the point of the outer square was moved.

The left image in figure 2.5 shows the same initial positions as in figure 2.4 except the anchor icon attached to the points of the inner square. This anchor symbols indicate, that in this example another additional constraint is applied to the system. The constraint is that the points emphasized by the anchor icons of the inner square are at a fixed location in space. Again the right point of the outer square is moved. This results in a relocation of the outer square while the inner square keeps its position. Both constraints are met. The constraint that the inner square is inside the outer square touching its edges in the middle and the fixed position of the vertices of the inner square.

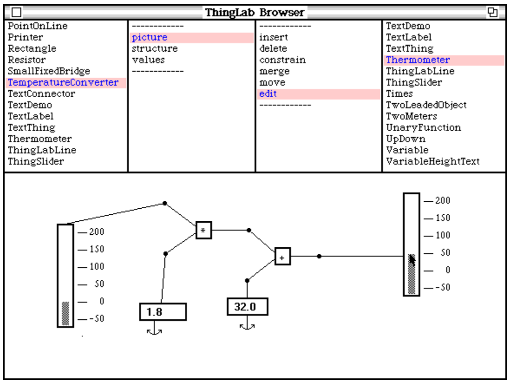

Figure 2.6: ThingLab user interaction example[Bor79]: The two thermometers are linked and show degrees Fahrenheit and degrees Celsius. While changing the value of the right thermometer using mouse pointer the value of the left thermometer changes also [Bor79][Bor81].

ThingLab also linked several objects together. Figure 2.6 shows two thermometers. The right one is selected by the user using the mouse pointer. The objects *"picture"* and *"values"* as labeled in figure 2.6 are linked. Once the user moves the bar of the right thermometer the values of the left thermometer are recalculated and the left bar is updated accordingly.

ThingLab is a good example for object–oriented design in early computer graphics. Constraints can be defined easily and are fulfilled by the framework. ThingLab uses variables and values to define constraints. A *constraint satisfier* keeps the system balanced if values change. The data flow model used in SketchPad and ThingPad is not visible and accessible directly by the user. The data flow is a result of the constraint and linked objects. In some cases like in figure 2.6 the user can perceive the data flow from the linking of objects quite well, but the data flow is not abstracted and visualized as a graph.

Borning extended his work on ThingPad and included *constraint hierarchies* [AHB87]. Constraint hierarchies are still used nowadays for instance in inverse kinematics. Today all established visualization software toolkits no matter whether open source or not rely on object-oriented design.

2.1.3 Object Oriented Visual Programming [1988]

A development towards an abstract view of the data flow model was the *Fabrik framework* [IWC+88]. Figure 2.7 shows a screen shot of that framework which is similar to ThingPad. The data flow model used in the Fabrik framework was however more obvious to the user. The visual components and the paradigms used were lent from electrical engineering.

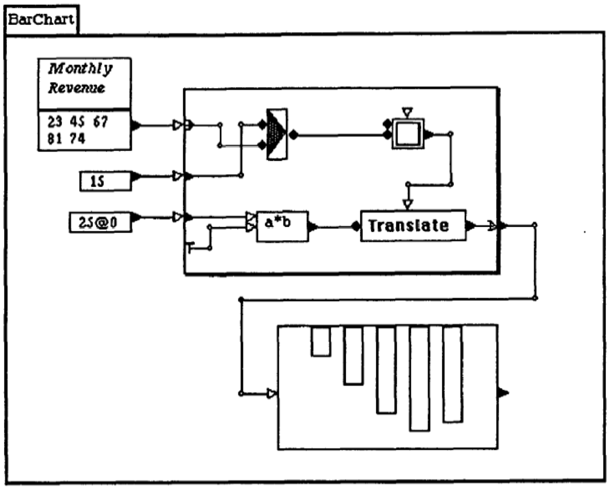

Figure 2.7: Data flow of Fabrik framework [IWC+88]: Components are linked to form a bar chart. Top left item provide values for bar chart. Middle item labeled "15" is a zero–shift. Lowest item labeled "250" is used to translate the bars.

The Fabrik framework is also an example for object–oriented visual programming. An introduction to object-oriented visual programming was published by Burnett [Bur94]. Several components are combined using a visual interface. The components are symbolized by icons of fields. Several icons are connected to form a directed acyclic graph inside a visual editor An example of such a directed acyclic graph is shown in figure 2.8 on page 13. The graph represents the data flow model.

Object–oriented visual programming mainly consists of two linked levels:

- *Verbal programming object*
  A programming object is a created in a verbal programming language.
- *Application & visual editor*
  Applications are created by linking together *programming objects* using a visual editor.

It is also common to visualize relations of applications or source code using a variety of diagrams and tools. For example Borland's Together® [Tog02][6] is capable to

[6]Borland®Together® details at http://www.borland.com/together/

either extract data from existing source code or to create new source-code by dragging and inserting icons inside a visual editor. A general introduction to visual programming in given in [Sch98] and [RP99]. An overview of advantages and disadvantages of visual programming in general was summarized by Schiffer [Sch96a].

Data Flow Networks

After the *Fabrik framework* next logical step in the evolution of data flow models was to introduce a higher level of abstraction and make extensive use of the object–oriented paradigms using hierarchies of objects. Such a high–level top–down approach as used in today's software frameworks consists of the following components:

- **Source**
  This object provides data labeled "S" in figure 2.8.
- **Filter**
  The transformation of data is done in the filter node. Filter modify the data or create new derived data. In figure 2.8 it is referred to as transformer "T".
- **Sink**
  This kind of node has only incoming edges and no outgoing edges. In figure 2.8 the sink nodes are called *Render* nodes "R". The VTK data flow model denotes sinks as *Mapper* as shown in figure 2.8.

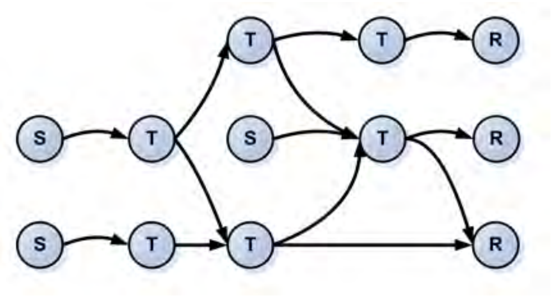
$$\begin{tikzcd}
 & & T \arrow[r] \arrow[dr] & T \arrow[r] & R \\
S \arrow[r] & T \arrow[ur] \arrow[dr] & S \arrow[r] & T \arrow[r] \arrow[dr] & R \\
S \arrow[r] & T \arrow[r] & T \arrow[ur] \arrow[rr] & & R
\end{tikzcd}$$

Figure 2.8: Data flow model shows directed acyclic graph (DAG). Sources "S" provide data and have outgoing edges only. Filters are defined as Transformer "T" and have incoming and outgoing connections. Sinks are referred as Renderer "R" and have incoming edges only.

Components are represented as icons that are connected by the user to form a directed acyclic graph (DAG). An example for such a directed acyclic graph including the components mentioned is shown in figure 2.8. *Source nodes "S"* have outgoing edges only, *filter nodes* are called *Transformer nodes "T"* and sink nodes are called *Render nodes "R"*. Note that some *Transformers* use multiple inputs and multiple outputs and one *Renderer* is connected to multiple inputs.

2.1.4 AVS, the Application Visualization System [1989]

AVS introduced the analysis cycle as shown in figure 2.3 on page 7. The analysis cycle was used to compare different needs of algorithms from different fields. The analysis cycle is similar to the visualization pipeline in figure 2.2 page 6.

The initial work on AVS was published by Upson et al [CUL89] and many other publications on AVS followed [Vro94][CC91][PL90][Cal91]. The design of the AVS system evolved and was extended in 1995 to AVS Express [Vro95] which is available as a commercial product and AVS Express is still a market leader. The AVS framework uses a hierarchy to categorize 3D scalar fields (see figure 2.9).

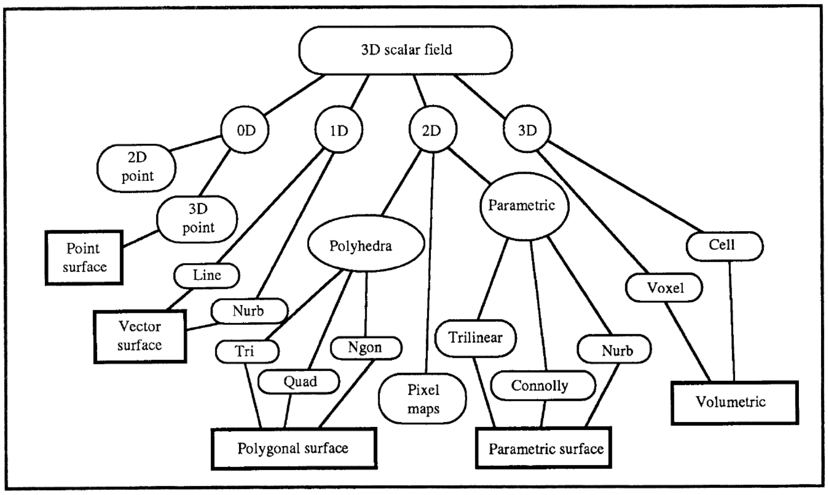

Figure 2.9: AVS Data flow[CUL89]: AVS mapping 3D scalar fields.

On the first level the *dimension* of the data is taken into account in the range of 0D to 3D (circular icons). On the second level the dimension primitives are mapped to possible *geometric representations* which is indicated by the rounded boxes. On the third level indicated by rectangles the geometric representations are combined to form *surfaces* or to define *volumes*. This hierarchy, consisting of the three levels shown in figure 2.9, forms a fundamental representation in computer graphics. The concept of AVS[CUL89] is similar to the work of Floriani et al[FF88] and to the *Field Model library*[Mor03][7]. The *Field Model library*, published by Patrick Moran [Mor01], is implemented as a C++ template library using *partial template specialization*.

AVS also introduced a computational flow network showing the linkage of software components. In figure 2.10 the main blocks correspond to the analysis cycle form figure 2.3 (page 7) which are:

$$\textit{Source} \rightarrow \textit{Filter \& Map} \rightarrow \textit{Render} \rightarrow \textit{Output}$$

[7] *Field Model library* available at http://field-model.sourceforge.net/

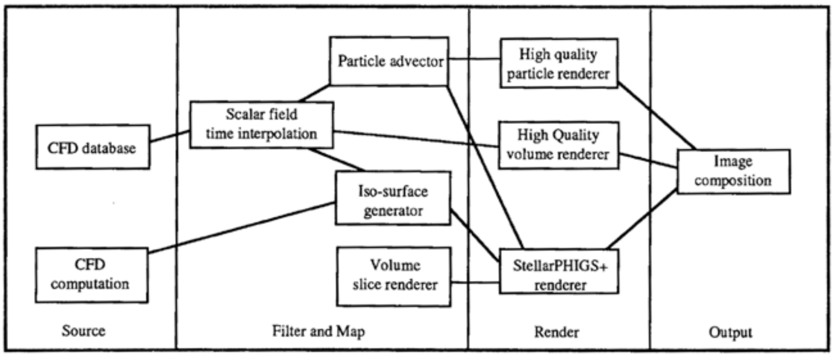

Figure 2.10: AVS computational flow network[CUL89]: Linking of software components in AVS. Direction of process flow from left to right. PHIGS+ is substituted by OpenGL & DirectX nowadays. Subdivision in source, filter & map, render and output correspond to figure 2.3 on page 7.

The Render block in figure 2.10 includes the StellarPHIGS+ renderer which was replaced by DirectX and OpenGL.

Nowadays *AVS Express©* added several components to the framework but the design in general is still the same. *AVS Express©* also uses visual programming to quickly assemble the visualization pipeline. Unfortunately *AVS Express©* is a commercial product of **Advanced Visual Systems Inc.©** and not open source although a research license is available.

2.1.5 Open DX, Open Visualization Data Explorer [1991]

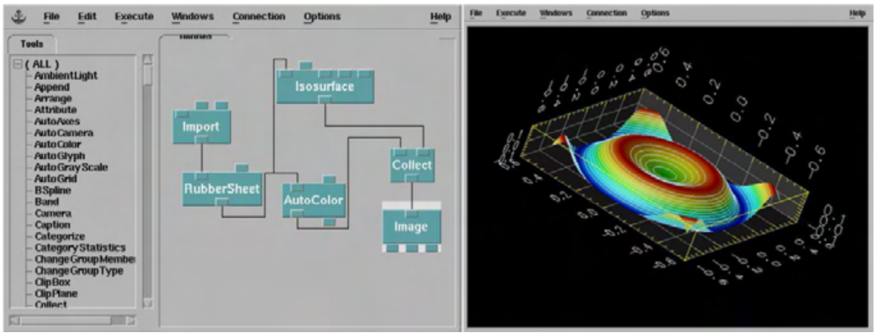

Figure 2.11: IBM OpenDX data flow model[DX91]: (left)OpenDX supports visual programming. Nodes are connected by edges. Edges are annotated and imply defined events. (right) Resulting image from shown pipeline.

This visualization system was one of the first systems having a visual programming editor in combination with several modules. Figure 2.11 shows an example of Open Visualization Data Explorer © IBM (OpenDX) with the visual programming editor on the left side and the created image on the right side.

Let's take a closer look at the data flow in figure 2.11. Modules consist of inputs, displayed as boxes on the upper side of the icon, and outputs, shown as boxes on the lower side of the icon. The importer loads data into the system and is connected to the RubberSheet object. The RubberSheet object creates a 3D height field from the 2D data. The output of the RubberSheet object is branched to the AutoColor and the Iso–Surface object. The AutoColor object calculates a color map. The Iso–Surface object generates the white contour lines. The output of AutoColor object and the Iso–Surface object are merged by the Collect object that is linked to the Image object.

One disadvantage of the visual programming approach is, that it is difficult to keep the general view in larger projects, because of the huge number of icons and conjunctions between them.

Although OpenDX is a Open Source project not much progress was made in the recent years, especially in comparison to VTK (see section 2.1.7 on page 18), which is Open Source project too.

2.1.6 Open Inventor [1992]

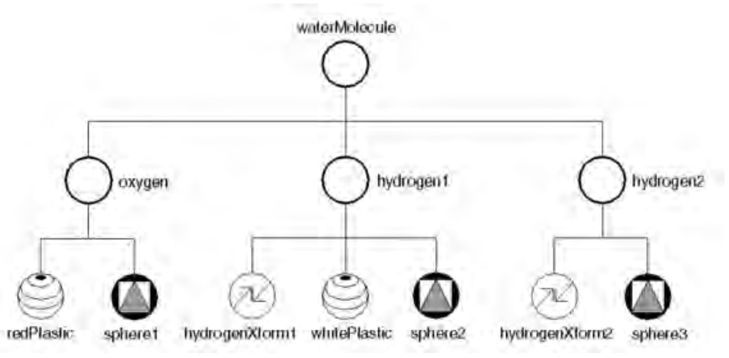

Figure 2.12: OpenInventor example scene graph

Open Inventor [8] is an object–oriented rendering toolkit based in IRIX Inventor [9] by SGI. Open Inventor uses a *scene graph* data flow model. A simple scene graph consists of a directed acyclic graph with nodes connected by edges. The scene graph is traversed from the root node using a depth–first–search order and a left–to–right convention on the same level.

[8]Open Inventor™ by SGI© [SGI]

[9]The IRIX Inventor™ file format was first released in July 1992 by SGI©

Open Inventor comprised different data flow networks based on the following concepts:

- actions & elements
- field connection
- engines
- sensors

A so called *Action* traverses the scene graph storing parameters for keeping information on the traversal state. The objects for these parameters are called *Elements*. *Actions* and *Elements* implement the visitor design pattern [Gam95]. The ordering of nodes in the scene graph defines the rendering. Rendering nodes that are visited before others render their content first. Some nodes only change values of the elements (property nodes) while others read the data from the elements generating 3D content (shape nodes). Special *separator nodes* group all sub-nodes and store the traversal state before processing the sub-nodes. After all sub-nodes are processed the prior traversal state is restored by the separator node. An example of a simple scene graph is shown in figure 2.12 on page 16.

Open Inventor is ideal for rapid prototyping and it comes with a file format that is an enhancement of VRML[VRM]. The Open Inventor files can be used to *script* an application. For further information and documentation see the Inventor Toolmaker[Wer94] and the Inventor Mentor[Wer93].

A second update mechanism is also available in OpenInventor. Each node can use several *fields* storing different kind of data. Two nodes at least can be linked by connecting the fields of the nodes. Using the *field connection* nodes can form a *field network graph*, because every time a field changes its value all nodes connected to that field via field connection receive the update. Each node receiving the update can implement a response that may include an update of other fields of the node leading to a recursion. Problems arising from that concept are discussed in section 3.3 on page 43. A reasonable overview on different strategies used for processing data flow networks and scene graphs was published in [ACT98].

Fields can also be connected to nodes outside the scene graph called *Engines*. *Engines* are only part of the *field network graph* and its concept is similar to the *chain–of–responsibility design pattern*[Gam95]. *Engines* are used to create complex connections between fields.

Finally *Sensors* can perform scheduling of tasks triggered by specific events. Events may be an update of parts of the scene graph, like fields, nodes or sub-graphs or a certain time passed. The sensor notifies a callback function.

Recently a visual programming editor was introduced for OpenInventor called *Coin Designer* [Des].

2.1.7 VTK, the Visualization ToolKit [1996]

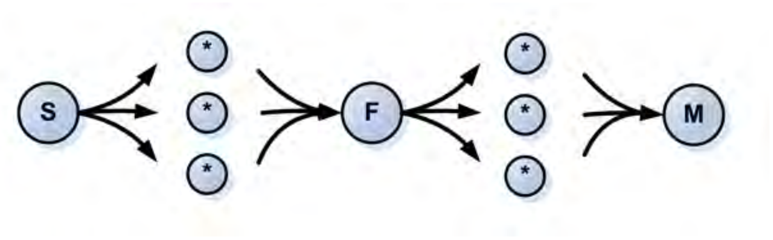

Figure 2.13: Data flow model of VTK [WJSL96]: Source "S", filter "F" and mapper "M" nodes. Node "" marks any node that may be connected to other node. Each node may have multiple connections to other nodes also known as fan–in & fan–out.*

The data flow model of VTK [WJSL96] (see figure 2.13) consists of *Source*, *Filter* and *Mapper*. *Source* nodes store data and are able to serve multiple output links. *Filter* nodes can handle multiple input and output sources. VTK puts the focus onto the data and therefore a *Filter* node transforms data. *Mapper* nodes may be linked to various input streams and create images, but do not produce any data output. VTK uses *lazy evaluation* to avoid unnecessary re–computation. The *lazy evaluation* concept was introduced by Henderson [HM76][Wad84]. There are two possible ways *Filter* nodes can handle data:

1. A filter node operates on the existing data and all nodes linked to the output of the filter node reference this data as sketched in figure 2.14a) .
2. The filter duplicates the input data (blue) leaving it unchanged (sketched in figure 2.14b)). The filter operates on the duplicated data (red) display on the right hand side of the filter. In that case nodes linked to the output of the filter, which are colored red and labeled "*" use the reference to the duplicated data (red).

Each node defines which kind of input or output is requested and which is optional. In other words each node defines a semantic for the link. Additional documentation on VTK is available at [VTK96][WJSL97][Sch97b]. A visual programming editor for VTK called DVA[DVA96] is also available as a commercial product.

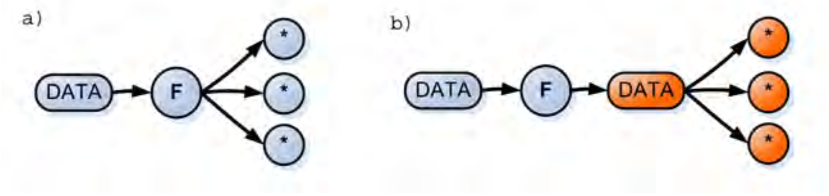

Figure 2.14: VTK Filters Data flow model [WJSL96]: Filters "F" offer two possibilities in VTK: (left) a) filter operates on input data passing references to linked nodes "". (right) b) filter duplicates input data(blue). Linked output nodes "*" (red) receive reference to duplicated or altered data (red).*

2.2 Flow Visualization Algorithms

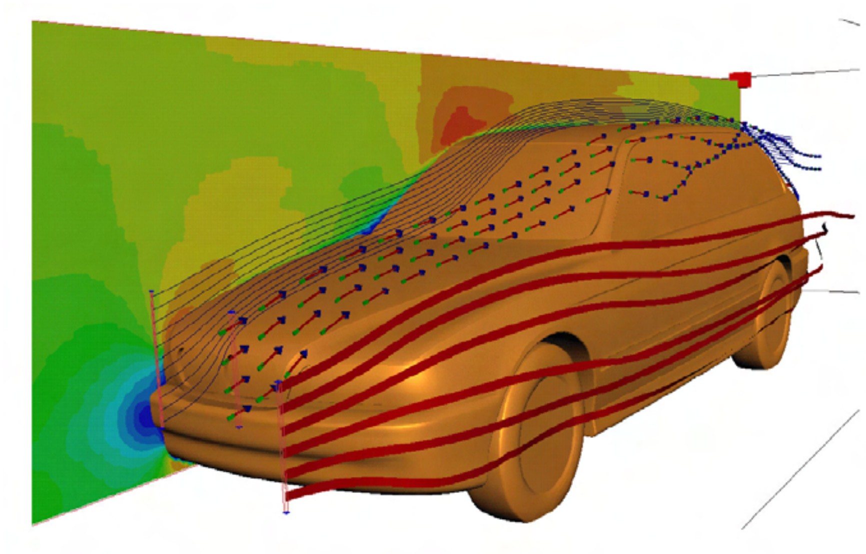

Figure 2.15: Combination of 3 different streamline techniques, color mapping of energy and iso–surfaces representing the hull of the car [MSE99].

A specific focus of this thesis are flow–visualization techniques. There are several well known techniques for visualizing flow fields. They can be divided in the following major groups:

1. **Basic Techniques**
2. **Particle Trace**
   (section 2.2.2 page 22)
3. **Contour Lines and Surfaces**
   (section 2.2.3 page 27)
4. **Texture Advection**
   (section 2.2.4 page 29)
5. **Flow Field Topology**
   (section 2.2.7 page 33)
6. **Volume Rendering**
   (section 2.2.8 page 36)

Figure 2.15 shows a useful combination of several visualization techniques combined in one image. Three different kinds of streamlines are used in this image. The red ribbons in front of the car are stream ribbons showing direction and vorticity of the flow. Behind that stream ribbons an arrow plot, which is bound to a stream line, emphasize the change of direction. In the back simple stream lines point out the increasing density of the flow towards the roof. Finally the color map generated from the scalar value *total energy* is applied to the cutting plane in the back. The hull of the car is an iso–surface, where velocity magnitude is equal zero. This image proves, that a combination of simple visualization techniques results in an image easy to perceive.

Classification of Visualization Algorithms

These algorithms can be grouped by their complexity:

- **L**ow complexity
  These simple algorithms do not generate intermediate data but render directly.
  - Basic techniques
  - Glyph, Arrows and Hedgehogs
  - Color mapping onto surfaces
- **Medium complexity**
  This group of algorithms generate intermediate data, which can be used by other algorithms also. Best examples are stream lines and iso–contours. These techniques have medium complexity.
  - Particle Trace
  - Contour Lines and Surfaces
  - Texture Advection
  - Volume Rendering
- **High complexity**
  These algorithms are based on results from medium complexity techniques. These techniques have high complexity.
  - Flow Field Topology
  - Interactive Data Generation
  - Streamline Seed point Strategies

2.2.1 Basic Techniques

- **Glyphs, Arrows and Hedgehogs**
  Using arrows or hedgehog's to show the direction and the magnitude of the flow is an old concept. The extension to that is the use of glyphs. Glyphs do not only show the direction and magnitude of the flow, but they also could be bound to other attributes. Figure 2.19b) on page 23 shows a Glyph visualization of a tornado. This visualization is very intuitive as long as each arrow can be perceived

and only a few arrows overlap. Due to that, arrow and glyph representations are very powerful when combined with focus and context methods. Different glyph representations were compared by [DHL01] shown in figure 2.16. Vector plot on irregular grids were addressed by [Dov95].

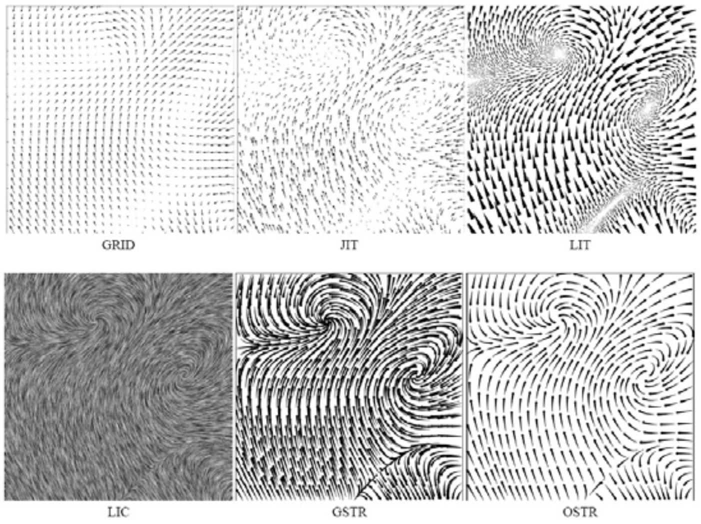

Figure 2.16: Example for different 2D flow field visualizations[DHL01].

- **Cutting planes**
  In order to be able to use the large number of algorithms for 2D, a cutting plane in 3D is a very important tool. There are several visualization techniques that can not be extended from 2D to 3D like LIC and spot noise.

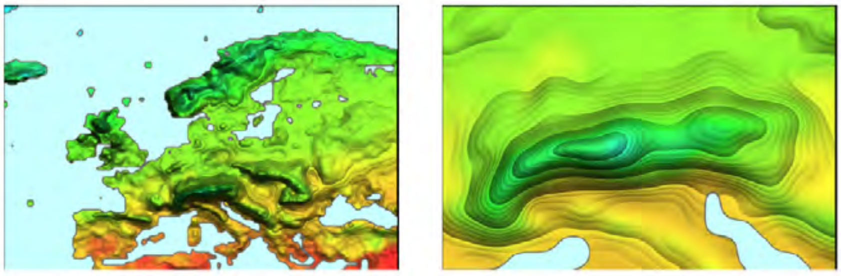

Figure 2.17: Enriched contour maps[vWT01]: Cubic color mapping to emphasize regions and simulate contour lines.

- **Color lookup table**
  Using a cutting plane is an easy powerful way to emphasis features and regions

in a flow field (see figure 2.15 on page 19). This is done by mapping one scalar value to a color by using a color lookup table. Color mapping on surfaces can be extended to surfaces embedded in 3D also.

A new kind of color map was published in 2001 by [vWT01]. Enriched contour maps use a cubic interpolation scheme to fade colors shown in figure 2.17 on page 21. Applying the contour maps to geographical 2D maps results in a common way of color coding and simulates height fields.

Another important technique suited for multi–valued data in 2D flow fields was introduced by [RMKL99]. Several transparent objects are mapped to different scalar data and vector data. In figure 2.18 black glyphs visualize the velocity of the flow. Underneath ellipses indicate pressure and its orientation is perpendicular to the flow. The third parameter used in the image is vorticity indicating regions of high rotational energy. If rotation due to vorticity is clockwise it is mapped to cyan. If rotation is counter–clockwise vorticity is mapped to yellow and if vorticity is very small or zero no additional color is used.

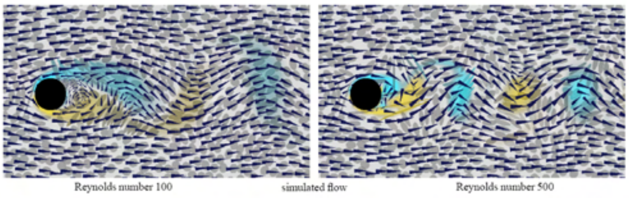

Figure 2.18: Multi–valued data visualized using concepts from painting [RMKL99]. Combination of transparent glyphs over ellipsoids mapped to pressure. Colors blue and yellow indicate vorticity of the flow.

2.2.2 Particle Trace

This is a very important group of algorithms, because a lot of other high level algorithms like texture advection (see section 2.2.4 on page 29) and flow topology extraction (see section 2.2.7 on page 33) rely on particle traces. Several types of particle traces exist in dynamics of fluids which are stream lines(1), potential liens(2), streak lines(3), time lines(4) and vortex lines(5).

Steady Flow and Unsteady Flow

Once a source or a sink is introduced into the system, we have an unsteady flow field. It can be described mathematical by: $div(F(x,y,z)) \neq 0$ where $F(x,y,z)$ defines the flow field at position $P(x,y,z)$ in space. The visualization of unsteady flow is very challenging, because the amount of data is huge and the problem of flickering animations have to be solved caused by aliasing. This problem was addressed by Bruckschen et al [RBJ01] shown in figure 2.22 on page 26.

When dealing with steady flow only one kind of particle traces does exist by definition, which are stream lines.

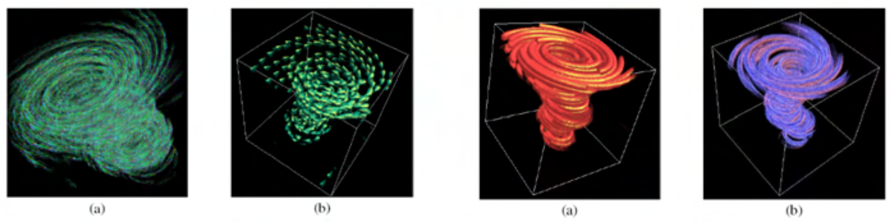

Figure 2.19: Different streamline visualizations for tornado data set[GSLS03]: Used techniques are (a)line bundles (b) Glyphs (c) depth cuing by lighting and (d) depth cuing by tone shading

Types of Particle Traces

A particle placed in the flow field creates a trajectory called the particle trace. Several kinds of trajectories and lines with special properties are defined in the dynamics of fluids.

1. **Stream line**
   Trajectory from a position $P(x, y, z)$ in the flow field with

$$\frac{\partial v}{\partial \phi} = 0$$

$$\psi = \frac{\partial v}{\partial y} \text{ ... direction perpendicular to the flow}$$
$$\phi = \frac{\partial v}{\partial x} \text{ ... direction of the flow}$$

   This is the case, if the velocity perpendicular to the streamline is zero. Due to that constraint all points on the streamlines are tangent to all velocity vector at a point in time (see [HD90] chapter B6, page B47). Several stream lines passing through a simple closed curve in space form a *stream tube*.

   In unsteady flow fields the particle trace is not identic with the streamline, since the orientation of the flow field changes. Thus the streamline shows the properties of the flow field at a certain time, while the particle trace shows the "history" of the particle in the flow field.

- **Potential line**
  The potential lines are perpendicular to the stream lines (see [HD90] chapter B, figure 23, page B57).

- **Streak line**
  Is a connection of adjacent points during trimester t_0. The points are traced over time and so is the streak line. In experiments this was done by repeatedly inserting an edge with ink on it into the flow.

  Defined for unsteady flow only.

- **Time Line**
  A time line is a connection of particle inserted into the flow field at a particular time. This is very intuitive way to visualize the acceleration of a flow.

  Defined for unsteady flow only.

- **Vortex Line**
  A vortex line is a closed line which has the same direction as the vorticity vector at a particular time. Thus it is the equivalent to a streamline and its velocity vector.

  Several vortex lines passing through a simple closed curve in space form a *vortex tube*. Figure 2.20 shows several deformed vertex tubes. Note, that vortex lines are created only by a starting turbulence and thus vorticity can only be created in unsteady flow fields.

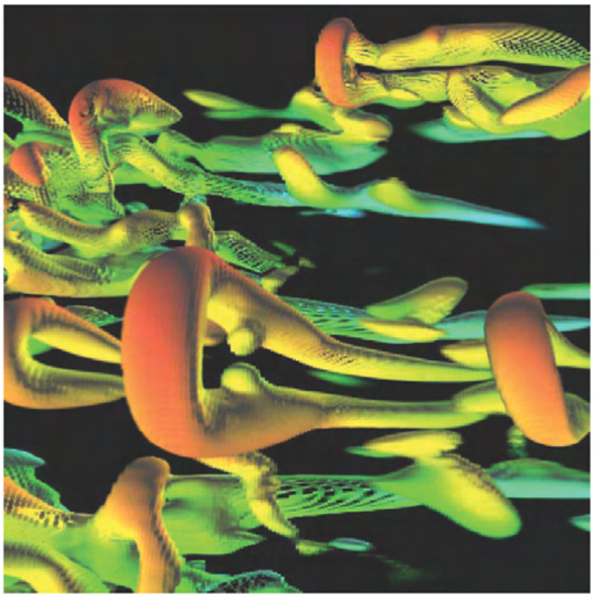

Figure 2.20: Vorticity Skeleton Visualization in 3D. Color denotes velocity magnitude. [SE04].

Aside of fluid mechanical definitions rendering a particle trace can be divided into the following steps:

- **seed point placement**
  This is very important if many streamlines are rendered especially in 3D. A common solution is a user guided seed point placement[RBJ01] like it was done in the *virtual wind tunnel*[KSK01] or the *virtual workbench*[OWD$^+$96]. Several strategies for seed point placement[VVP00][TB96] and interactive seed point placement[RBJ01] have been published.

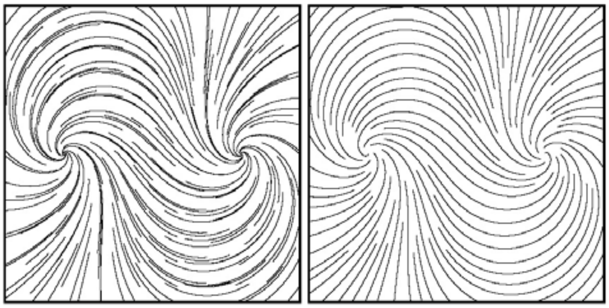

Figure 2.21: Evenly spaced streamlines: Streamlines are created using a special seed point strategy [BJ97]. The right image shows the same number of streamlines as in the left image. The evenly spaced streamlines algorithm was used to create the right image.

  * **evenly spaced streamlines** One other important concept for streamlines are *evenly spaced streamlines* [BJ97] (see figure 2.21). This algorithm creates a set of streamlines by a simple and effective combination of seed point placement and streamline integration. The seed points for streamline integration are set in dependence to the distance to existing streamlines. Evenly spaced streamlines were also extended to 3D [MTHG03].

- **forward and/or backward integration**
  Integration of streamlines can either be done in a pre–processing step or on–the–fly using Euler integration or higher–order Runge-Kutter integration. Streamline integration can be very difficult and time–consuming on curvilinear and non–regular grids. Most of the time optimized data structures like quadtree and octree for instance are used to speed up the integration step. The integration step is critical for the numerical stability of the visualization system.

  The first visual plausible and numerical stable simulation of fluids relayed on a simplified navier–stokes equation was published by Stam[Sta99]. It was also extended to surfaces of arbitrary topology[Sta03]. Stam's "stable fluids" were extended to animate fire by Nguyen et al [DQNJ02] and was adapted to the GPU by Harris et al[MJHL02].

- **streamline rendering**
  Many rendering techniques are common for visualizing the streamlines like for example StreamTubes, StreamRibbons, StreamBoxes and illuminated streamlines (see figure 2.23 next page). Out–of–core rendering of particle traces in real–time was published by [RBJ01] (see figure 2.22 next page). An extraordinary visualization from 1995 using StreamTubes shows a magnetic dipole reversal 2.24 by Glatzmaier and Roberts published in *Nature*.

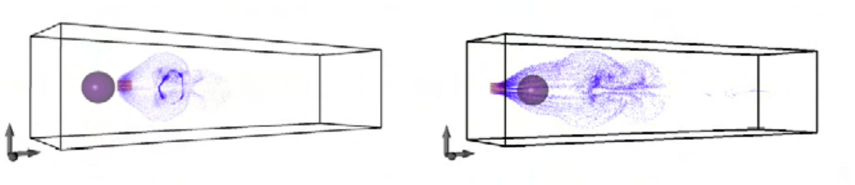

Figure 2.22: Real–time out–of–core Particle Traces[RBJ01]. The user can set particle seed points interactive using the red cube.

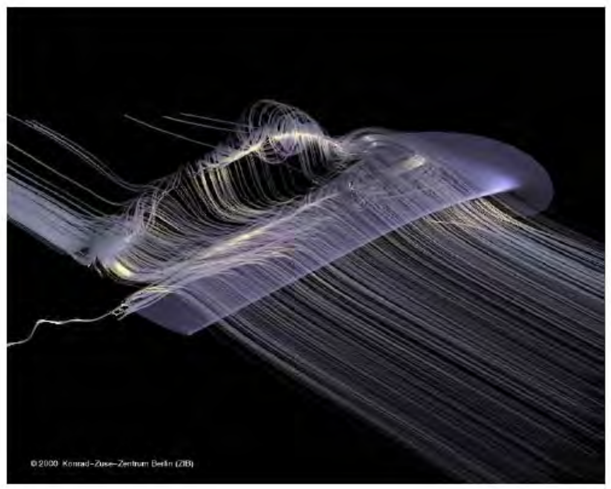

Figure 2.23: Illuminated streamlines passing a wing[MZH96]. Illumination improves depth perception.

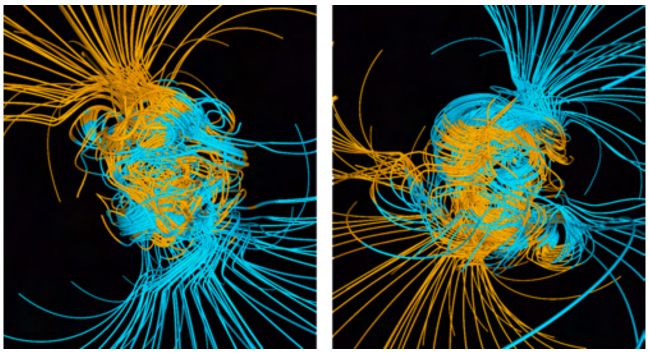

Figure 2.24: Magnetic Field of the Earth. (left) 500 years before magnetic dipole reversal. (right) 500 years after magnetic dipole reversal [GR95b][GR95a][TCCG99]. The color of the StreamTubes denotes magnetic North Pole and South Pole.

2.2.3 Contour Lines and Surfaces

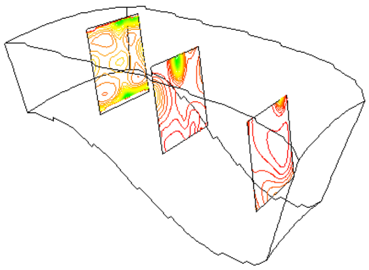

Figure 2.25: Iso–line showing velocity magnitude from a VTK example[VTK96] .

This visualization method is very old. Contour lines are also well known as contour lines in road maps. Since this concept was so intuitive, it has been adapted by mechanical engineers and extended to several data attributes like:

- **Isobar**
  Line of constant pressure. Is shown in most weather forecast maps circular around the high and low fields. Figure 2.26 on page 28 shows a visualization of a three–dimensional Isobar.

- **Isotherm**
  Line of constant temperature. Also frequently shown on weather forecast maps.

- **Isentropic**
  Line of constant entropy. Entropy describes the level of disorder in a thermodynamic system. Its opposite is Enthalpy, which is the available thermic working ability of a medium.

- **constant velocity**
  Line showing same amount of velocity magnitude (see figure 2.25 on page 27)

Marching cubes [LC87] is the most popular approach for iso–surfaces extraction and was extended by many others. The principle of Marching cubes was also applied for iso–surface extraction from irregular volume data [CMPS96]. Also Marching Tetrahedron [GS01] is quite fast and more simple to implement since there are not so many different cases to distinguish as in Marching cubes. The visualization pipeline was broken into pieces and iso–surface extraction was even done via the internet as web–based volume visualization by Klaus Engel[KEE99].

Iso–surface extraction still is an important research topic [LB02][Nie03]. For instance a speedup by using hierarchical iso–surfaces allows the user a quick change of the iso–value [VBL$^+$03][HLS03]. Even out–of–core iso–surface extraction has been

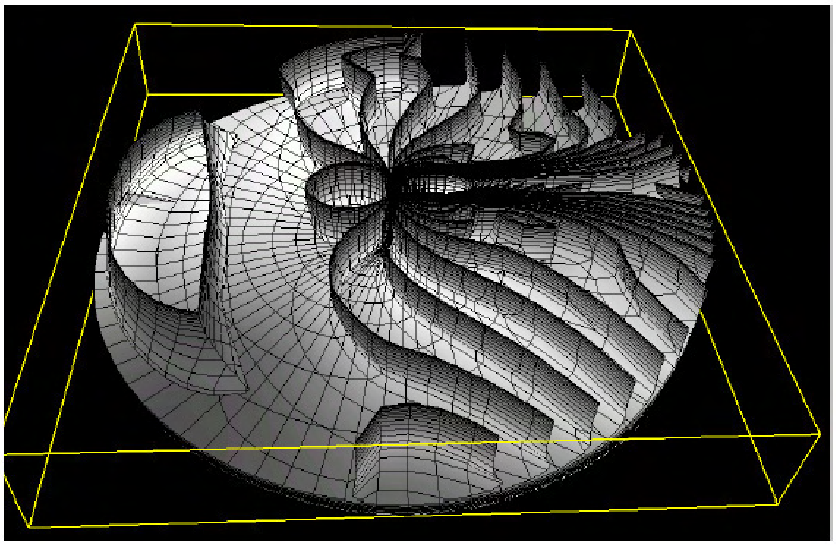

Figure 2.26: Iso–surface of oxygen post dataset[BLC01].

done [Chi03]. An extraordinary work on improving the quality of marching cubes was done by [TJW02]. Tao and Losasso[TJW02] included vector data on orientation of the iso–surface per iso–surface intersection point and calculated new rendering points that are not on the planes of the cells. For details see figure 2 on page 340 of Siggraph proceedings [TJW02]. Recently an astonishing paper on GPU–based isosurface rendering was published by Pascucci [Pas04].

2.2.4 Texture Advection: Spot Noise

Spot Noise [vW91]

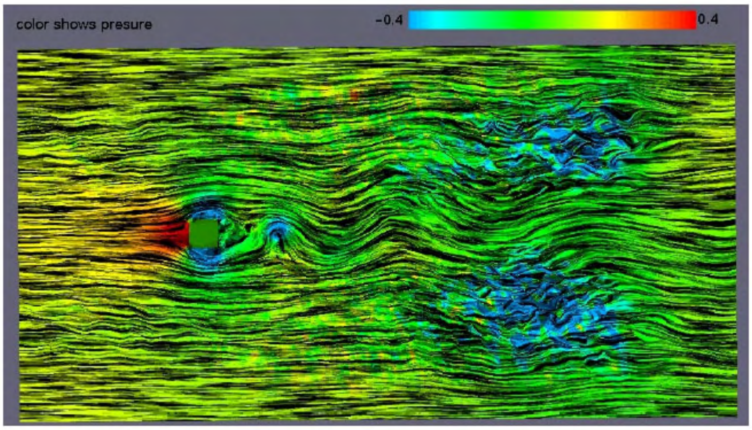

Figure 2.27: Spot noise [vW91]: Direction of flow is from left to right. The box cause swirl and vorticity.

Spot noise was the first texture method [vW91][dv95] (see figure 2.27). This method is used in 2D and is applied to surfaces embedded in 3D. A large problem of self occlusion occurs if this approach is applied in 3D (see figure 2.28). Since the new graphics cards are capable of handling large textures, this method is not used any more and was replaced by the PLIC rendering algorithm (see figure 2.29) and direct volume rendering (see figure 2.38).

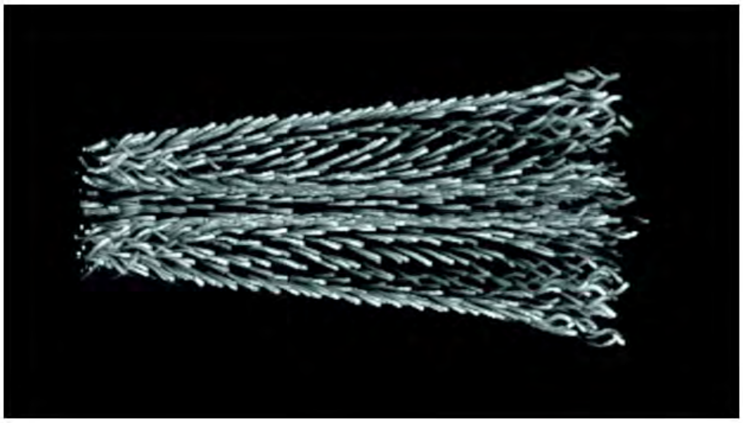

Figure 2.28: Spot Noise extended to 3D[vW91].

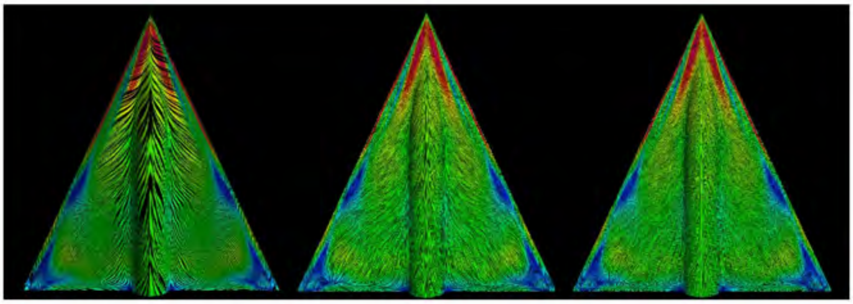

Figure 2.29: PLIC on a surface: Pseudo-LIC was an improvement of LIC including color[VVP99].

2.2.5 Texture Advection: LIC

Line Integral Convolution (LIC) [CL93]

Accumulating grey values from a *white noise* picture [Per85] along a stream line (see figure 2.16 on page 21). An example of LIC on a plane embedded in 3D shows figure 2.31 on page 31. To explain why LIC is so successful and why so many papers have been published on LIC is, that it generates images easy to perceive and very similar to the images from physics classes at school.

- **Pseudo – LIC (PLIC)** [VVP99]
  One of the most important extension to LIC [CL93] is PLIC [VVP99]. The author closed the gap between LIC and stream line drawing algorithms. The results from PLIC mapped onto 4 wheels of an aeroplane landing gear are also shown in figure 2.31. This algorithm uses one unique 1D texture for each streamline and speeds up LIC.

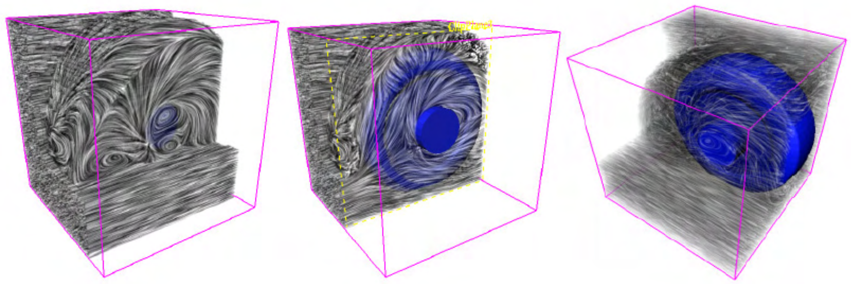

Figure 2.30: LIC in 3D with cutting planes and regions of interest [CRE99].

- **Unsteady Flow LIC (UFLIC)**
  This extension to LIC is also capable of visualizing unsteady flow fields. In combination with motion maps, which is a focus and context method, UFLIC is a reasonable method to visualize 2D unsteady flow [SK97][mSK98][WG97]. In figure 2.35 on page 34 UFLIC is combined with the results of a flow field topology algorithm by [XTH00].

- **LIC in 3D**
  Since LIC gained great results in 2D the next step was to extend it to 3D [IG97]. Unfortunately LIC in 3D suffers from one major disadvantage which is occlusion. This was one of the reasons why one of the first papers on LIC in 3D also focused on interactive exploration[CRE99] (see figure 2.30). That is just an elegant description for solving the problem of occlusion by forcing the user to explore the data interactively. Rezk et al[CRE99] also used cutting planes and regions of interest to emphasize the usefulness of LIC in 3D.

  Even volume rendering the 3D LIC texture as shown in the outer right image of figure 2.30 was one of their approaches.

  Future proved however something different. 3D–LIC is not really in use but never the less LIC on arbitrary surfaces is very successful. Since graphics hardware was extended by shaders new applications creating animated LIC images [GL05][RSLH05] like in figure 2.32 (page 32) will be the future of LIC in 3D.

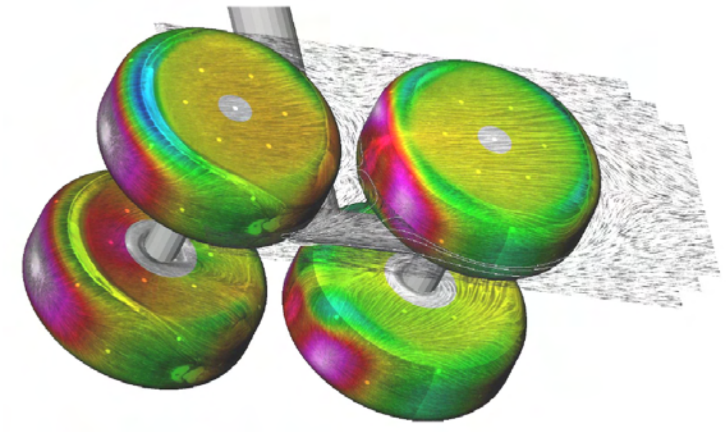
Figure 2.31: Fusion of colored LIC on landing gear with LIC in a plane. [KSK01].

2.2.6 Texture Advection in 3D

- Image based flow visualization
  New method based on animation of images [vW02] and the extension to curved surfaces [vW03].

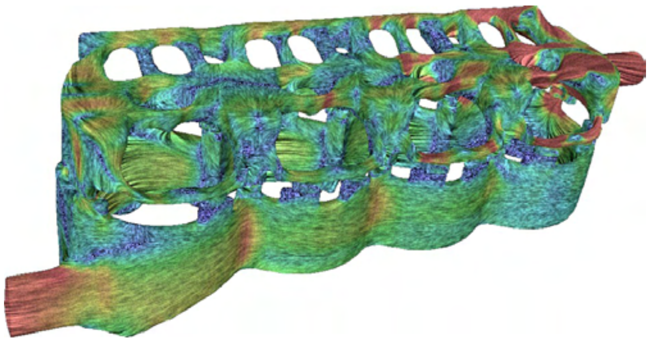

Figure 2.32: Image Space Advection on GPU [GL05]: This is a water jacket of a 4 cylinder 4 values combustion engine. Color is mapped to velocity magnitude.

- **Texture Advection for Unsteady Flow**
  One of the first papers on hardware accelerated texture advection [BJH00] was just the staring point for many papers on that very topic.
- **Texture Advection in 3D**
  Since vertex shaders and pixel shaders got so powerful animated textures similar to LIC are now rendered on the GPU [RSL04] [GL05]. Even arbitrary surface topology is no limit to that algorithm (see figure 2.32). Also texture advection creating visual plausible animated textures were ported to the GPU using shaders [PN01].

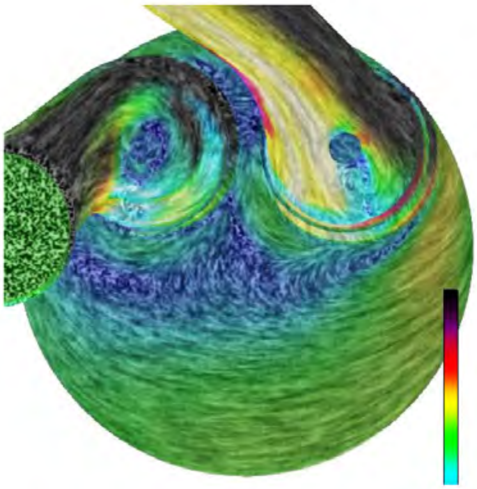

Figure 2.33: LIC on cylinder head of combustion engine (top view)[GL05]. Blue circular regions mark borehole of valves.

2.2.7 Flow Field Topology

All former techniques address the structure of the flow based on features like particle path finding in section 2.2.2. The next step is to divide the flow field into areas with similar properties. This was also done creating 2D & 3D contours in section 2.2.3, where a single scalar value was used to define regions. Due to the nature of a flow field some points are of special interest, namely the critical points. Critical points are classified as center, sources, sinks or saddle as shown in figure 2.35.

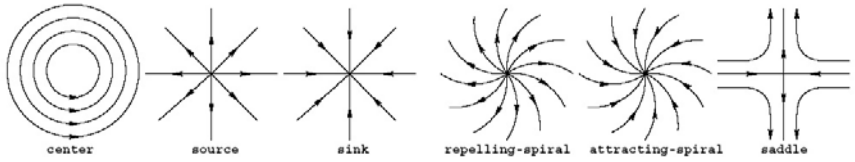

Figure 2.34: Different flow field topologies [VVP00].

Flow topology features can be compared to a sphere in 3D moving on a 2D surface. Using this analogy a particle of the flow behaves equal to the sphere moving on the surface. The flow topology features in 2D shown in figure 2.34 are:

1. *center* is a closed path of a particle with constant circulation. It is equivalent to an ideal horizontal plane with a sphere on it. A center describes an indifferent state.
2. *source* is a point generating new particles spreading in all directions.
3. *sink* is a point consuming particles from all directions.
4. *repelling–spiral* is a special case of a *source* with streamlines having a radial velocity component.
5. *attracting–spiral* is a special case of a *sink* with streamlines having a radial velocity component. Repelling–spirals and attracting–spirals can be observed frequently in nature, for example the flow of water exiting a washbasin.
6. *saddle* is a special case were all particles left of the north–south axis stay on the left side and vice versa all particles from the right side also stay on the right side. Further more the east–west axis also separates the flow. This leads to four separated regions.

The connection of critical points via particle traces generates a common visualization of the flow field topology. Figure 2.35 shows such a visualization in 2D amplified with a UFLIC[SK97] underdrawing. Red dots indicate critical points while blue lines are particle traces between the critical points. The blue lines delimitate regions of equal flow topology. This technique was also extended to 3D by [XTDS$^+$04] shown in figure 2.37 on page 35. The color coding used is matchable to figure 2.35. Red points indicate critical points connected by stream lines. Volume rendering is used to visualize the regions in 3D generating the sets of green and pink torus.

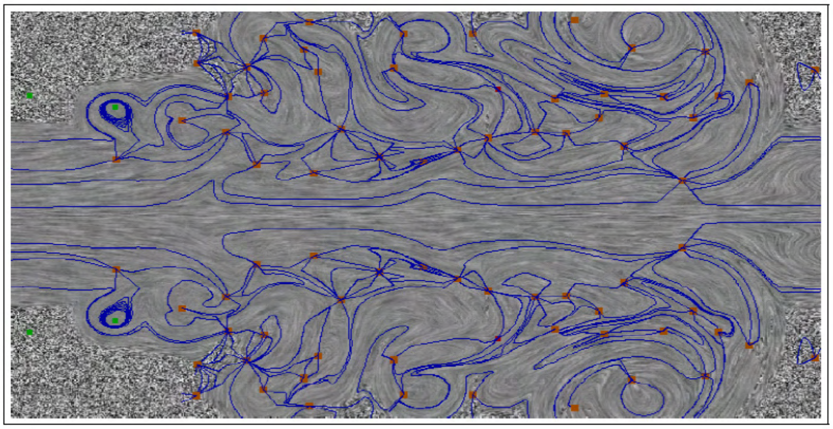

Figure 2.35: Colored Flow field topology extraction in 2D[XTH00]. Red and green dots mark critical points.

Vorticity & Turbulence

One of the important flow features are vortical phenomena. In most flow field analysis it is very important to locate turbulence and centers of vorticity. Either these regions are desired in specific locations or they have to be avoided strictly. For example, on the one hand vorticity and turbulence at the rear spoiler of a car are very important for good aerodynamics. On the other hand vorticity in the front part of an aeroplane wing has to be avoided. The detection of vortical phenomena was addressed by [HGPR99] and [Sad99]. Recently Stegmaier ported vortex detection to the GPU [SE04] as shown in figure 2.20.

Another concept for flow topology detection was also published by [dv99]. The creation of vector field hierarchies [BHH99] was one of the low level. The pure visualization of flow field topologies was addressed by [XTH00] and by [TTS03] in an artistic way similar to [Löf98][HLG98]. Most recent work on vortex visualization was published by Garth et al.[CG04].

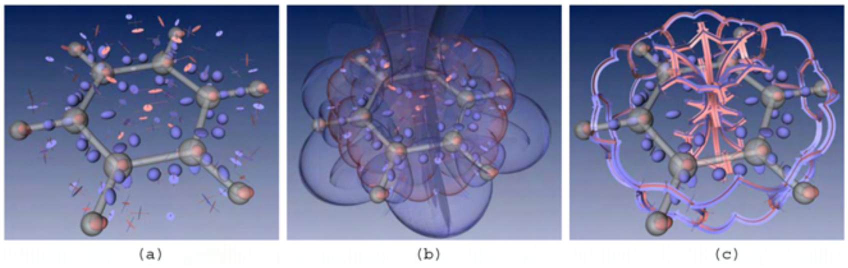

Figure 2.36: Topological Skeleton Visualization in 3D [HTS03].

Flow Field Topologies in 3D

Due to the difficulties of detecting critical points in 3D, as described by [Sad99] and [HGPR99], just a few papers have been published on field topologies in 3D. A lot of methods in 2D have been published like [XTH01]. Also hierarchical approaches for flow field analysis were published [TvW99]. A recently published paper on flow field topologies visualization in 3D by [XTDS$^+$04] was already mentioned above (see figure 2.37 page 35).

A somehow different approach was published by [HTS03]. The focus of their work was the visualization of molecular structures. Their solution shown in figure 2.36 looks convincing when applied to molecular structures, while their results on regular flow fields appear similar to the work of [Löf98][HLG98].

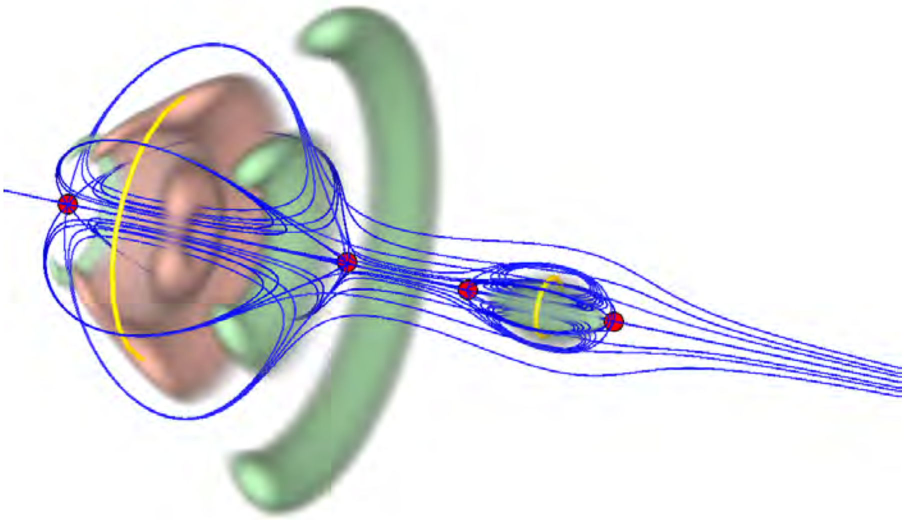

Figure 2.37: Example for Flow field topology in 3D combined with direct volume rendering [XTDS$^+$04] For details see section 2.2.7 on page 33.

(http://www.sci.utah.edu/stories/2004/fall_vortex-flow.html © by SCI Institute,University of Utah)

2.2.8 Volume Rendering

Direct volume rendering is a slightly different approach compared with the previous ones mentioned. While all other techniques take input data, process them and generate polygonal data from it, which is rendered afterwards, direct volume rendering works different. The volumetric data is processed directly using transfer functions and composed into one image [Lev90] (see figure 2.38 on page 36). Several optical models [Max95][Max86] are in use and some cause long computations. Special algorithms for volume rendering 3D vector fields were introduced by [CM92][EYSK94][Dov95]. The results are impressive and a large number of algorithms are available. SIM[Motay], the makers of Coin3D[Coi00], released a volume renderer called SIM Voleon[Vol10], which is not part of this implementation. An interface for passing data from *CashFlow* to the SIM Voleon can be one useful extension.

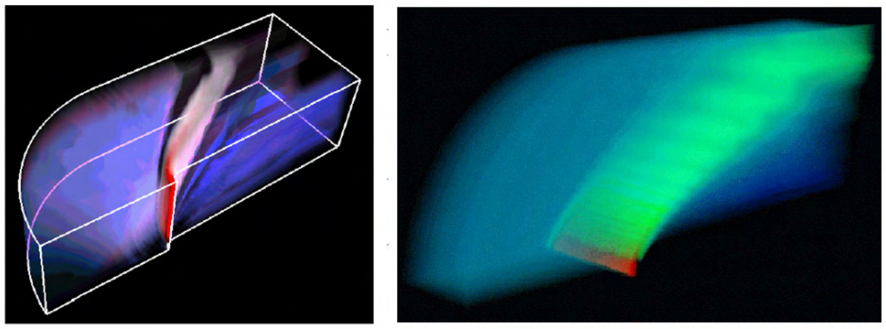

Figure 2.38: Volume rendering of blunt fin dataset. [BLC01] sec 2.2.8 © ACM Press

Volume rendering can be subdivided into five main algorithms:

- **Ray Casting** [Lev88]
  For each pixel in the final image one ray is cast through the data–cube from the image plane. By using front to back composite in combination with early ray termination, ray–casting can be a quiet fast algorithm[JD92].

  This is a high quality algorithm running at medium speed.

- **Splatting**
  Splatting can be described as an inverse raycasting algorithm [Wes90][KMC99][ZC02][WR00]. Each cell from the data cube is splattered onto the image plane. The accumulation of the splats is done using different filter kernels.

  This is a slow high quality algorithm.

- **3D Texture Mapping** [CCF94]
  The hole dataset is stored in 2D texture slices, which can be trilinear interpolated

in hardware on modern graphics cards. The disadvantage of this approach is that the dataset has to be small, in order to fit into texture memory.

This is a fast and medium quality algorithm.

- **Shear Warp** [LL94]
  This algorithm is the fastest volume software renderer. By processing the hole data–cube one by one in cache order and compositing the voxels onto an intermediate plane this algorithm is really fast. The intermediate plane is warped in a postprocessing step using OpenGL.

  This is a fast and low quality algorithm.

- **Fourier volume rendering** [Lev92] [Mal93]
  This is a very fast algorithm but it has the big disadvantage, that only maximum intensity projection (MIP) images in grayscaling looking like X–ray images can be produced. The big advantage is the very good resampling abilities of this algorithm. Because no color images can be created this is a medium quality algorithm. Fourier volume rendering was recently applied to body–centered cubic lattices [Dor03].

Volume rendering uses transfer functions for mapping scalar values to color and opacity. Volumetric flow visualization using transfer functions was recently addressed by [Mle03][HM03]. Regions of interest are selected in several linked 2D scatter–plots using smooth brushing. Selected volumes are rendered using the shear–warp algorithm in combination with user defined transfer functions. A great comparison of traditional volume rendering and implicit stream volumes is given in [XZC04].

For more details on visualization and a wide introduction to scientific visualization see [HP97] and [SM00]. Details on real–time rendering is given in [AH02b].

Chapter 3

Software Design

3.1 CashFlow Visualization Pipeline

One thing most scientific visualization toolkits have in common is the traditional visualization pipeline from figure 2.2 on page 6 in section 2.1.1. It is independent from the rendering pipeline and consists of the the following elements:

$$Data \rightarrow Filter \rightarrow Mapper \rightarrow Render \rightarrow Image \tag{3.1}$$

Due to the scripting approach the *CashFlow* visualization pipeline from figure 3.1 differs from the classic visualization pipeline sketched in (3.1). The major change is the possibility to apply the *Filter* and *Mapper* several times as well as the option, that the *Render* object can be used repeatedly.

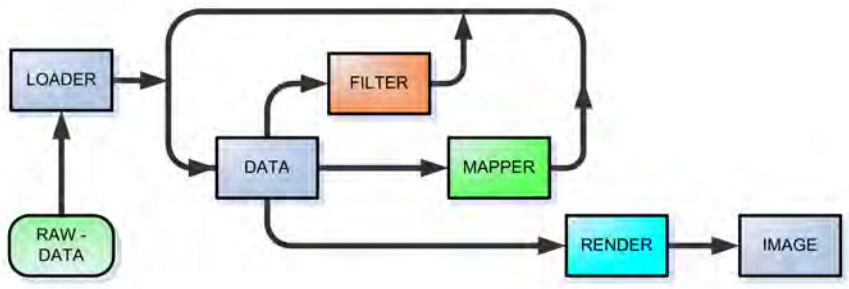

Figure 3.1: Process flow diagram of CashFlow. Raw data is imported by the Loader. Data can be processed afterwards by Filters and Mappers producing derived data. Derived data may be altered several times before it is passed to the Render unit. Class hierarchy shown in fig 2.2 on page 6

Recall that a data flow network is composed of nodes as well as arcs connecting the nodes as shown in figure 2.8 on page 13. A naive approach towards implementing the arcs in a data flow network would be to rely on Coin3D's SoEngine network mechanism (3.2), which is in fact already a kind of data flow mechanism.

$$[input] \qquad Node \rightarrow SoEngine \rightarrow Node \qquad [output] \tag{3.2}$$

A *Node* stores data that is used as input for the *Engine*. The *Engine* implements an algorithm and the generated new data is stored in another *Node* as sketched in (3.2). However, it is not suitable for implementing the desired kind of data flow, because

1. engines are not first class objects in the scene graph, due to the fact that they are not effected each time the scene graph is traversed.
2. there is limited control over the execution order of an SoEngine network.
3. the store–and–forward architecture of engines is not easily compatible with desired behaviors such as filtering.

The first *CashFlow* prototype was based on such a *"Node & SoEngine"* data flow network. Soon the limitations of that approach got obvious and we changed the software design, which is now based on *"Elements & Actions"* and will be described in the following chapter. Therefore, we choose to model the objects of the data flow as nodes in *CashFlow*, which allows convenient handling and scripting. The data flow network is built implicitly by the traversal order of the scene graph. Specialized elements are used to communicate between the nodes during the traversal. In that way, the scene graph traversal is used to dynamically build a data flow, similar to the dynamic way in which property nodes affect shape nodes (e. g., SoMaterial affects SoCone).

The *CashFlow* visualization pipeline in figure 3.1 consists of the following objects:

- **RAW–DATA**
  is generated mostly by numerical simulations (CFD,FEM)[1] or is created based on sensor data like CT,MRI,PET and SPECT[2] scans.
- **LOADER**
  The *Loader* node imports the raw data into *CashFlow*. Regrettably most data formats are binary and each type of grid has its own data format. This causes a large number of proprietary binary loaders.
- **FILTER**
  The *Filter* node selects a subset of the data. The result is a new (virtual) data object which is described by an additional node called *DataAccess node* (not shown in figure 3.1). No actual copying of data happens. The new data node is subsequently used as input for a Mapper node or Render node.

[1]CFD.... computational fluid dynamics, FEM.... finite element method

[2]CT.... computer tomography, MRI.... magnetic resonance imaging, PET.... positron emissions tomography, SPECT... single–photon emissions computer tomography

- **DATA**
  The obvious purpose of this node is to store data, but also meta–information like the type of grid and the topology information is stored in this node. *Mapper* nodes and *Render* nodes can process the data stored in the *Data* node. Sometimes *Mapper* nodes also generate intermediate data, that is used by other *Mapper* nodes or *Render* nodes. An example for useful intermediate data is the gradient information per vertex.

  The Data node is simplified in figure 3.1 as one node, but in *CashFlow* it consists of the following three nodes (not shown in figure 3.1):

  1. **MultiData node:** Is a container for data and consists of several multi–valued arrays for various forms of raw data like i.e. floating point values (coordinates, pressure, density etc.).
  2. **DataAccess node:** allows the user to define *virtual arrays* as described in section 3.4 on page 46. The *DataAccess* nodes link to *MultiData* nodes. *DataAccess* nodes provide a basis for *Filters* executed at runtime.
  3. **Grid node:** provides topological information on the data. Raw data does not have any associated topology. We need to know if the data describes a height field, a regular quad mesh, or an irregular tetrahedral grid. *Grid* node has a large number of subclasses that provide such topological information. We distinguish 2D/3D, irregular/regular, rectangular/spherical and so forth. While the regular grids need only a few parameters (such as width and height) for characterization, the irregular grids have an index field (comparable to SoIndexedFaceSet). The Grid node does not perform any rendering, it merely defines topology and is requested by Mapper nodes and Render nodes.

  The *Data node* is a semantic construction to keep the software design as independent for the implementation as possible. Note, that a Data node does not exists in our implementation of *CashFlow*. Nevertheless one Data node serving all mentioned needs may also be a solution as the Field Model library proves. Our experience shows that separating the data in MultiData, DataAccess and Grid node enables the widest variety possible in combination with the scene graph.

- **MAPPER** The *Mapper* node serves as a base class for all nodes that transform input data into output data, involving creation or copying of data. Also most visualization algorithms are implemented as Mapper node's. The Mapper node therefore refers to an input data node as well as an output data node, which is fed with computed data. This is necessary if any new data is synthesized or computed, such as in the creation of streamlines from seed–points or the computation of an iso–surface from iso–values, to name some.

  Recomputation of the Mapper can be slow, so the circumstances for triggering recalculation can be configured and are not necessarily managed by the traversal order.

- **RENDER** The *Render* node creates images using OpenGL calls, based on the input received from current DataAccess node and Grid node (simplified in figure 4.20 as "Data"). Together, these two nodes define an arbitrary mesh and its topology, while Render nodes define the rendering style for visualizing the mesh. Rendering styles (see section 3.1 page 38) are generated by rendering algorithms like for example point cloud rendering, wire–frame mesh rendering, 3D vector plot rendering and polygonal rendering to name some. It does not produce other output data.

- **IMAGE**
  The final image is created without explicitly combining visualizations, like in OpenDX. The order of rendering nodes during traversal directly affects the rendering. It can consist of several visualizations generated by different *Render* nodes. The user can interact with the resulting visualization by manipulating it. Also the user may want to change attributes to effect the visualization algorithms at different stages of the data flow. the final image. This will be discussed in detail in section 3.12 on page 64.

◇ **DATA CONSUMER**
On a syntactic level *Loader*, *Mapper* and *Render* nodes are *Data Consumer* nodes, because they either read data from the *Data* nodes or write data to the *Data* nodes. All derived nodes can be classified easily by read and write access to Data nodes. Loader nodes use either write access only when loading data or read access only when storing data. Render nodes generate images only and therefore have read access only. Mapper nodes process data and generate new data which is a read–write access. This concept is addressed in detail in section 3.5 on page 49.

The data flow pipeline from figure 3.1 page 38) shows that data can be accessed in many different ways. It can be rendered directly without passing any filter or mapper. This is useful if only a point cloud should be visualized. On the other hand filter and mapper nodes can subsequently alter the data. Intermediate data can be stored using the loader. This is especially useful when costly algorithms like particle trace algorithms create streamlines. Various rendering methods can be applied to the same dataset. Another advantage using a visualization pipeline combined with a scene graph is, that complex variations of visualization algorithms can be created easily.

3.2 Using the Scene Graph

In this section we want to take a closer look at the data flow inside *CashFlow* and how the scene graph is used. As mentioned in the previous section 3.1 *CashFlow* uses a visualization pipeline (see figure 3.1 on page 38). The three basic components *Data*, *Mapper* and *Render* are used in a scene graph.

A simple example is shown in figure 3.2. Artificial data is mapped to four different grids. On the bottom left a polar grid with a color map created from the artificial data. On the top left the same polar grid mapped on a height field generated from the artificial data. On the bottom right a Cartesian grid with the same color map. And on the top right a Cartesian grid with a height field also generated from the same artificial data. The same artificial data is used to generate these four visualizations. Note, that the data itself is not defined on one of those grids, which makes no visual representation more appropriate than the other.

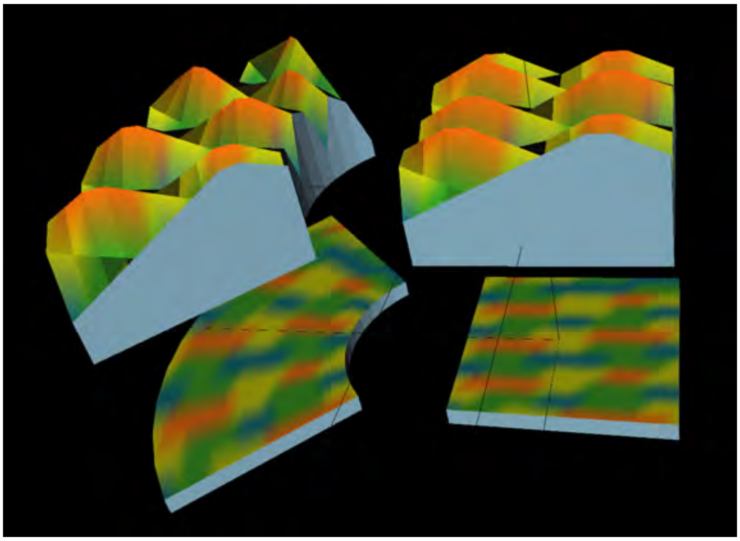

Figure 3.2: CashFlow: Abstract data mapped onto Cartesian and polar grid. Data values are used to create color map and height field. Corresponding CashFlow scene graph shown in figure 3.10 on page 51.

A simplified scene graph of the visualization in figure 3.2 is shown in figure 3.3 on page 43. The scene graph is traversed from root node R. Node A and node C stored data and are empty in the beginning. Node B is a loader loading data from storage to node A. Node D is a mapper mapping the data from node A to a color map stored in node C. During the first traversal node B insert new data into the scene graph. Node D also create new data in the form of a color map. In the following traversals node B and node D are inactive since the data is already loaded and has not changed.

The recursion of the depth–first–search scene graph traversal including node A-D is resolved and nodes E-H et cetera are traversed. After traversing node A up to node

D the nodes on the right hand side are traversed. The mapper node E creates a polar grid taking the data values from node A as elevations per grid point. The render node F uses the grid from the mapper node E and the color map from node C. Node G is also a mapper node creating the Cartesian grid shown on the top right in figure 3.2. Node G also uses the data from node A to create the height field and the render node H visualize the grid. The last two mappers generate a flat polar and Cartesian surface that is rendered by the two render nodes.

Note, that this is just a simplified scene graph that differs from the actual scene graph used in CashFlow. *The final* CashFlow *scene graph is shown in figure 3.10 on page 51.*

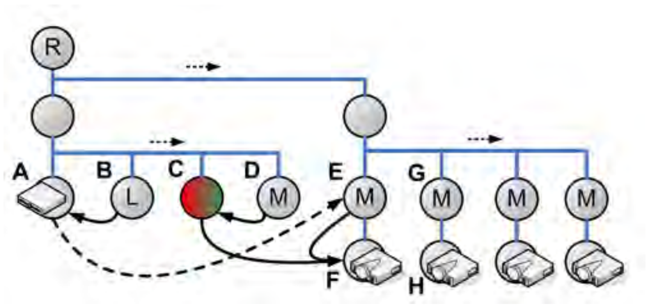

Figure 3.3: CashFlow: Simplified Scene Graph: Loader B loads data from disk to data node A. Mapper D maps data from node A and creates a color map in node C. Mapper E reads data from node C and creates a height field on a polar grid. Render node F visualize the height field from node E using the color map from node C. The result from node F is the top left grid in figure 3.2 on page 42.

3.3 Indirect References & Linking of Nodes

Using a scene graph allows two ways to exchange data between nodes.

- **Direct linking of nodes** [3]
- **Implicit linking of nodes** [4]

Two nodes in the scene graph can exchange data via a direct link between the nodes. This is also known as *field connection*, because the data is stored in fields inside the node. The field from one node is connected to the field of the other node. The disadvantage of this approach is, that each node holding data has to be explicitly connected to each other node in order to share data among nodes. This is inflexible and if the script file gets big it is quite confusing too.

The second strategy is the implicit linking of nodes. The concept is simple and powerful. The position of one node with respect to another node during the traversal of the scene graph should effect the linking of these nodes. If the order of the nodes

[3]for implementation details on direct linking of nodes see Implementation Field connection

[4]for implementation details on Implicit linking of nodes see Implementation Action and Elements

during the scene graph traversal is changed the linking of nodes should change in the same way.

For example in figure 3.4 node A and node B are data storage nodes. Let us assume, that the nodes R, S and T are three different kinds of visualization algorithms that are able to render the data from node A and node B. While traversing the scene graph node A is passed first. Afterwards Node R renders the data from node A. The next node in traversal order is the second rendering node S processing the same data from node A. Node B holds different data which is rendered by node T. Using *implicit linking* enables the user to simply exchange node A with node B and obtain a plausible behavior of the visual result. Now the nodes R and S will render the data from node B and node T will render the data from node A. The assumption made in this example is, that node A and node B use the same type of data. The unique key used for that is called a *relative address key*. The opposite to the *relative address key* is the *absolute address key* which is a unique string.

These short example emphasize the possibilities and advantages of a scene graph used for data flow modeling.

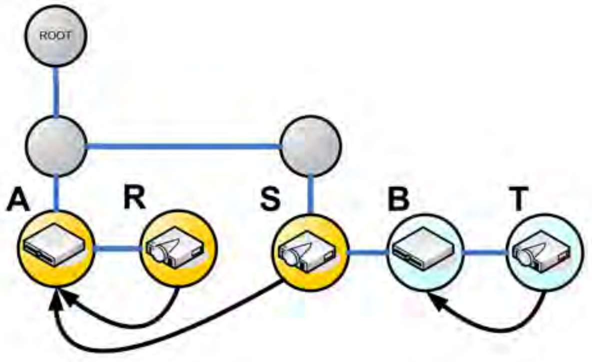

Figure 3.4: CashFlow: Implicit Linking on Nodes: Node A stores data. Node R visualize the data from node A. Node S also renders the data from node A, because no new data node is used in between. Node T finally renders the data stored in node B.

Figure 3.4 illustrates such a scene graph traversal dependent data flow model. Note, that the traversal order used is depth first search and left before right node order.

Each time the scene graph is traversed a *traversal state* is maintained. The nodes in the scene graph are visited using a depth first search order and a left before right order. The traversal state stores parameters and references that can be modified by the nodes. These transient parameters only exist during one traversal of the scene graph and are referred as *traversal state elements*.

While processing the scene graph each node can get access to the traversal state elements [5]. These elements store all kind of data used for rendering as for example the current color. This is especially useful, because nodes can refer to other nodes holding certain pieces of information without being explicitly linked to these nodes. These

[5]for details on elements see Implementation SoElements.

concepts provide a linking of different nodes depending on the scene graph traversal. Thus the data flow model can be altered at runtime by simply changing the scene graph or even by only altering the traversal order of nodes as mentioned in the previous example.

Indirect Reference Strategies

In the last example some assumptions were made on the traversal state element, which need to be discussed in detail. In general there are two kinds of elements namely

- **Replaced elements**
  Each new value replaces the old value of the element.
- **Accumulated elements**
  New values are stored, while old values are kept. The best example is a number of affine transformations stored in an element. At each stage the new transformation is accumulated with the existing transformation while the original data is also stored.

This design is used in OpenInventor[Inv92]. One other important classification when talking about traversal state elements is whether an element refers to one other node or to many other nodes. In *CashFlow* elements are needed that are able to provide a many–to–many connection between data sets and visualization algorithms.

- **Reference to one node**
  The element stores only one reference to another node at a time. It is a *replaced element*. Once a new data node registers itself to the element the reference to the first node is overwritten.

  For example the left image in figure 3.5 shows node A setting its reference to the element E. The right image in figure 3.5 shows node B replacing the reference to node A with its own reference.

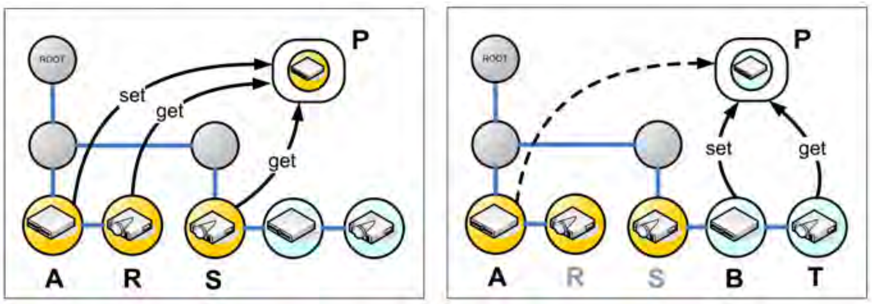

Figure 3.5: CashFlow: Implicit Linking of Nodes using an Element: (Left) Node A stores data. It registers itself to the state traversal element E (set). Node R renders the data from node A by requesting data from element E (get). Node S also renders the data from node A using object P (get). Right: Node B stores different data. Registering itself to element E (set) also removes the link to node A (dashed line). Node T now renders data from node B (get).

- **Reference to many nodes**
  This element stores many reference to other nodes at a time. Assuming such a multi–reference element node B in the right image of figure 3.5 could add its reference without erasing the reference to node A.

  Once the traversal state element stores multiple reference it is an *accumulated element*. The difference between an accumulated element and the element we require is, that we need an element which grants access to all its references at all times. A new question arises which is: "Who to query for a certain reference in such an element?"

  To solve this problem two pieces of information are used per reference:

  - **Unique name**
    Each reference is bound to a unique name.
  - **Order of insertion**
    When inserting a new reference its sequence is stored too.

  Using both unique name and order of insertion enables us to provide direct and implicit linking simultaneously.

  - **Unique name for direct linking**
    A node can query the persistent object using the unique name in order to receive the reference to a certain data node. This can be thought of as a field connection between nodes.
  - **Order of insertion for implicit linking**
    When inserting a new reference the sequence is stored too. Another node can query for the last inserted object or for the last but one inserted object etcetera.

Using such a customized accumulated element makes it possible to link different data sets with several visualization algorithms. We are also able to use such an element as a buffer for data sets. Several visualization algorithms require multiple input data. By swapping references inside the element we can easily change the visual results.

3.4 Using Virtual Arrays as a Filter

The minimal visualization pipeline in *CashFlow* consists of three components *Loader*, *Data* and *Render*. Raw data is imported from storage using the loader. The render node accesses the Data node and creates a visualization. Such a minimal pipeline is used in figure 3.6 page 47 visualizing the space shuttle data[6] directly with a point render node. The data set consists of 9 different curvilinear grids. Three grids are visualized namely the space shuttle using green dots the external tank in red and two solid rocket boosters in blue. Only the most inner layers of each grid are visualized.

The whole dataset is stored in one large array. It is important that each grid being inside that large array can be easily addressed and accessed. Our approach was to create

[6]dataset from NASA at: http://www.nas.nasa.gov/Research/Datasets/data_sets.html [NAS05]

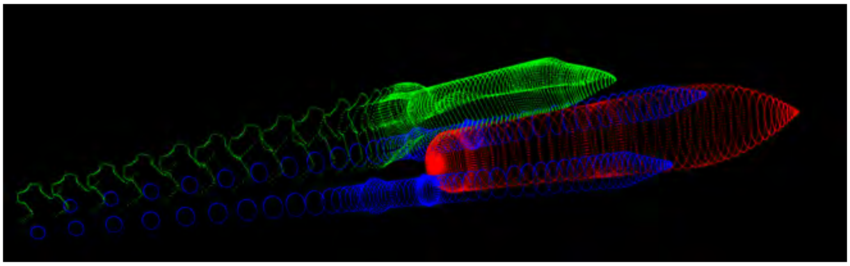

Figure 3.6: CashFlow: Space shuttle visualized with PointRenderer: 3 different curvilinear grids visualized with PointRender node. Space shuttle in green, external tank in red and solid rocket boosters in blue.

virtual arrays inside the real array in memory to define the grids and access them via an interface. All nodes handling data inside *CashFlow* use that interface. These nodes that access the data are *DataConsumer* nodes and the interface is the *DataAccess* node. Using three different types of virtual array provides even more flexibility. The other important argument for a virtual array defined inside a node is, that we are able to manipulate the DataAccess nodes inside the scene graph. The virtual array can even be defined in the OpenInventor script files.

Now we are able to set virtual arrays in a script files defining the data flow. Also we can change the data flow during runtime by altering the scene graph without changing the visualization. The basic components of the visualization pipeline

- **filtering data**,
- **mapping** and
- **rendering**

can be changed independently by manipulating nodes in the scene graph. The process flow of the virtual array is visualized in figure 3.7 on page 47. Raw data is loaded from storage to memory. The *DataAccess* node defines a virtual array by defining a filter that is evaluated at runtime. The *Render* node accesses the virtual array and generates the visualization.

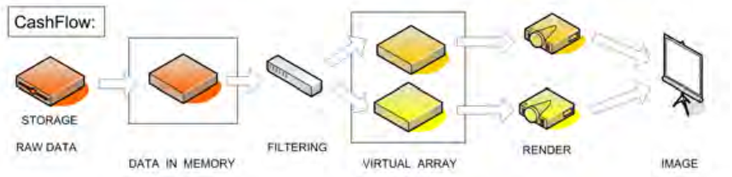

Figure 3.7: CashFlow: Virtual array process flow

Even selected parts of the grid can be rendered, once we apply a virtual array only by altering the DataAccess node as shown in figure 3.8 on page 48. The lower image shows the same dataset with selected regions of the upper visualization. A small band of green points along the side of the shuttle in one principle direction. A selection of red points of the external tank equally spaced as in the original image. A wide selection of blue points to visualize the solid rocket booster and its dense parts of the grid.

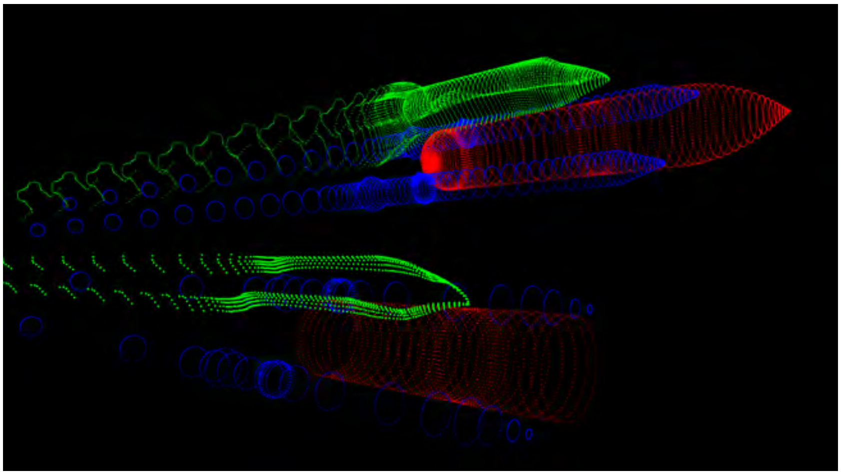

Figure 3.8: CashFlow: Space shuttle visualized with PointRender node and a selection of the data.

When handling a virtual array several strategies are useful which are:

- **Single Block Array**
  where one contiguous block in the real array is the virtual array.
- **Multiple Block Array**
  using several contiguous blocks in the real array to form the virtual array. Each block consists of a certain number of successive fields and a number of successive skipped fields.
- **Random Access Array**
  combining random fields of the real array to build the virtual array.

The three types of virtual arrays will be discussed in detail in section 3.7.1 on page 54 For ease of use iterators are also introduced to access the virtual array.

3.5 General CashFlow UML Diagram

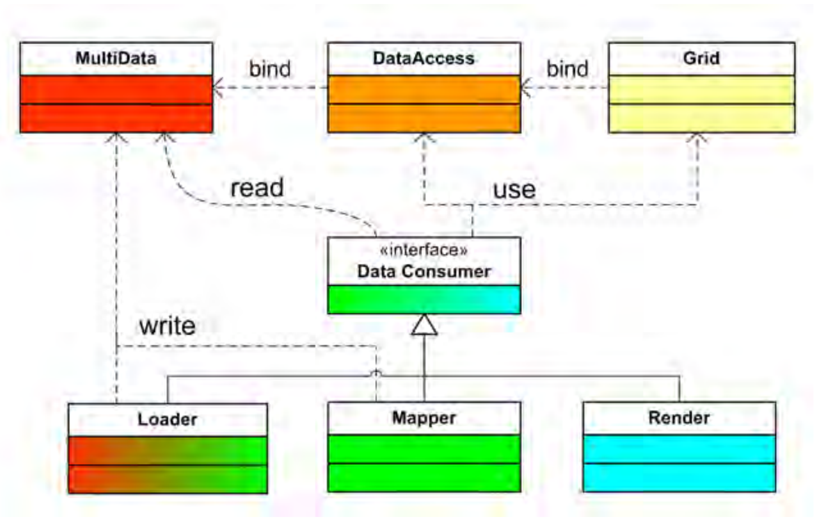

Figure 3.9: Simplified CashFlow UML diagram showing data access. All nodes accessing data are derived from Data Consumer. A DataConsumer uses a virtual array provided by the DataAccess node to access the actual data inside the MultiData node. Topology and connectivity information is offered by the Grid node. The Loader inserts data from files into the MultiData node (write access). The Mapper nodes create new data by processing existing data (read/write access). Render nodes read data only and create images. Complete CashFlow UML diagram in figure 4.5 page 71.

When looking at the *CashFlow* framework using a top–down approach all nodes can be separated in *Data nodes* and *Data Consumer nodes*. These nodes are sketched in an UML diagram in figure 3.9.

- **Data nodes**
  are used to store and access data.
  - **MultiData nodes** are used to store any type of data.
  - **DataAccess nodes** define the virtual array and provide *virtual array iterators*. These nodes always link to a MultiData node the virtual array is defined on.
  - **Grid nodes** specify the type of grid the data is defined on. Grid nodes also provide *grid iterators* used for accessing several different grids using one interface. A Grid node always links to a DataAccess node.

  The *DataConsumer* nodes can either handle grids or access data without referring to a grid structure. If grids are processed a reference to a grid node and a reference to a DataAccess node is required. The DataAccess node itself links to a MultiData node. On the other hand if only data is manipulated one reference to a DataAccess node is sufficient. As already mentioned the DataAccess node refers to the MultiData node. A sample scene graph is shown in figure 3.10 on page 51 containing all three *data nodes*.

- **Data Consumer nodes**
  are nodes handling data. The Data Consumer node is an abstract interface only.

  Data Consumer nodes can be subdivided into:

  - **Loader nodes** inserting data into a data node and therefore have write access only.
  - **Mapper nodes** accessing data which is processed by algorithms generating new data. Accordingly these nodes have read and write access.
  - **Render nodes** are defined as nodes reading data only and generating images. In general no other data output is created.

  All *Data Consumer* nodes link either to a *DataAccess* node only or to a DataAccess node and a Grid node, if topology of the grid is also requested by the algorithm.

- *Traversal state objects*
  are the third group of objects handling traversal state changes while processing the scene graph.

  Note, that *traversal state objects* are not part of the scene graph but store attributes during the scene graph traversal.

  - *Data node elements*
    used for implicit linking of nodes and for dynamic linking between MultiData nodes, DataAccess nodes and Grid nodes.
  - *Refinement elements*
    keeping parameters for grid resampling and interpolation refinement like for instance used by streamline integration algorithms.
  - *Update Action*
    to specify, when a loader node should reload the data or an asynchronous mapper node should process the data.

In the following sections all nodes will be described in detail.

3.5.1 Example for CashFlow Scene Graph

In the previous section the scene graph was simplified, because not all nodes and concepts had been introduced at that time. In section 3.2 on page 42 a simplified scene graph was used in combination with the visualization of abstract data mapped onto a polar and a Cartesian grid (see figure 3.2 page 42).

Figure 3.10 on page 51 shows the correct *CashFlow* scene graph to figure 3.2 page 42. Node (1) is a MultiData node charged by a Loader node (4). The Loader also sets new values in the DataAccess nodes labeled (2) and the defines the type of

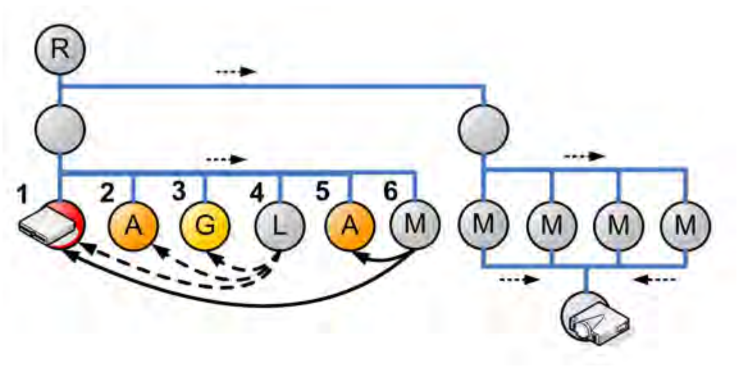

Figure 3.10: CashFlow: Scene graph for figure 3.2 on page 42. (color of node correspond to fig 3.9)

grid in the Grid node (3). Since the visualization uses a color map generated from the original data the color mapper node (6) requests the data from MultiData node (1) via the DataAccess node (2). The ColorMapper node also creates new color data stored in a new virtual array. The new virtual array is parameterized by the DataAccess node (5) storing its data in MultiData node (1).

The traversal of the right half of the scene graph is very similar to the scene graph in figure 3.3 page 43. The four mapper nodes labeled M generate polar and Cartesian height fields defining new virtual arrays in DataAccess node (2) and also store the data in MultiData node (1). Finally each of the four grids is visualized by one render node. The Render node requests the vertex data via DataAccess node (2) and the color map via DataAccess node (5). Note, that both DataAccess nodes link to the same MultiData node in this example. One render node is able to create all four different visualizations shown in figure 3.2 page 42, because all grids are so called structured grids.

3.6 MultiData Node

The MultiData node acts as an attribute supplier. Inside the MultiData node the current data is stored. The MultiData node is a container for several different arrays, each specified for a single data type. The supported data types are:

- integer
- string
- float
- vec2f
  consisting of two float values.
- vec3f
  consisting of three float values.

The MultiData node is accessed by the DataConsumer node using the DataAccess node as described in figure 3.9 on page 49. The DataConsumer node receives the necessary information on the virtual array from the DataAccess node. Using that information the DataConsumer node can read and write data to the virtual array.

In order to be able to render the grids properly and be able to use mapping algorithms on different grids, the type of grid must be taken into account also. The naive approach would be to implement the same mapping algorithm for different grids, which would result in a large number of classes doing basically the same thing. To avoid such a bad design we separate the topology data from the vertex data. The information on topology is stored in the grid node.

3.6.1 Scripting, Unique Key Concept

Using a scene graph also carries some inconvenience. Well known concepts like shared memory are not so easy to port to scene graphs, because the scene graph should remain scriptable. To provide this some conditions have to be met.

- **Insert generated data into existing node**
  Nodes creating new data, that needs to be inserted into the scene graph must use existing storage nodes. Otherwise it is not possible to refer to that data, which is generated during runtime in a script. For example let's assume that node G generates new data and node A is traversed after node G. If another node D is inserted into the scene graph and node G stores the generated data into node D it is trivial to link node A to node D.

  Thus the condition is, that each node generating data is not allowed to create a new node inside the scene graph. Each node generating data must use an existing MultiData node to store the data.

- **Bind DataAccess node to MultiData node**
  Data stored in a MultiData node is always associated with a corresponding DataAccess node. The DataAccess node holds information on the virtual array defined on top of the real memory. Instead of addressing a MultiData node directly the DataAccess node is accessed.

- **Define a unique key per node**
  Each MultiData node and each DataAccess node have a unique key. Each DataAccess node stores the key to the MultiData node it is bound to. The DataConsumer node or rather its derived classes, stores the key to the DataAccess node. This level of indirection allows to use the DataAccess node as a filter executed during runtime.

3.7 DataAccess Node

The functions of the DataAccess node are:

◇ providing information on how to access the virtual array

◇ dispatching the reference to the MultiData node, where the raw data is stored.

◇ supplying suitable iterators for accessing the virtual array.

It is a level of indirection between the MultiData node holding the data and the DataConsumer node accessing the data. The DataAccess node is a filter evaluated at runtime. This node enables the user to select parts of the data and change the selection on the fly.

It consists of the following components:

- **unique key**
  used to identify this node. All DataConsumer nodes use this unique key to get access to this node.
- **key to MultiData node**
  define the MultiData node this node is bound to. The definition of the virtual array is based on the real data stored in the MultiData node.
- **offset**
  form the first element of the real array to the first element of the virtual array.
- **length**
  number of values in the virtual array.
- **type of virtual array**
  The information is needed to define the corresponding type of virtual array iterator. Based on the type of virtual array special attributes are required like:
  - **increment**
    used for Multiple Block Virtual Array
  - **repeat**
    used for Multiple Block Virtual Array
  - **random access lookup table**
    used for Random Virtual Array

3.7.1 Virtual Array Types in CashFlow

For more flexibility three different kinds of virtual arrays are available. Since they act as data filters evaluated at runtime it is important to keep their abilities and weaknesses in mind. Table 3.1 on page 58 gives an overview on the memory consumption and CPU load of each virtual array type. The three different kinds of virtual arrays encapsulate in the DataAccess node are:

◇ **Single Block Virtual Array**

◇ **Multiple Block Virtual Array**

◇ **Random Virtual Array**

Each of the three types has its own iterator. All iterators use the same interface. The following attributes are used for all virtual arrays:

- **length**
  Number of items inside the virtual array.
- **offset**
  Number of items from the first item in the real array to the first item in the virtual array. Possible values are in the range [0..(real array length-1)].
- **key**
  A unique string used to refer to the real data array, which is stored in a MultiData node.

Additional attributes at runtime used by iterators:

– **index**
  Position in the virtual array used by iterators at runtime.
  Possible values are in the range [0..(length-1)].

– **real index**
  The index in the real array is computed either from the index in the virtual array or is returned from the iterator.

Single Block Virtual Array

This iterator is very simple. It defines a block of successive values as a new virtual array shown in figure 3.11. The length of the virtual array is equal to the length of the block of the real array. The real index can be computed by adding the offset to the virtual index. The range of the virtual index is [0..5]. The difference between the virtual index and the real index is constant while the virtual index increases.

Figure 3.11 shows an example of a single block virtual array. The attributes for figure 3.11 are:

- offset = 3
- length = 6
- unique key

The blue colored boxes in figure 3.11 indicate the virtual array array items in the real memory array. The lower part of the sketch shows the virtual array.

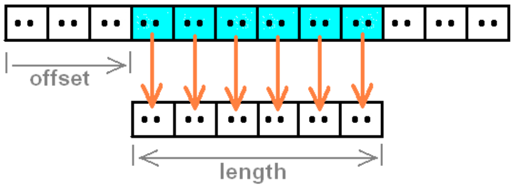

Figure 3.11: CashFlow: Single Block Virtual Array: One continuous block in the real memory is used as virtual array. Attributes are the offset and the length of the virtual array.

Multiple Block Virtual Array

This iterator requires two additional values which are:

- **increment**
  Is the number of items from the beginning of one block to the beginning of the next block. In figure 3.11 increment = 3, because on block consists of 3 items.

- **repeat**
  Is the number of successive items inside a block being part of the virtual array. For example repeat = 2 in figure 3.11.

Figure 3.12 shows a multiple block virtual array with the following attributes used:

- offset = 2
- length = 6
- unique key

  additional attributes:

* increment = 3
* repeat = 2

The parameter *length* defines the number of items inside the virtual array. Figure 3.12 sketches a real array of length 12 and a virtual array of length 6. The multiple block virtual array defines several successive block were each block has a constant size. The constant size of the blocks is addressed as *increment*. The range of the virtual index is [0..5]. Inside of each block a constant number of successive items are accessed and the corresponding attribute is *repeat*. In figure 3.12 these items are colored blue.

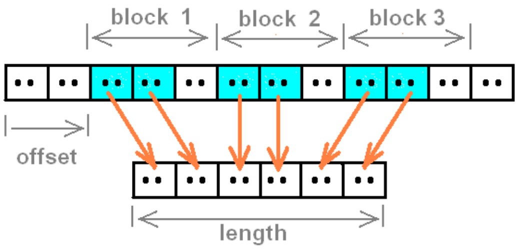

Figure 3.12: CashFlow: Multiple Block Virtual Array: Several successive blocks are reorganized as a virtual array. Attributes are the offset the number of blocks, the size per block and the number of values read per block.

To compute the real index from the virtual index a division or modulo operation is needed. Thus iterators are useful and random access to a virtual index is more expensive to compute compared with the *single block virtual array*. The difference between the virtual index and the real index increases while the virtual index increases. If the parameters *increment* and *repeat* are the same, the multiple block degenerates to a single block virtual array. This should be avoided and the single block virtual array should be used, because of its better performance.

Random Virtual Array

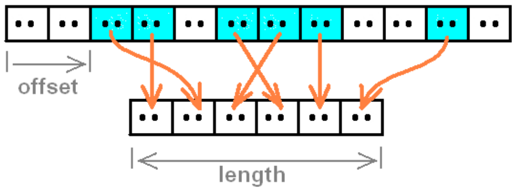

Figure 3.13: CashFlow: Random Virtual Array: Random fields of the original array create the virtual array. Attributes are a list of indices providing information on the random mapping.

The random virtual array requires one additional array of attributes.

- **look–up table**
  Each virtual index is mapped to one real index. Even more than one virtual index may point to the same real index. Accordingly the number of items in the lookup

table must be equal to the *length* of the virtual index.
Note: The index in the real array is computed by adding *offset* to the values from the *look–up table*.

Attributes used for figure 3.13 are:

- offset = 2
- length = 6
- unique key

  additional attributes:

* look–up table
Size of the lookup table is equal to the length of the virtual array.

The virtual index is mapped to the real index by using the index stored in the look–up table. See figure 3.13 showing an example for a random virtual array. The offset attribute is used to suite the virtual array interface.
The range of the virtual index is [0..5].

Virtual Array & DataAccess Map

The detailed information on the virtual array is encapsulated inside an object called *DataAccess Map*. The *DataAccess Map* is used in the interface of the virtual array. All nodes requesting data, like the DataConsumer nodes, query a DataAccess node and receive a *DataAccess Map*. Based on the information provided by the *DataAccess Map* the DataConsumer nodes can instantiate the appropriate virtual array iterator and finally access the data inside the MultiData node. Thus the DataAccess node provides an interface to the data for the DataConsumer node.

3.7.2 Drawback of the Virtual Array

The overhead using the virtual array consists of:

- Managing the DataAccess and MultiData nodes and the necessary *DataAccess Element* and *MultiData Element*. During each scene graph traversal each DataAccess node and each MultiData node has to register itself to its Element (details in section 4.1 on page 68).
- Each DataConsumer needs to query for the DataAccess nodes before being able to access the data itself. This is a small overhead, as long as the number of DataAccess nodes is comparable small to the size of the data. In case of a large number of DataAccess nodes addressing just very little amount of data, this could cause a significant overhead.
- Additional computation using the iterators

  depending on the type of virtual array being in use (see table 3.1 page 58).

type of virtual array	size in memory	attributes stored
Single Block Virtual Array	O(1)	3
Multiple Block Virtual Array	O(1)	5
Random Virtual Array	O(n)	n+3

Table 3.1: Virtual array strategies: Comparisons of memory overhead and CPU load for three different strategies.

Table 3.1 gives an overview on the overhead using the virtual array. Each type of virtual array has certain memory requirements. While single block and multiple block virtual arrays only require constant complexity per allocated virtual memory the size of memory needed for a random virtual array depends on the size of the virtual array which is linear complexity. All virtual arrays store two integers, *offset* and *length*, and a string used as *key* for linking to the real data array. A *single block virtual array* uses only the three previously mentioned attributes. A *multiple block virtual array* additionally uses two integers, *increment* and *repeat*, to define the size of a block and how many successive items per block should be used inside the virtual memory. Finally the *random virtual array* uses the three default attributes and a *look–up table* of the size *length* to store one real index for each virtual index.

3.8 Grid Node

The Grids node is a geometry and topology supplier. The Grid node provides an universal interface together with special iterators to access different grids in a standard way.

Type of Grids

Using grids to handle scientific data allows three types of grids:

- **Structured grid**
  defined as a n–dimensional array. The index in the array also implies neighborhood information. Adjacent vertices can be calculated directly based on their index. One of the basic examples in 2D is a regular 2D Cartesian grid. One example in 3D is a 3D curvilinear grid.
- **Unstructured grid**
  consisting of vertex information and lists defining geometric primitives by indexing the vertices. Neighborhood information has to be either calculated as a pre–processing step or it has to be stored additionally. Well known examples are:
  - **polygonal triangle grids** generating a 2D surface in 2D or 3D and
  - **tetrahedral grids** in 3D.
- **Hybrid grid** as a combination of the above mentioned grids. One common way to avoid hybrid grids is to split grids into parts and separate structured grids from unstructured grids. The big disadvantage of hybrid grids is, that combining structured grids with unstructured grids erases the advantage of structured grids. On the other hand hybrid grids have the advantage, that they are a very universal definition.

  Hybrid grids are not supported by *CashFlow*.

3.8.1 Support Several Different Grids

One main problem is, that there is a large amount of grids available and in use. If each algorithm has to be implemented on each grid or be adapted to the grid, the number of classes implemented, derived from the general algorithms would be huge. One other disadvantage of this approach gets obvious, if a new grid is introduced to the framework. In order to be able to render and process the grid, all existing algorithms have to be derived to be able to access this new grid. This approach was used in MeshVis [Mes] by TGS [TGS] former known as DataMaster.

Our approach is to abstract the pieces of information from the grids needed by the visualization algorithm to access the grids. Information provided by the grid include *topology data*(1) and *grid iterators*(2) for accessing the *geometric primitives* of the grid. The visualization algorithm only communicates with the grid via this interface. Once a new grid is introduced to the system, the new grid only has to provide the topology data and overload the iterators defined by the interface. Also if a new algorithm is implemented it can access all grids via this abstraction layer. Specialized algorithms optimized for speed defined on one specific grid can also be implemented. This concept is similar to the approach introduced by Patrick Moran [Mor01], who created the FIELD MODEL library [Mor03] at NAS[NAS] which is a C++ template library. It was also published at IEEE Vis2001 in Tutorial 1 [Tut01].

Interface for accessing grids:

- **Geometric primitives**
  embedded in 3D
  - **Vertex** zero dimensional; Point in 3D
  - **Edge** one dimensional; Line.
  - **Face** two dimensional
  - **Cell** three dimensional
- **Connectivity information**
  contains adjacency of Edge, Face and Cell. Each connectivity information can be queried using the interface.
- **Grid iterators**
  Grant access to each cell, each face and each edge for the algorithms.

For most pairs of grids and visualizations special ways of fast rendering algorithms have been published.

3.8.2 Decoupling of Visualization Algorithm and Type of Grid

In order to decouple the visualization method from the grid type we use the Design Pattern [Gam95] "Strategy" (page 373 in [Gam96]) . Not all combinations of grid types and visualization techniques may be useful and possible, but never the less a separation is very important.

The design pattern is applied for the *visualization techniques* (see section 2.2 page 19) and for the *grid types*.

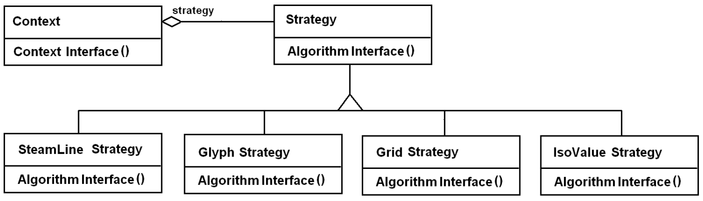

Figure 3.14: Design Pattern: Strategy [Gam95] [Gam96]

3.9 Mapper Nodes

Mapper nodes are used to transform given data into new data. The Mapper node have read and write access to the MultiData node and DataAccess node.

Executing a Mapper node can be triggered by:

- **Update field**
  Some Mappers are only used for pre–processing and will be triggered by a field connection.

- **Update action**
  Some Mappers implement costly algorithms like streamline integration of iso–surface creation and thus are triggered by update actions.

- **Listener on virtual array**
  Since it may be useful only to update a Mapper if certain data has changed, it is useful to trigger the Mapper as soon as the data in the virtual array has changed.

- **On–the–fly**
  Other Mappers may be executed every time the scene graph is traversed. This could be reasonable for the generation of a color–map for instance.

Mapper nodes are very important, because they implement a large number of visualization algorithms. The complexity of Mapper nodes can be categorized in 3 levels:

- **Low complexity**
  All algorithms operating directly on the raw data without support of grid topology. Examples are color mapping and parameterization of data.

- **Medium complexity**
  All algorithms operating on the raw data with support of grid topology. Examples are iso–surface extraction.

- **High complexity**
  Algorithms requesting grid topology data and results form the medium complexity level as input. Examples are vortex detection and flow field topology calculation, which are not supported by *CashFlow* yet.

After this introduction to the four kinds of update mechanism and the subdivision based on the complexity of the algorithm we want to describe the different types of Mapper nodes used in the *CashFlow* framework.

- **Color Mapper**
  Scalar or vector data is mapped to color. These are low complexity algorithms.

- **Data Mapper**
  Data is used to create new data without respect to the type of grid and its topology. These are also low complexity algorithms.

  - **Parameterization Mapper**
    Data is reparameterized with a set of constant, linear and cubic intervals.

- **Geometry Mapper**
  This large group of mappers include algorithms operating on grids.

  - **Iso–value Mapper**
    Creation of polygonal surfaces using 2D & 3D cells. A scalar value called iso–value is defined as a parameter. At each vertex in each cell a scalar value has to be defined, which is used to create the polygonal surface or the iso–line in 2D.
  - **Carpet Plot Mapper**
    A field of scalar values or a vector field is used to distort a surface.
  - **Streamline Mapper**
    This mapper is responsible for all kinds of particle tracers and streamlines integration using seed points.
  - **Cutting Mapper**
    Cutting planes and surfaces are used to segment data.

Only one group of algorithms are not implemented as mappers namely the rendering algorithms.

3.10 Render Nodes

This group of nodes implement rendering algorithms. The render nodes request data from the DataAccess nodes and from a Grid node. Per definition Render nodes create images only and do not create new data, that is stored in a MultiData node. If a Render node also creates data, that is stored in a MultiData node it has to be derived from Mapper node.

The Render algorithms can be subdivided as follows:

- **Point Cloud Renderer**
  The most basic Render node draws a point where a vertex is. The point can also be color coded. This visualization can be useful for example to investigate the density of a grid.

- **Glyph Renderer**
  This is the complement of the point renderer in vector fields. At each selected point a glyph shows the direction of the flow. Glyphs can be also used to visualize multidimensional data.

- **Cell Renderer**
  The grid is divided into cells and each cell is rendered. One traditional cell renderer is the cuberille renderer [LC85].

- **Wireframe Renderer**
  The grid or parts of the grid are visualized using a wire–frame.

- **Surface Renderer**
  This renderer is similar to the wire–frame renderer. The grid is rendered using surfaces.

- **Streamline Renderer**
  Rendering of streamlines in different ways. In extension to the default streamline renderer an up–vector per each interpolation point can be taken into account to visualize vorticity.

- **Polygonal Renderer**
  Simply renders polygonal data.

- **Volume Renderer**
  CashFlow does not use its own volume renderer, because a volume renderer called SimVoleon[Vol10] is already available in Coin3D[Coi00].

3.11 Loader Nodes

Unfortunately there are several binary data formats for CFD data sets and medical data sets. Due to the large dissemination of some packages a set of data formats became more important. This is far away from a standardization of data formats, but it also reduces the number of loaders needed.

Since it is not useful to implement several loaders to proof that a visualization algorithm works correct we decided to use only two loaders at the moment:

- **CFD Plot3D**
  for computational fluid dynamics (CFD) data sets

- **CT data set**
  for medical data sets.

3.12 User Interaction

When using the scene graph library a wide variety of input and output devices is offered to the user. Subject to the data source in general the user can alter the attributes of the data generation or data simulation. When using a CFD – systems[7] the change of parameters is a highly non–trivial task. Thus a user friendly interface is essential for quick results. In terms of the visualization pipeline the user can change the raw data. If this should be not possible he can still adjust the parameters used inside the *CashFlow* framework, which are the attributes for the *Filter*, *Mapper* and *Render* nodes.

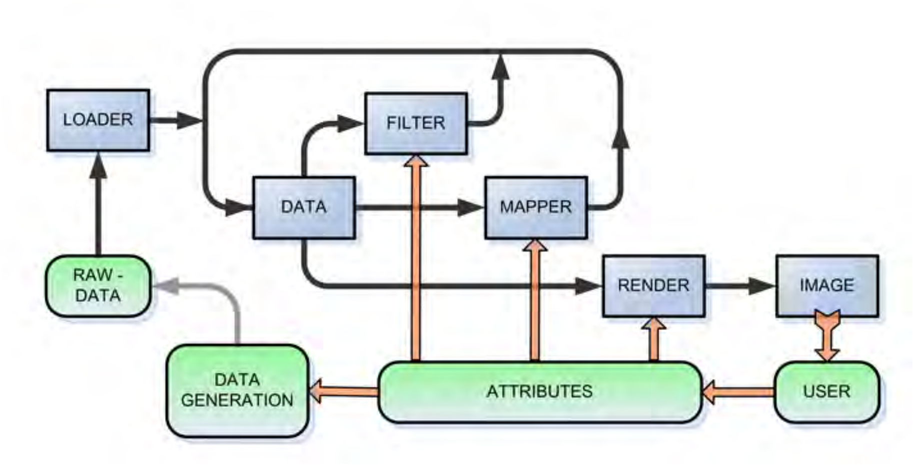

Figure 3.15: CashFlow: Boxes and grey arrows show process flow. Round boxes and tailed arrows indicate user interaction (UI) process. (compare to fig 3.1)

The process flow of external data generation in combination with *CashFlow* is shown in figure 3.15. In this figure the blue boxes show components of *CashFlow* while the green rounded boxes in the lower part of the figure indicate components altered by the user while interacting. The *Loader node* imports the raw data into the *CashFlow* framework. The data may be processed now by a *Mapper node* before it is converted into the final image by *Render node's*. Now the user can either alter the attributes for *CashFlow* to generate a new image or the user may even be able to change the attributes of the data generation producing new raw data.

Note, that this process flow is an extension to figure 3.1 on page 38, where the blue boxes were used only.

[7]CFD...computational fluid dynamics, numerical simulation of fluids

3.13 Conclusion

In this chapter the following concepts have been introduced:

- **The *CashFlow* visualization pipeline**
  This visualization pipeline is an adaption of the traditional visualization pipeline. It is designed to suite the special needs of a dynamic data flow model crafted into a scene graph with respect to scripting. The main components *LOADER – DATA – MAPPER – RENDER* used in *CashFlow* were introduced. Each of these components are defined in order to make scripting the OpenInventor files easier.

- **Virtual Array**
  The virtual array is implemented in the DataAccess node. It is a filtering at run-time with small overhead depending on the type of virtual array. The virtual array can be accessed and handled easily using one of three *virtual array iterators*.

- **Indirect References**
  To be able to use absolute and relative linking between nodes a special traversal state object is needed. On the one hand this traversal state object allows absolute linking of nodes similar to field connections. On the other hand it also provides relative or implicit linking of nodes.

  While the scene graph is traversed the ordering of nodes have an effect on which nodes are linked together. This concept is only possible by applying implicit linking in addition to absolute linking.

- **Unique keys**
  To be able to address nodes in the script file unique keys are used. The unique key is a simple string, which has to be unique during the scene graph traversal. Two concepts are used:

  Absolute address key
  creating a direct node–to–node connection. The data source is addressed by its unique name comparable to a field connection.

  Relative address keys
  taking traversal order of nodes into account.

- **Decoupling of algorithm and type of grid**
  By separating the topology information of a grid from the other data of the grid algorithms can be defined to access several different types of grids instead of only one type of grid. The disadvantage of that approach is, that this additional layer slows down the algorithms.

- **Dynamic Data Flow Model** The components of *CashFlow* can be used to create a dynamic data flow model. These data flow models can be scripted and actually be altered at runtime.

Chapter 4

Implementation

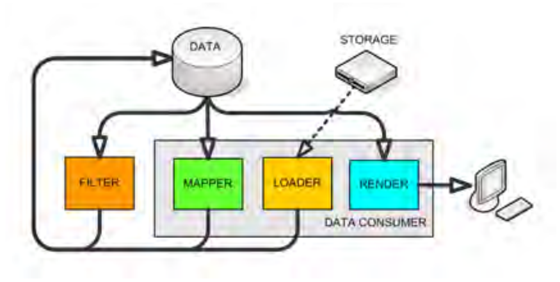

Figure 4.1: CashFlow Data flow model

The general class hierarchy of *CashFlow* is shown in figure 4.1 and will be introduced in the following chapters. A full collection of all *CashFlow* classes is listed in section 4.7 on page 97. The concept of filter, mapper and render nodes used on the following pages were introduced in section 2 page 3 and section 3 page 38. This is a brief summery of nodes used in figure 4.1, while the details were introduced in chapter 3 on page 38 .

- **SoCfDataConsumer** *(fig 4.1: Dataconsumer)*
  This abstract class provides an interface for all nodes processing data. Any output data can only be stored in existing nodes. No SoCfDataConsumer node may create new SoMultiDataNodes, SoDataAccessNodes or SoBaseGrid nodes and insert them into the scene graph. The reason for that policy is, that otherwise scripting would be difficult or even impossible.

  - **Filter**
    A Filter selects part of the raw data at runtime using a virtual array. The filter is implemented as SoDataAccessNode and SbDataAccessMap.
  - **SoCfMapperNode** *(fig 4.1: Mapper)*

  Most visualization algorithms are derived from SoCfMapperNode. A Mapper can handel several data input and data output streams.

- **SoCfRenderNode** *(fig 4.1: Render)*
  This node creates images and implements rendering algorithms. A Render node may have several data inputs. The only output of a Render node are OpenGL calls.

- **Data**
  Figure 4.1 shows the software design of *CashFlow* in general. Due to our proposed demand the implementation differs. Our demand was, that the data flow model creating the visualization network shall be scripted easily. To provide the user with maximum flexibility the data is stored in three nodes, each with its very special purpose.

  - **SoMultiDataNode** *(fig 4.1: Data)*
    All types of data are stored in SoMultiDataNodes. DataConsumer nodes process the data stored in the SoMultiDataNode using the SbDataAccessMap from the SoDataAccessNode.
  - **SoDataAccessNode** *(fig 4.1: Filter)*
    The SoDataAccessNode creates a virtual array on top of the real array stored in the SoMultiDataNode. This node is the interface to all raw data and derived data. All DataConsumers access data via this interface using a *DataAccessMap*. The DataAccessMap stores information on the kind of data, their location in the SoMultiDataNode and the type of virtual array used. (see section 4.4.1 on page 77)
  - **SoBaseGrid**
    The Grid node stores topology information for the grid as well as the location of the vertex data and additional data, if available. The Grid node also defines *geometric primitives* and *grid iterators* in order to abstract the topology data of a grid. This approach provides a general interface for accessing several grids.
    - **Geometric Primitives**
      In order to be able to use visualization algorithms on many different girds an interface for geometric properties is used. This allows to implement a certain algorithm only once and let it access all kinds of known grids. The grids also need to implement this interface.
      (see section 4.5.1 page 83)
    - **Grid Iterator**
      Due to the nature of many algorithms a list of iterators are introduced to access the grids via geometric primitives. Naturally not all grids can provide all iterators. Thus the number of grids used by one algorithm is limited. (see section 4.5.2 page 83)

4.1 Accessing Data

Using scene graphs allows two ways to access data in general. Either one node is linked directly to the other node (1), in case these nodes want to exchange or share data, or a dispatcher is used (2). The dispatcher collects requests and connects the the nodes indirectly. In OpenInventor these two concepts are called:

1. **Field connection**
2. **Action & Element**

Two nodes can exchange data via field connections. Each node stores information in a container referred to as *field*[1] In order to share that information the field of one node is to connected the other node. In the Open Inventor script file this is done by directly connecting the fields of the nodes. The first prototype of the *CashFlow* framework was based on this concept shown in figure 4.2. In order to create data based on existing data so called *SoEngine's*[2] were used. The main disadvantage of *SoEngine's* in general is, that they are not part of the scene graph traversal. Thus *SoEngine's* can not be triggered like other *SoNode's* can. The disadvantage of the *field connection* approach is, that each node holding data has to be explicitly connected to each *Data Consumer* node.

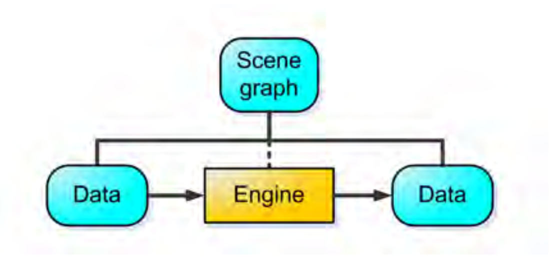

Figure 4.2: CashFlow old design: Mapper implemented as Engine. Replaced by "Action & Element"–concept, because of the encountered limitations.

Relying on *field connection* only makes it very difficult to script the scene graph. Due to the encountered limitations of the *field connection* approach (using SoNodes & SoEngines) we decided to implement a second prototype. The second prototype of *CashFlow* and therefore *CashFlow release R1* is based on the *Action & Element concept*.

[1]for details on fields see [Wer93] and [Wer94] chapter 3
[2]see Coin3D SoEngine and [Wer94] chapter 6

4.1.1 Action & Element

While the scene graph is traversed special attributes and parameters for the traversal state are needed. These attributes are stored in so called *Elements*[3]. The traversal of the scene graph itself is called an *Action*[4]. *Actions* are either triggered by events or created repeatedly like the *GLRender action* responsible for rendering. Once an *Action* is initialized, it creates the necessary *Elements* needed. These *Elements* can be accessed by all *Nodes*[5], that are touched during the scene graph traversal.

Using this design pattern *Nodes* can exchange data without an explicit field connection. Nodes can either alter properties inside the Elements or use the Elements to exchange data. One important thing to know is, that once the action is terminated it destroys all created elements. Thus all *CashFlow nodes* have to register to their Elements each time the scene graph is traversed.

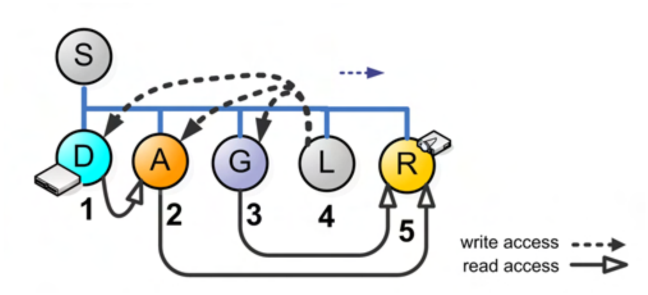

Figure 4.3: CashFlow: Accessing data. Left – to – right traversal order. Used nodes are: Separator node(S), MultiData node(D), DataAccess node(A), Grid node(G), Loader node(L) and Render node(R).

Figure 4.3 shows how data is inserted from a Loader node and accessed by a Render node. The Loader node (4) inserts data from storage into the MultiData node (1), creates a suitable DataAccessMap inside the DataAccess node (2) and updates the grid information in the GridNode (3). Afterwards the Render node (5) reads data from the MultiData node (1) using the DataAccess node (2) and uses the Grid node (3) to render the data. This image also shows one general assumption being made for scene graphs, which is the left–to–right order of nodes. The node being most left is traversed first and the node being most right is visited at last in the traversal.

[3]see Coin3d SoElement
[4]see Coin3d SoAction, SoGLRenderAction
[5]see Coin3d SoNode

4.1.2 SoElements in CashFlow

One of the core concepts of the CashFlow framework is the use of Elements to exchange data between nodes using direct and indirect linking of nodes. The Elements created for the CashFlow framework are:

- SoMultiDataElement Used to access SoMultiDataNodes.
- SoDataAccessElement Used to access SoDataAccessNodes.
- SoGridElement Used to access SoBaseGrid node.

The scene graph in figure 4.4 shows how nodes and Elements interact. Let us assume, that the traversal passed nodes (1)–(4) and that the RenderNode "R" is now active. Note, that all Elements in figure 4.4 b) are sketched as round boxes in the upper region of the image and all nodes are sketched as circles in the lower part. The RenderNode accesses the GridElement (a) requesting the reference to the GridNode(3). Using this reference from the GridElement(b) the RenderNode reads the data from the GridNode (see figure 4.4a)). The GridNode stores a unique key pointing to the DataAccessNode(2). In order to get the reference to the DataAccessNode the RenderNode uses the unique key (c) and connects to the DataAccessElement (D). The DataAccessElement returns the reference to the DataAccessNode (d). The parameters for the virtual array stored in the DataAccessMap also contain another unique key (e) pointing to the actual data inside the MultiDataNode (f). Finally the RenderNode connects to the MultiData Element*(D) and receives the desired reference to the MultiDataNode (1).

The RenderNode accesses the MultiDataNode (1) via the DataAccessNode (2) and renders the image using the topology information from the GridNode (3). This scene graph in figure 4.4a) is the same as in figure 4.3.

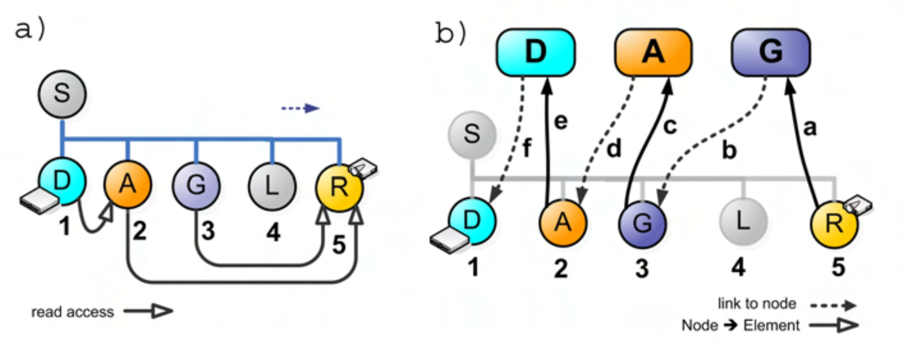

Figure 4.4: Process flow diagram: Use of DataAccess Element (1) and Grid Element (2) to exchange data between nodes. Render Node (3) reads information from Elements and access data (4) to create an image (5). Based on visualization pipeline from fig 3.1. For UML sequence diagram see fig. 4.6.

Details on CashFlow Elements

The demand for CashFlow Elements is, that they should be able to provide direct and indirect linking. This problem was solved using unique keys for *key–value* pairs on the one hand and a list storing the order of insertion for each nodes on the other hand. Once a Node registers to its Element using a unique key it is easy to store the order of insertion. The gain of that concept is, that a large number of nodes can be registered to an Element at the same time and each Node can be addressed in two ways.

Using this new concept provides the basis for a dynamic data flow network.

4.2 UML Inheritance Diagram

The UML inheritance diagram in figure 4.5 shows the core classes of the *CashFlow* visualization framework. A light–weight version of this figure was introduced in section 3.5 on page 49 without the SoElements and the derived classes.

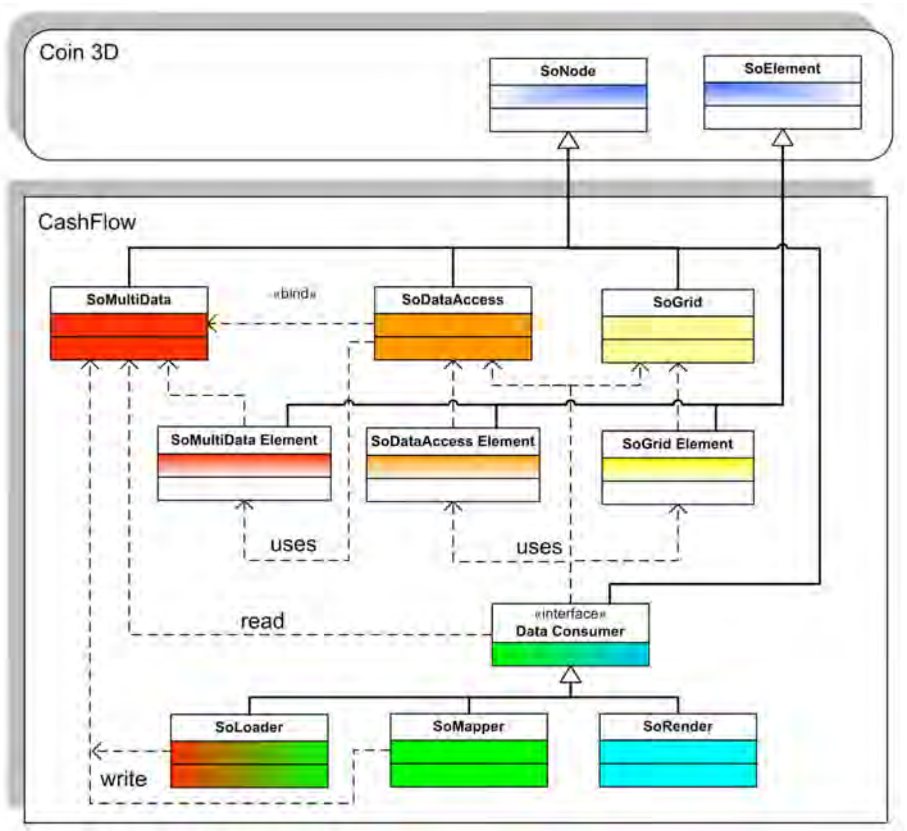

Figure 4.5: CashFlow UML Inheritance Diagram: based on the visualization pipeline from figure 3.1 on page 38.

4.2.1 UML Sequence Diagram of Data Access

The concept of **SoAction & SoElement** from OpenInventor [Coi00] is very important to understand, because the dynamic data flow model of the *CashFlow* framework is based on this concept. SoActions[6] can either be created by events or being generated repeatedly like the SoGLRender action. SoActions traverse the scene graph and use SoElements[7] to carry information from one node to another nodes. SoActions and SoElements have a life cycle including a repeated creation and destruction. Once an SoAction is initialized, it creates the utilized SoElements. The SoElements only exist while the SoAction is not terminated. As soon as the SoAction is terminated it deletes all its SoElements in a post–mortem step. All nodes touched by the traversal can access the SoElements (see figure 4.4 on page 70).

The UML sequence diagram in figure 4.6 on page 73 shows the interaction between scene graph nodes, one SoAction and three SoElements. The UML sequence diagram can be divided into four phases:

1. Initialization Phase for SoElements
2. Register Phase where nodes register to their corresponding SoElements
3. Access Phase where nodes exchange data using the SoElements as dispatchers.
4. Deallocation Phase where the SoElements are destroyed during the post–mortem step of the SoAction.

When using a larger scene graph like on page 51 in figure 3.10 phase 2) and phase 3) are passed through several times. The scene graph used for the UML sequence diagram in figure 4.6 consists of the following nodes.

$$\text{GridNode} \rightarrow \text{DataAccess Node} \rightarrow \text{MultiData Node} \rightarrow \text{Render Node}$$

The SoAction creates [init] the three new Elements Grid Element, DataAccess Element and MultiData Element. The initialization phase is now finished. The next step is traversing the scene graph. The Action calls the method [doAction] for each node that is touched by the traversal. In our simple example all nodes are effected by the traversal, since no SoSwitch nodes are used in the scene graph.

The three nodes GridNode, DataAccessNode and MultiDataNode register themselves to their corresponding Elements [register(this)]. The GridNode stores a unique key pointing to at least one DataAccessNode. The DataAccessNode holds several DataAccessMaps , which define virtual arrays in the MultiDataNodes. Last node in traversal order is the RenderNode. It queries the Grid Element for the current GridNode [getGrid] and reads the information from the GridNode [getGrid]. Based on that information from the GridNode the RenderNode requests the required DataAccessNode from the DataAccess Element [getGrid]. The DataAccessNode links to a MultiDataNode via the MultiData Element [getData]. Finally the RenderNode is able to generate the image [render]. In the last phase the Action is terminated and removes its Elements [destroy], while all nodes await the next scene graph traversal.

[6]for details on SoAction see Coin3D[Coi00]
[7]for details on SoElements see Coin3D[Coi00]

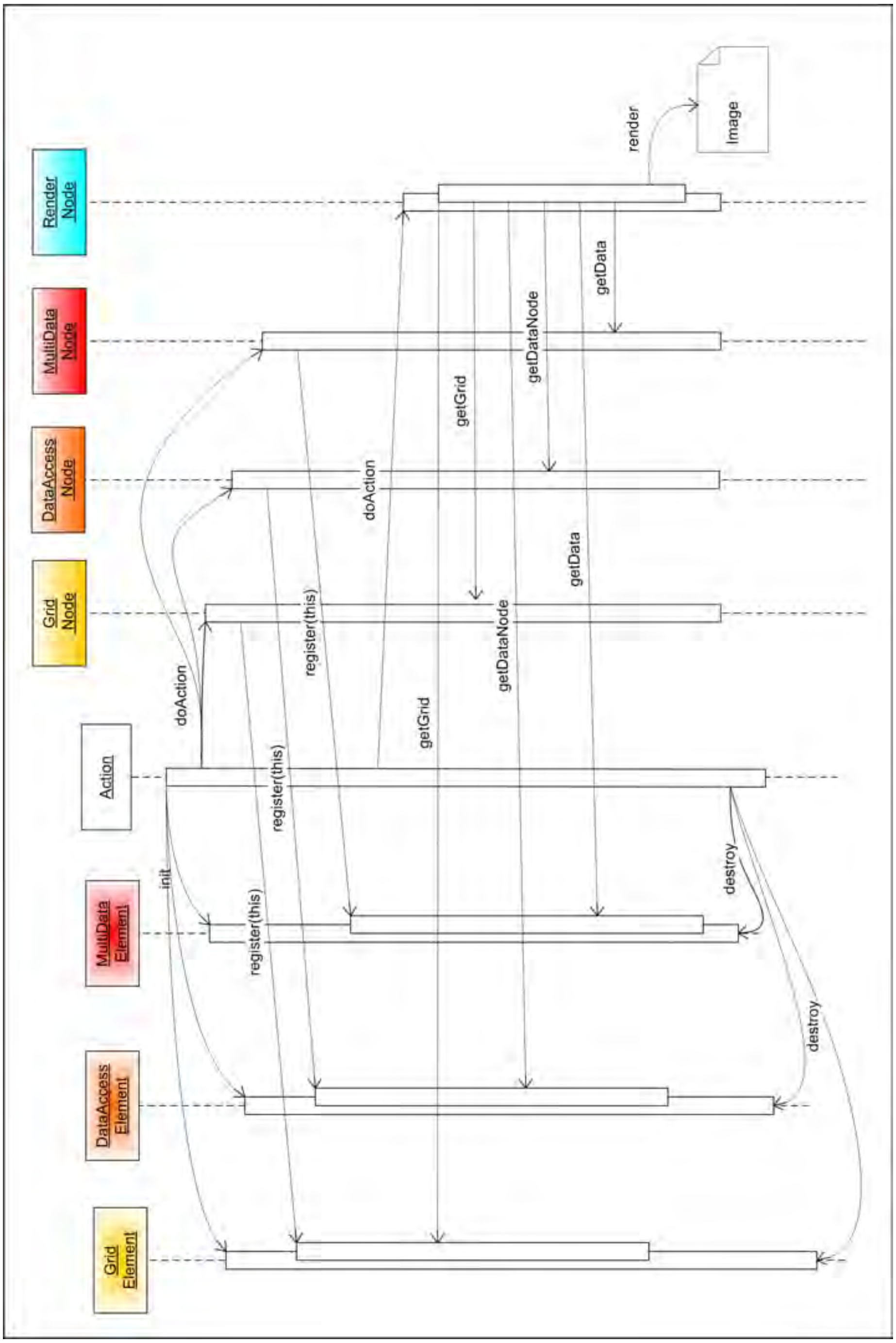

Figure 4.6: CashFlow UML Sequence Diagram: Collection of virtual array types.

4.3 SoMultiDataNode

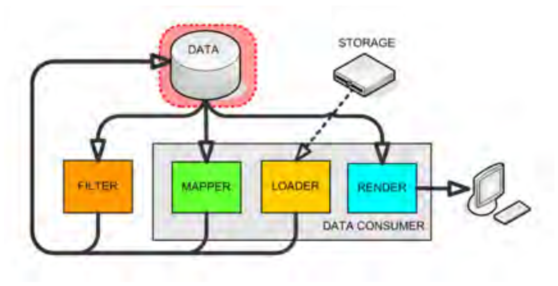

Figure 4.7: CashFlow Data Flow Model - MultiData node: Accessing virtual array using the attributes offset, inc, length, label and data type (not shown)

The purpose of the MultiData node was already mentioned in section 3.6 on page 51. This section focuses on the kind of data stored in this node.

- **Raw Data**
  This is data on permanent storage created by numerical simulations like
  - computational fluid dynamics (CFD) systems or
  - finite element methods (FEM).

  Also a lot of medical units produce reasonable data like
  - computer tomography (CT) or
  - magnetic resonance imaging – tomography (MRI)(MRT)(MR) and
  - positron emission tomography (PET). PET data is very similar to FEM data, because both create 3x3 tensors as output.
- **Intermediate / Process / Derived Data**
  Derived or intermediate data is generated by most algorithms. Often it is useful to store that data before it is processed or rendered directly. The most frequently derived data in scientific visualization are:
  - gradient, often used instead of the normal vector
  - stream lines
  - iso surfaces

  The reason for storing that information is:
  - **reuse instead of recalculate**
    Some information like gradient would otherwise be calculated again by different algorithms.

  - **expensive to compute**
    Most derived data are expensive to compute, like stream lines for instance, especially on curvilinear–linear and unstructured grids. The placement of new seed points using existing streamlines is one own field.
  - **base for other algorithms**
    Especially iso–surfaces can create new geometry as input for a large set of algorithms.

- **Render Data**
  When dealing with huge data sets a lot of techniques are used to speed up rendering.

  - **View frustum culling**
    In large scenes only a small visible part is selected and rendered.
  - **Portals and cells**
    Scenes can sometimes be divided into cells connected by portals. Only visible cells are rendered.
  - **Level of Details (LOD)**
    Objects can be rendered with different amount of details depending on their size in screen space and on the distance between the object and the camera.

4.3.1 SoMultiDataNode – File Format

SoMultiDataNode iv–file

```
#Inventor V2.0 ascii

SoMultiDataNode {
  fields [SoSFEnum    mode,
          SoSFString  keyData,
          SoSFString  label,
          SoMFInt32   int32Data,
          SoMFString  stringData,
          SoMFFloat   floatData,
          SoMFVec2f   vec2fData,
          SoMFVec3f   vec3fData,
          SoMFColor   colorData ]

  mode        ADD
     # values (NONE, ADD, SET, CLEAR, LIST)

  keyData     "my_unique_MultiDataKey"
   # value: unique absolute key ( any string )

  label       "myRaw_data"
  int32Data   [ ]
  stringData  [ ]
  floatData   [ ]
  vec2fData   [ ]
  vec3fData   [ ]
  colorData   [ ]
}
```

CashFlow iv–file

4.4 SoDataAccessNode

This node enables the user to define virtual arrays and link them to any *Data Consumer* node without copying the data in memory. It is essential for scripting to be able to define the virtual arrays in the script file. The functionality can be compared to a filter selecting parts of the data at runtime. This class is used to separate the raw data, stored in arrays, forming an interface to data. This separation allows the user to crunch different values stored in memory in various ways. The three different types of virtual arrays in *CashFlow* are *Single Block Array* (1), *Multiple Block Array* (2) and *Random Block Array* (3) as shown in figure 4.9 below. The virtual arrays were introduced in detail in section 3.4 on page 46.

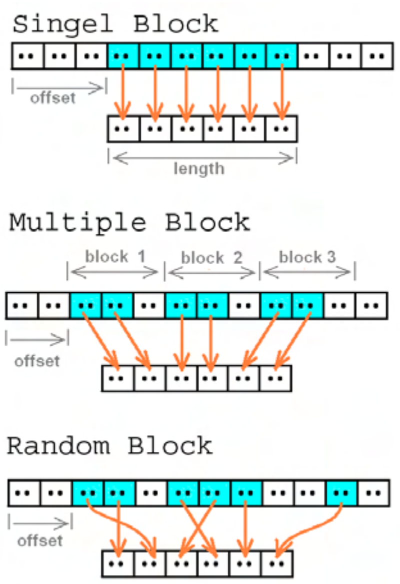

Figure 4.8: SoDataAccessNode Virtual Array Types: Collection of virtual array types.

All pieces of information, that are required to handle the virtual array properly, are stored in a data structure named **SbDataAccessMap**. *Data Consumer* nodes access the raw data stored in the *SoMultiDataNode* via a *virtual array iterator*, which was constructed from the *SbDataAccessMap*. One *SoDataAccessNode* can store several *SbDataAccessMap*.

4.4.1 SbDataAccessMap

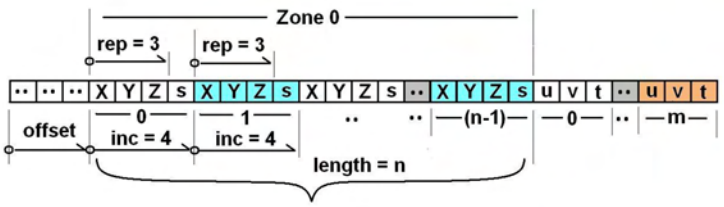

Figure 4.9: SoDataAccessNode Virtual Multiple Block Array: Accessing virtual array using the attributes offset, length, inc and repeat.

The SbDataAccessMap consists of the following attributes, which are also shown in figure 4.8 on page 77

- **mode**

This attribute handles how this SbDataAccessMap is registered to the SoDataAccessElement. The possible values are:

NONE (default)
: this SbDataAccessMap is not registered to the SoDataAccessElement.

ADD
: The SbDataAccessMap is added to the SoDataAccessElement. If the *key* already exists the SoDataAccessElement throws a warning and ignores this SbDataAccessMap.

SET
: The SbDataAccessMap is added to the SoDataAccessElement. If the *key* already exists it is replaced.

CLEAR
: This attribute removes the *key* from the SoDataAccessElement, if the *key* exists. Note, that all other parameters of this SbDataAccessMap are ignored.

CLEAR_ALL
: This attribute removes all entries from the SoDataAccessElement. Note, that all other parameters of this SbDataAccessMap are ignored.

LIST
: This attribute prints all SbDataAccessMap's from the SoDataAccessElement. Note, that all other parameters of this SbDataAccessMap are ignored.

- **type**
  Type of data from a list containing int,float,double,string

- **arrayType**
  Defines which type of virtual array the DataAccessMap defines. The three possible values are:

SINGLE	(default) Single block virtual array
MULTI	Multiple block virtual array
RANDOM	Random block virtual array

- **key**
  name of this data access map. Should be unique during traversal otherwise the SoDataAccessElement will report this as an error.

- **keyData**
  name of target MultiData node.

- **label**
  This field is optional and can be used for visual programming and debugging.

- **offset**
  from the start of the array to the point of the first item.

- **length**
  number of items, that can be accessed from the first until the last item in the virtual array (see figure 4.8). Note, that the length does not show the amount of memory used.

◇ **repeat**
Attribute of Multiple Block array only. Number of items read per block (see figure 4.8 and figure 4.10).

◇ **inc**
Attribute of Multiple Block array only. Number of item per block.

⊗ **lookupTable**
Attribute of Random Block array only. Defines the mapping of real array index to virtual array index (see Random Block in figure 4.8).

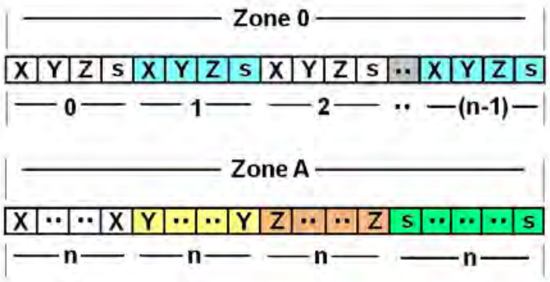
Zone 0
X Y Z s X Y Z s X Y Z s ·· X Y Z s
0
1
2
··
(n-1)
Zone A
X ·· ·· X Y ·· ·· Y Z ·· ·· Z s ·· ·· ·· s
n
n
n
n

Figure 4.10: SbDataAccessMap Multiple Block Details: comparison of two virtual multiple block arrays.

Figure 4.10 on page 79 shows a comparison of two DataAccessMap's interpreting the same data in two ways. The upper DataAccessMap labeled "Zone 0" reads X,Y,Z and a scalar value s in a row and repeats that n–times. The lower DataAccessMap labeled "Zone A" reads n–times value X followed by n–times value Y, value Z and the scalar value s. Note, that the data is not duplicated only the virtual arrays are defined differently.

4.4.2 Virtual Array Iterator using SoDataAccessNode

To simplify the use of SoDataAccessNode an iterator enclosures the SoDataAccessNode. The iterator calculates the real index in the array based on the index of the virtual array. The iterator provides the following methods for easy use.

- **void begin()**
  Resets the iterator to the first element in the virtual array.
- **void end()**
  Resets the iterator to the last accessible element in the virtual array.
- **bool isOngoing()**
  Returns TRUE as long as the end or the begin of the virtual array is not reached.
- **void copy(Iterator)**
  Duplicates all values of the passed iterator (see Table 4.1).
- **int getVirtualIndex()**
  Returns the current virtual index.
- **void setVirtualIndex(int virtualIndex)**
  Set the current virtual index and forces to recomputed the real index.
- **int idx()**
  Returns the current index in the real array based on the current virtual index.
- **++ operator**
  Increases the current virtual index.

Table 4.1 shows which combinations of virtual arrays can be handled by the **void copy (Iterator)**.

from	SINGLE	MULTI	RANDOM
to SINGLE	√	√	
to MULTI	√ *)	√	
to RANDOM			√

*Table 4.1: CashFlow Virtual Array Iterator: possible mapping of Iterator's. *) SINGLE to MULTI should be avoided because of low performance.*

4.4.3 SoDataAccessNode – File Format

SoDataAccessNode iv–file

```
#Inventor V2.0 ascii

SoDataAccessNode {
  fields [SoMFEnum    mode,
          SoMFEnum    type,
          SoMFEnum    arrayType,
          SoMFString  key,
          SoMFString  keyData,
          SoMFString  label,
          SoMFInt32   offset,
          SoMFInt32   length,
          SoMFInt32   repeat,
          SoMFInt32   inc,
          SoMFInt32   lookupTable ]

  mode        [ ADD ]
              # values (NONE, ADD, SET, CLEAR,
              #         CLEAR_ALL, LIST)
  type        [ T_FLOAT ]
              # values (T_NONE, T_STRING, T_INT32,
              #         T_FLOAT, T_COLOR, T_VEC2F,
              #         T_VEC3F, T_VEC4F, T_VEC9F)

  arrayType   [ SINGLE ]
              # values (SINGLE, MULTI, RANDOM)

  key         [ "my_unique_DataAccessKey" ]
              # unique key
  keyData     [ "my_MultiDataKey" ]
              # points to SoMultiDataNode
              # value: absolute key  (any string)
              #     or relative key @(integer)

  label       [ "my_label" ]  # optional
  offset      [ 0 ]
  length      [ 0 ]

  inc         [   ]   # for MultiBlock array
  repeat      [   ]   # for MultiBlock array
  lookupTable [   ]   # for RandomBlock array
}
```

CashFlow iv–file

4.5 Implementation of SoBaseGrid

One challenge in scientific visualization is, that there is a large amount of grids available and in use (see section 3.8 on page 59). If each algorithm has to be implemented on each grid or be adapted to the grid, the number of classes needed derived from the general algorithms would be huge. One other disadvantage of this approach gets obvious, if a new grid is introduced to the framework. In order to be able to render and process the grid, all existing algorithms have to be derived to be able to access this new grid. This approach was used in MeshVis [Mes] by TGS [TGS] and is discussed in detail in appendix A on page 120.

Our approach is to abstract the pieces of information from the grids needed by the algorithm. The algorithm communicates with the grid only via this interface. Once a new grid is introduced to the system, the grid only has to provide the topology data and overload the iterators. Also if a new algorithm is implemented it can access all grids via this abstraction layer. This concept is similar to the approach of the Field Model library [Mor03] and was also published at IEEE Vis2001 Tutorial 1 [Tut01].

The information required by the objects for accessing the grid include:

- **Geometric Primitives**

  There are four types of geometric primitives embedded in 3D.

  - **Vertex** zero dimensional
  - **Edge** one dimensional
  - **Face** two dimensional
  - **Cell** three dimensional

  This concept is similar to the one introduced in section 2.1.4 and figure 2.9 on page 2.9 published by [CUL89] as well as B-rep lists by [Sam90].

- **Topology Data**

  Connectivity information can be queried and is one way to provide all data needed by an algorithm.

  - **per Edge:** Information on the faces sharing this edge.
  - **per Face** Information on adjacent faces.neighborhood
  - **per Cell** Information on adjacent cells.

- **Grid Iterator**

  These iterators grant access to each cell or each face etcetera.

4.5.1 Geometric Primitives

In order to separate the visualization algorithms from the grid representation we have chosen to use *geometric primitives* in combination with *grid iterators* (see section 4.5.2). This approach was also introduced at IEEE Vis2001 tutorial 1 [Tut01] by Patrick Moran, who created the *Field Model library* [Mor03] implemented as a C++ template library using *partial template specialization*. Moran also published a paper on the field model library [Mor01].

Primitives describe basic geometric properties. This concept is also known as boundary representation (B-rep) [Sam90].

- **Vertex** [dim=0]
  A point in 3D.
- **Edge** [dim=1]
  An edge is a connection of two vertices.
- **Face** [dim=2]
  A face is created by at least three Edges and three vertices.
- **Cell** [dim=3]
  A cell is a closed volumetric object, that can be described as a set of faces.

4.5.2 Grid Iterators

A lot of algorithms traverse *geometric primitives* like the marching cubes algorithm[LC87]. Another example are cutting planes removing parts of a geometry. These algorithms can easily be implemented by traversing the grid and checking the distance to the cutting plane. One fundamental distinction when talking about grids is between *regular grids* and *non-regular grids*. Details will be provided in 2) and 3). To suite the needs of a large groups of algorithms we selected the following *grid iterators*.

1. **General Iterator**
   Iterators for Regular and Non-Regular Grids. Return all primitives of one kind, that is part of the grid. These iterators base on properties that all used grids have in common.
2. **Regular Grid Iterator**
   The most important property all regular grids have in common is, that the hole regular grids can be stored in one array. Information on adjacency is implicit. Only the dimensions of the grid must be known.
3. **Non–Regular Grid Iterator**
   This grid type provides no implicit adjacency information, but holds a separate list of adjacent *primitives*. Sometimes even this list has to be computed from the raw data. Based on that data structure and the properties of that grids useful iterators are chosen.

   Remark: There is a difference between non-regular grids and irregular grids. For instance the so called α-grids addressed by [Sad99] are regular grids with missing cells. Based on the definition of regular grids 2) α-girds are non-regular grids, but are not irregular grids.

4.5.3 Collection of SoBaseGrid Nodes

The following pages show a hierarchy of grid nodes including a brief description of their parameters.

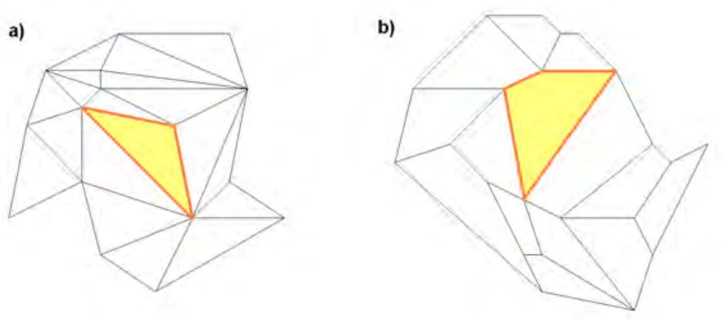

Figure 4.11: Polygonal Triangle grid (a) and polygonal Quad grid (b)

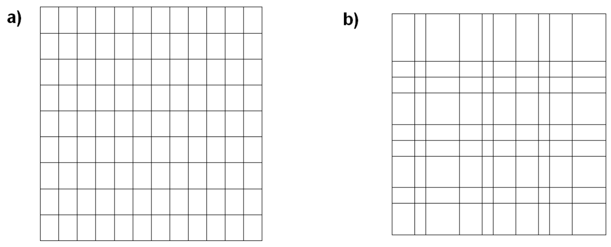

Figure 4.12: Rectangular 2D gird (a) and rectilinear 2D grid (b)

SoBaseGrid Nodes	
SoNode (Coin3D)	
SoBaseGrid	Abstract class for all grids.
SoStructuredGrid	Any grid, that can be stored in an n–dimensional array, where the index in the array implies its topological attributes.
SoStructuredGrid2D	A 2D grid, that can be stored in a two dimensional array, where the index in the array implies its topological attributes.
…	
SoStructuredGrid3D	see page 87
…	
SoUnstructuredGrid	see page 88
SoUnstructuredGrid2D	
…	
SoUnstructuredGrid3D	

SoStructuredGrid2D Nodes	
SoStructuredGrid2D	SoStructuredGrid → SoBaseGrid → *SoNode* → . . .
SoCurvilinearGrid2D	Each face contains of 4 vertices. The edges can be aligned in any angle.
SoRectangularGrid2D	Each face contains of 4 vertices. Each adjacent edge is orthogonal to each other.
SoRegularSpacedRectangularGrid2D	The spacing is constant for each principal direction. This is also known as Cartesian grid [Wal95] (see figure 4.12a) on page 84).
SoIrregularSpacedRectangularGrid2D	Same as a Cartesian grid, but the spacing may differ for all faces per row and per column. (see figure 4.12b) on page 84).
SoRadialGrid2D	
SoCylindricGrid2D	This grid is the 2D shell of a cylinder in 3D (see figure 4.17 on page 90). Parameters are the radius and the height of the cylinder.
SoRegularSpacedCylindricalGrid2D	The spacing of the grid is constant like the SoRegularSpacedRectangularGrid2D (see figure 4.13 on page 89).
SoIrregularSpacedCylindricalGrid2D	The spacing of the grid is variable like the SoIrregularSpacedRectangularGrid2D
SoPolarGrid2D	Top surface of a cylinder in 3D. Identical to the polar coordinate system in 2D. The inner radius may be zero.
SoRegularSpacedPolarGrid2D	The radius increment is constant as well as the angular increment. (see figure 4.14a) on page 89)
SoIrregularSpacedPolarGrid2D	The radius increment is variable and the angular increment is also variable. (see figure 4.14b) on page 89)
SoSphericalGrid2D	Surface of a sphere with constant radius. Two variables are mapped to angular azimuth and angular altitude (see one layer in figure 4.18 on page 90).
SoRegularSpacedSphericalGrid2D	The angular azimuth increment and the angular altitude increment are constant.
SoIrregularSpacedSphericalGrid2D	The angular azimuth increment and the angular altitude increment are variable.

SoStructuredGrid3D Nodes	
SoStructuredGrid (continued)	SoBaseGrid → *SoNode* → . . .
SoStructuredGrid3D	A 3D grid, that can be stored in a three dimensional array, where the index in the array implies its topological attributes.
SoCurvilinearGrid3D	One cell consists of 8 vertices. One face consists of 4 vertices. The faces must not be planar. In a regular case no cell intersects the another cell (see figure 5.18 on page 114). This is one of the most important grids for CFD data sets.
SoRectangularGrid3D	One cell consists of 8 vertices. One face consists of 4 vertices. Each face is orthogonal to its aligned faces. This grid is mostly used for medical data sets like CT and MR data sets. This grid is a specialization of the SoVoxelGrid.
SoRegularSpacedRectangularGrid3D	All cells have the same size in the three principle directions (see figure 4.16 page 90).
SoIrregularSpacedRectangularGrid3D	The cells may have different size in the three principle directions.
SoCylindricGrid3D	This grid defines the well known cylinder coordinates in 3D. Figure 4.17 on page 90 shows a visualization of a 3D cylindrical grid.
SoRegularSpacedCylindricalGrid3D	All cells have the same angular increment and the same height. The radius increment is constant too.
SoIrregularSpacedCylindricalGrid3D	The angular increment may differ as well as the height increment and the radius.
SoSphericalGrid3D	This grid is used for spherical coordinates in 3D (see figure 4.18 on page 90).
SoRegularSpacedSphericalGrid3D	The angular azimuth increment, the angular altitude increment and the radius increment are constant.
SoIrregularSpacedSphericalGrid3D	The angular azimuth increment, the angular altitude increment and the radius increment are variable.

SoUnstructuredGrid Nodes	
SoBaseGrid	*SoNode* → . . .
SoUnstructuredGrid	Any grid that can not be stored in an n–dimensional array, where the index in the array defines its topology is a SoUnstructuredGrid. Vertices are stored in one list. The geometric primitives such as triangle, quads etc. are defined as index lists representing the topology of the grid.
SoUnstructuredGrid2D	2D grids, that can not be stored in a two dimensional array, where the index in the array defines its topology.
SoTriangleGrid2D	A grid consisting of many connected triangles.
SoQuadGrid2D	A grid consisting of many connected quads. A quad is a closed polygon with 4 vertices.
SoUnstructuredGrid3D	Any grid, that can not be stored in a three-dimensional array, where the index of the array implies the topology information is a unstructured grid. These grids often use a list of vertices and a list of faces build by the vertices (compare to SoIndexedFaceSet in Coin3D[Coi00]).
SoTetrahedralGrid3D	Grid consists of closed cells with 4 vertices per cell and 4 faces (see figure 4.15 on page 89).
SoVoxelGrid3D	Grid containing of closed cells with 8 vertices per cell and 6 faces. The faces of a cell must not be orthogonal to each other (see figure 4.16 on page 90).
SoHybridGrid3D	Grid containing of closed cells with variable number of vertices per cell and face. This is a combination of different grids.

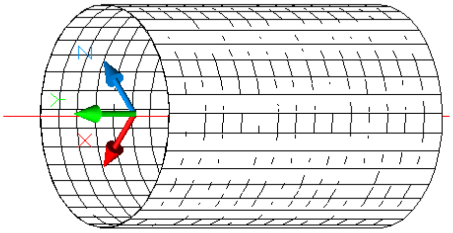

Figure 4.13: Cylindric 2D wire–frame grid embedded in 3D.

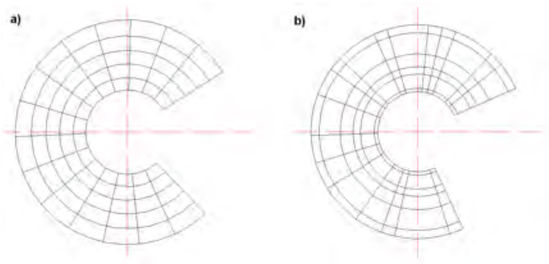

Figure 4.14: Regular polar grid (a) & non–regular polar grid (b) in 2D.

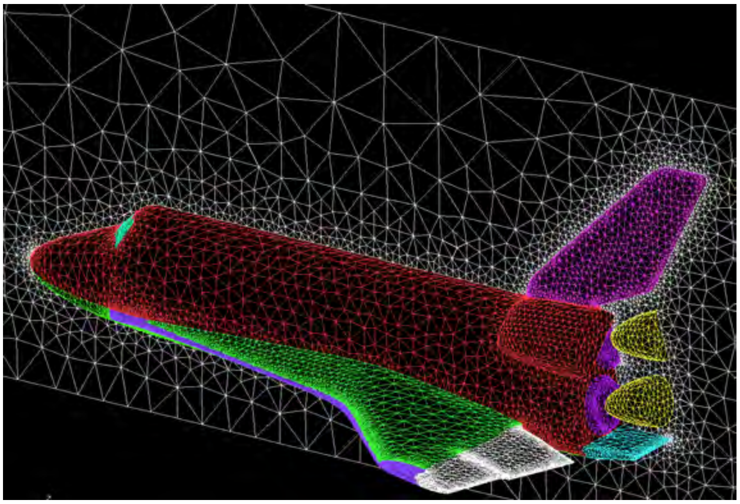

Figure 4.15: Visualization of tetrahedral grid. Surface of space shuttle and a cutting plane are shown.

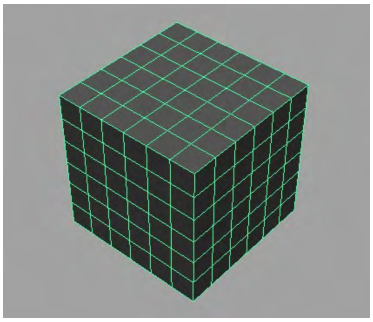

Figure 4.16: Voxel grid in 3D

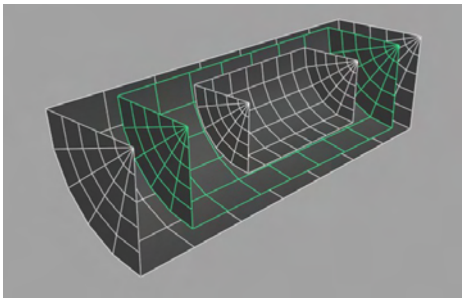

Figure 4.17: Cross section of cylindric grid in 3D

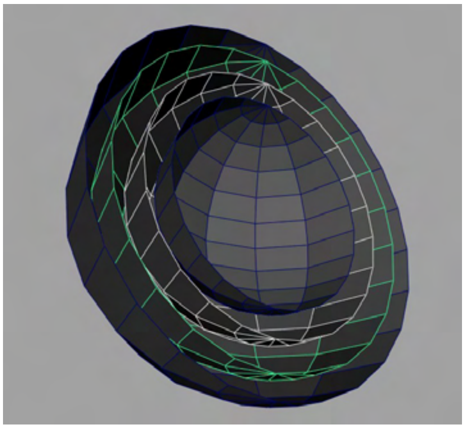

Figure 4.18: Cross section of spherical grid in 3D

4.5.4 SoStructuredGrid2D – File Format

SoStructuredGrid2D iv–file

```
#Inventor V2.0 ascii

SoStructuredGrid {
  fields [SoSFEnum    mode,
          SoSFString  label,
          SoSFString  grid_Key,
          SoSFString  dataAccess_Key,
          SoSFString  gridDim_Key ]

  label          "myGrid"   #optional
  mode           ADD
                 #values (NONE, ADD, SET, CLEAR, LIST)

  grid_Key       "myGrid_from_Loader"
                 #unique key for this grid

  dataAccess_Key "myVirtualArray_from_Loader"
                 # points to a DataAccessNode
                 #   with raw-data

  gridDim_Key    "myVirtualArray_from_Loader"
                 # points to DataAccessNode
                 #   with dimension of the grid
}
```

CashFlow iv–file

4.6 Implementation of Data Consumers

The *Data Consumer node* is an abstract class unify all nodes reading or writing data in the *CashFlow* framework. The remaining part of this chapter resumes the three derived classes of *Data Consumer node* and closes with a class hierarchy of *CashFlow* nodes on page 97.

4.6.1 SoLoaderNode

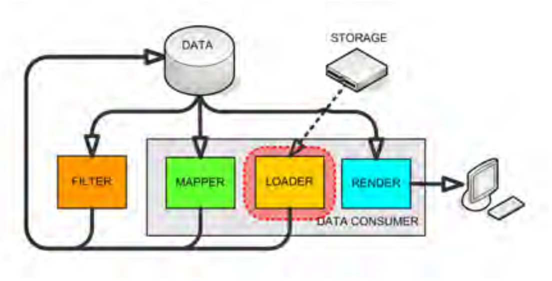

Figure 4.19: CashFlow: Data flow model - Loader node

One minor part is to be able to load several binary data files. The SoLoaderNode is the base class for binary loaders. It provides useful methods like conversions from little endian to big endian and vice versa as well as methods for checking the size of a file. This is important, because a lot of binary data files can only be loaded correctly if the header information in the file is crosschecked with the file size. Based on this pieces of information a assumption is made whether the binary files byte order is in big endian[8] in little endian[9] byte order.

[8]big endian byte order: the lowest address in memory is stored in high-order byte, the highest address in memory is stored in the low-order byte (MAC,SGI).

[9]little endian byte order: the lowest address in memory is stored in low-order byte, the highest address in memory is stored in the high-order byte (intel,PC).

SoLoader – File Format

This is an example for a binary loader for CFD – data sets. This loader was used for figure 5.12 on page 109 and figure 3.8 on page 48. The behavior of this node was described in section 4.1.1 on page 69. The fields

grid_AccessKey, *dataAccess_AccessKey*, *multiData_AccessKey*
point to a SoBaseGrid node, a SoDataAccessNode and a SoMultiDataNode.

SoLoaderCFD_Plot3DNode iv–file

```
#Inventor V2.0 ascii

SoLoaderCFD_Plot3DNode {
  fields [SoSFBool     enableSwapByteOrder,
          SoSFBool     readFile_Grid,
          SoSFString   fileName_Grid,
          SoSFString   grid_Key,
          SoSFString   dataAccess_Key,
          SoSFString   multiData_Key ]

  fileName_Grid        "myCFD_dataset.grid"

  readFile_Grid        TRUE
  enableSwapByteOrder  TRUE

  grid_Key             "myGrid_from_Loader"
  dataAccess_Key       "myVirtualArray_from_Loader"
  multiData_Key        "myData_fromLoader"
}
```

CashFlow iv–file

4.6.2 SoMapperNode

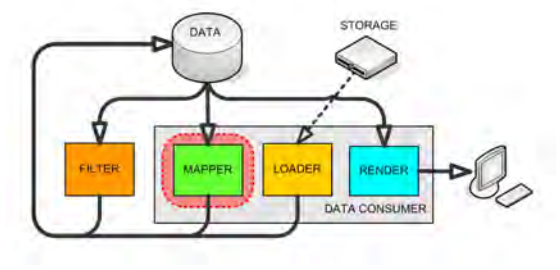

Figure 4.20: DataAccessMap: Accessing virtual array using the attributes offset, inc, length, label and data type (not shown)

SoMapperNode is one of the core parts, containing the visualization algorithms. Both of them can use several input streams and generate from one to many output streams This is also known as fan–in and fan–out.

The process flow between mapper and renderer gets more obvious in figure 3.15 on page 64.

The basic data access introduced in the UML class hierarchy (figure 4.5 on page 71) only shows an indirect way to access data from a DataComsumer. The UML sequence diagram in the next section 4.2.1 will provide more detailed information on that.

the most important representatives are:

- Texture Generation
- Streamline and Particle Tracer
- iso–surface extraction

SoMapperNode - File Format

This sample file generates a parameterization similar to the one shown in figure 5.9 and figure 5.10 on page 108. The field *lookup_kind* defines one of the three kinds of mappings:

0 .... constant mapping

All values in the interval are mapped to one constant output value. In our example the input interval[10] is $[7..9]$ and the constant output value is 4.

1 .... linear mapping

All values in the interval are mapped linear to the output value. In our example the input interval is $[3..7)$. Since the output interval $[6..4)$ decreases while the input interval increases an inverse linear mapping is achieve.

[10] Definition of intervals: brackets [ ] define a closed interval while ( ) define an opened interval.

2 .... cubic mapping

All values in the interval are mapped cubical to the output value using a Hermit cubic interpolation. The slope at the begin and the slope at the end of the interval is defined in the field *lookup_value_in_spline*, which is ignored if no cubic mapping is used. The values in field *lookup_value_in_spline* accord to $arctan(\alpha)$ with α as the angle of the tangent.

In our example the input interval for the cubic mapping is $[0..3)$, the output interval is $[1..6)$ and the slope are 0 at the first bound and -1 at the second bound. The slope value 0 results in a horizontal tangent, while the slope value of -1 leads to an tangent declined with is -45.

SoCubicDataMapperNode iv–file

```
#Inventor V2.0 ascii

SoCubicDataMapperNode {
  fields [SoSFString  scalarIn_Key,
          SoSFString  scalarOut_Key,
          SoMFInt32   lookup_value_in,
          SoMFInt32   lookup_value_out,
          SoMFInt32   lookup_kind,
          SoMFInt32   lookup_value_in_spline ]

  scalarIn_Key       "myVirtualArray_from_Loader"
  scalarOut_Key      "myVirtualArray_after_Mapping"

  lookup_value_in        [ 0 3 7 9 ]
  lookup_value_out       [ 1 6 4 4 ]
  lookup_kind            [  2 1 0  ]
       # 0... constant   1...linear  2...cubic

  lookup_value_in_spline [ 0 -1 ]
      # only used if lookup_kind is "2"
}
```

CashFlow iv–file

4.6.3 SoRenderNode

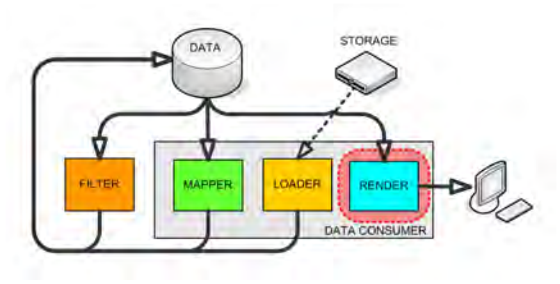

Figure 4.21: CashFlow data flow model: Render node

This node can handel several input streams. The major difference in comparison to the SoCfMapperNode and SoCfLoaderNode is, that it creates only OpenGL calls.

SoRenderNode - File Format

This SoPointRenderNode only needs a unique key pointing to a SoDataAccessNode, since the topology of a point cloud is not relevant in this example. In figure 3.8 on page 48 several SoPointRenderNodes are used in combination with various SoDataAccessNodes.

SoPointRenderNode iv–file

```
#Inventor V2.0 ascii

SoPointRenderNode {
  fields [SoSFString  dataAccess_Key ]

  dataAccess_Key   [ "my_unique_DataAccessKey" ]
    # points to SoDataAccessNode
    #
    # value: absolute key ( any string )
    #      or relative key @( integer )
}
```

CashFlow iv–file

4.7 CashFlow Class Hierarchy

This is a collection of the most important classes of CashFlow. Some classes use **SoCf *** as prefix, which is an abbreviation for *Scene graph Object Cashflow **.

Indention shows derived classes.

SoElement & SoAction

- *SoElement* (Coin3D)
  - SoCfMultiDataElement
  - SoCfDataAccessElement
  - SoCfGridElement
  - SoCfUpdataElement

- *SoAction* (Coin3D)
  - SoCfUpdateAction

Nodes used for data storage & data access

- *SoNode* (Coin3D)
  - SoMultiDataNode
  - SoDataAccessNode
  - SoBaseGrid (see page 98)

Virtual Array Iterator

- SbDataAccessIterator
  - SbDataAccessMultipleBlockIterator
  - SbDataAccessRandomBlockIterator
  - SbDataAccessSingleBlockIterator

- SbDataAccessMap

4.7.1 SoBaseGrid Nodes

```
SoNode (Coin3D)
  SoBaseGrid
      SoStructuredGrid
          SoGrid1D
              SoSpacedEvenGrid1D
              SoSpacedNonevenGrid1D
          SoStructuredGrid2D
              SoCurvilinearGrid2D
              SoRectangularGrid2D
                  SoRegularSpacedRectangularGrid2D
                  SoIrregularSpacedRectangularGrid2D
              SoRadialGrid2D
                  SoCylindricGrid2D
                      SoRegularSpacedCylindricalGrid2D
                      SoIrregularSpacedCylindricalGrid2D
                  SoPolarGrid2D
                      SoRegularSpacedPolarGrid2D
                      SoIrregularSpacedPolarGrid2D
                  SoSphericalGrid2D
                      SoRegularSpacedSphericalGrid2D
                      SoIrregularSpacedSphericalGrid2D
          SoStructuredGrid3D
              SoCurvilinearGrid3D
              SoRectangularGrid3D
                  SoRegularSpacedRectangularGrid3D
                  SoIrregularSpacedRectangularGrid3D
              SoCylindricGrid3D
                  SoRegularSpacedCylindricalGrid3D
                  SoIrregularSpacedCylindricalGrid3D
              SoSphericalGrid3D
                  SoRegularSpacedSphericalGrid3D
                  SoIrregularSpacedSphericalGrid3D
      SoUnstructuredGrid
          SoUnstructuredGrid2D
              SoTriangleGrid2D
              SoQuadGrid2D
          SoUnstructuredGrid3D
              SoTetrahedralGrid3D
              SoVoxelGrid3D
              SoHybridGrid3D
```

Indention shows derived classes.

4.7.2 SoCfDataConsumer Nodes

- *SoNode* (Coin3D)
  - SoCfDataConsumer
    - SoCfMapperNode
    - SoCfRenderNode
    - SoCfLoaderNode

4.7.3 SoCfMapperNode

- SoCfDataConsumer
  - SoCfMapperNode
    - SoColorMapperNode
      - SoColorScalarMapperNode
      - SoColorVectorMapperNode
    - SoDataMapperNode
      - SoCubicDataMapperNode
    - SoGeometryMapperNode
      - SoCarpetPlotMapperNode
        - SoCarpetPlotScalarMapperNode
        - SoCarpetPlotVectorMapperNode
      - SoCuttingMapperNode
        - SoCuttingPlaneMapperNode
        - SoCuttingSurfaceMapperNode
      - SoIsoValueMapperNode
        - SoIsoLineMapperNode
        - SoIsoSurfaceMapperNode
      - SoStreamlineMapperNode
        - SoStreamlineSimpleMapperNode
        - SoStreamlineVectorMapperNode
    - SoStreamlineGeneratorNode

Indention shows derived classes.

4.7.4 SoCfRenderNode

- SoCfDataConsumer
  - SoCfRenderNode
    - SoCellRenderNode
    - SoGlyphRenderNode
      - SoGlyphLineRenderNode
      - SoGlyphNormalizedRenderNode
    - SoOctreeRenderNode
    - SoPointRenderNode
      - SoAdvancedPointRenderNode
    - SoPolygonRenderNode
    - SoStreamlineRenderNode
      - SoIlluminatedStreamlineRenderNode
      - SoStreamBoxRenderNode
      - SoStreamGlyphRenderNode
      - SoStreamlineSimpleRenderNode
      - SoStreamRibbonRenderNode
      - SoStreamTubeRenderNode
    - SoStructuredGridRenderNode
    - SoWireframeRenderNode

4.7.5 SoCfLoaderNode

- SoCfDataConsumer
  - SoCfLoaderNode
    - SoLoaderCFD_Plot3DNode
    - SoLoaderGridDatNode
    - SoLoaderCTNode

Indention shows derived classes.

Chapter 5

Results

In the previous chapters the underlying concepts of *CashFlow* were introduced.

Our experience shows that the full power of the scene graph in combination with scientific visualization can only be unleashed using our proposed triumvirate of Multi-Data, DataAccess and Grid node.

5.1 Combining MultiData Node & DataAccess Node

To show the full potential of the MultiData node and the DataAccess node we want to compare four possibilities to combine these nodes. The visual result remains the same for all four possibilities and is shown in figure 5.1. The three images in figure 5.1 show the same set of periodic random data mapped to three different coordinate systems. In the left image the data is mapped to a spherical coordinate system, the middle image shows a 2D polar coordinate system and a Cartesian coordinate system is used in the right image. In this simple visualization each grid is stored separately, generated from the same initial data using the same color map. The color map is also generated from the initial data.

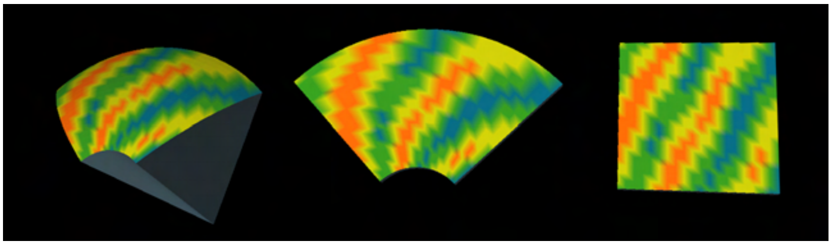

Figure 5.1: CashFlow basic example: A planar surface rendered using spherical (left), polar (middle) and Cartesian coordinate system (right). The color map is generated from periodic random data (20x20 values) and is applied to all three grids. Figures 5.3 – 5.6 show possible simplified scene graphs generating this visualization. The complete corresponding scene graph to this image is show in figure 5.7 (page 106) and figure 5.8 shows the corresponding process flow diagram.

The four combinations of MultiData nodes and DataAccess nodes shown in figure 5.3 – 5.6 on the previous page and in figure 5.2 on page 104 are:

1. **Smallest scene graph:** (*see figure 5.3 on page 105)*
   The first of four scene graph consists of one MultiData node (**D**)[1], one DataAccess node (**A**)[2] and three Render nodes (**R**)[3,4,5]. The three Render node [3,4,5] visualize the spherical grid, the polar and Cartesian grid shown in figure 5.1 (page 101). In this simplified scene graph the traversal starts at the Separator node (**S**). The MultiData node [1] and the DataAccess node [2] register themselves to the elements. The Render nodes [3,4,5] request the reference to the data via the elements.

   It is assumed, that the MultiData node [1] and the DataAccess node [2] contain the data for the three grids. For details on how that data is inserted into the scene graph read the next section 5.2, please.

   The process flow diagram in figure 5.3 (right image) shows, how the Render nodes [3,4,5] query for the DataAccess map's inside the DataAccess node [2]. The DataAccess map contains detailed information on the virtual array (see section 3.4) and a link to the MultiData node [1].

This is a simple example for a basic scene graph and its process flow diagram. It shows that though the nodes are processed by the scene graph in order [1] to [5] the process flow looks different. Since the render nodes request for the data is passed to the DataAccess node first and afterwards to the MultiData node the process flow is

$$[3] \rightarrow [2] \rightarrow [1]$$

for the first Render node. This is a basic concept of *CashFlow*. A DataConsumer node requests data from the DataAccess node receiving a DataAccess map that links to a MultiData node. Due to the constraint, that no new nodes shall be inserted into the scene graph during runtime by *CashFlow* nodes, the process flow runs *reverse* to the scene graph traversal for the previous mentioned nodes.

Scene graph traversal:

[**MultiData**] → [**DataAccess**] → [**DataConsumer**]
or
[**DataAccess**] → [**MultiData**] → [**DataConsumer**]

Process flow:

[**DataConsumer**] → [**DataAccess**] → (**DataAccess map**) → [**MultiData**]

One important simplification used in this section and in figure 5.3 – 5.6 is, that **no** *Grid node* was used. The intention to proceed like this is to keep the focus on the main topic of that section, namely the abilities of MultiData nodes and DataAccess nodes. The complete scene graph corresponding to the visualization in figure 5.1 (page 101) is more complex, because it requires an additional Grid node and a Mapper node. The *Grid node* will be added to the scene graph and to the process flow in the next section (see section 5.2 on page 106).

2. **Several MultiData nodes:** (*see figure 5.4 on page 105)*
   Figure 5.4 shows the possibility, that several MultiData nodes [1][2][3] are activated from one DataAccess node [4] via the DataAccess map's inside the DataAccess node. Note, that this is a very important example, because the MultiData nodes can also be spread across the scene graph. The only constraint is, that the nodes must be traversed before the Render node is touched. The MultiData node and the DataAccess node register themselves to their corresponding elements during scene graph traversal. So obviously the Render node or any DataConsumer node has to be in traversal order after both corresponding MultiData and DataAccess node had registered themselves.

   The process flow diagram shows, that the Render nodes [5][6][7] all request data from the same DataAccess node [4]. Even so all of them use the same DataAccess node [4] the requested DataAccess map's link them to three different MultiData nodes [1][2][3].

   The importance of this ability get's more evident if you remember, that the traversal order of nodes may guide their linking (see section 3.3 on page 43 for details). This means that changing two MultiData nodes for example node [2] and node [3] will cause the Render node [6] to visualize the data from node [3] instead of node [2] and vice versa Render node [7] previously linked to node [3] would be linked to node [2] instead. To accomplish this we have to rely on indirect linking via Elements using *relative address keys*. A relative address key refers to the order of insertion of MultiData and DataAccess nodes while scene graph traversal. Using this concept we can link between the MultiData nodes [1][2][3] and the Rendering nodes [5][6][7] taking their position in the traversal order into account.

   This concept can also be applied to DataAccess nodes as shown in figure 5.5.

3. **Several DataAccess nodes:** (*see figure 5.5 on page 105)*
   This example is similar to the first one in figure 5.3, because all three Render nodes [5][6][7] request DataAccess map's(grey) pointing to one MultiData node [1]. The only difference is, that each Render node accesses its own DataAccess node.

   As already mentioned in the previous section the use of indirect linking creates new options.

4. **Largest scene graph:** (*see figure 5.6 on page 105)*
   The last example shows three independent process pipelines consisting of a Render node, a DataAccess node and a MultiData node.

   $$\begin{array}{ll} \textit{pipeline A:} & [7] \rightarrow [4] \rightarrow [1] \\ \textit{pipeline B:} & [8] \rightarrow [5] \rightarrow [2] \\ \textit{pipeline C:} & [9] \rightarrow [6] \rightarrow [3] \end{array}$$

   The scene graph consists of three MultiData nodes[1][2][3], three DataAccess nodes [4][5][6] and three Render nodes. Besides the earlier mentioned concepts and strategies it is important to keep in mind, that only one constraint for each Render node has to be satisfied with is:

> *In traversal order the DataConsumer node must be after the corresponding MultiData and DataAccess node.*

Following that rule we can change the position of the nodes [1]–[6] as long as they are in front of the Render nodes [7][8][9]. When using *absolute address keys* changing the position of nodes in the scene graph does not influence the result, since the Render nodes link to the DataAccess nodes directly. This direct linking of nodes is comparable to a *field connection* and is accomplished via Elements and *unique keys* used as *absolute address keys*.

Otherwise when using *relative address keys* the ordering of nodes [1]–[6] effects the linking of Render nodes and DataAccess nodes (for details on *indirect references* see section 3.3 on page 43).

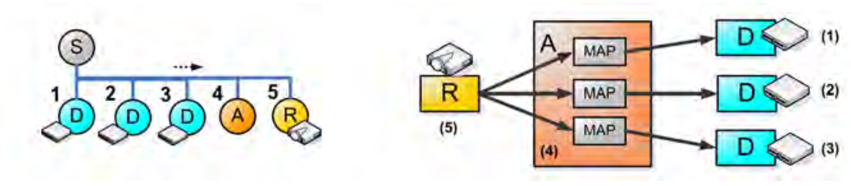

*Figure 5.2: CashFlow Scene Graph Example Render Node reads multiple Input (left): Three MultiData(***D***), one DataAccess(***A***) and a Render(***R***) node. Process flow (right): The Render node [5] requires multiple data input. It is linked via the DataAccess node [4] to the embedded DataAccess map(grey). Each DataAccess map(grey) refers to a separate MultiData nodes [1][2][3]. Note, that the MultiData nodes [1-3] and the DataAccess node [4] may be somewhere else in the scene graph.*

5. **Render node linked to several MultiData nodes:** (*see figure 5.2*)
   All other examples show one Render node linked to one MultiData node. This example shows a Render node requesting three MultiData nodes. An example for such a rendering algorithm is an StreamRibbon Renderer. The first MultiData node stores the interpolation points of the streamline, the second node stores the orientation per interpolation point and the third node provides a color–map for the Ribbon mapped to a scalar value.

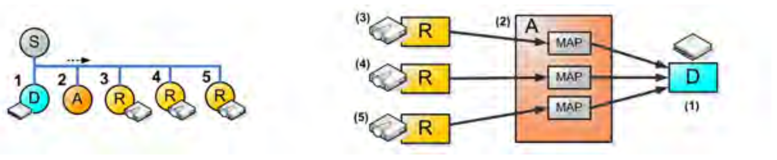

*Figure 5.3: Simple CashFlow Scene Graph Example (left): One MultiData(**D**), one DataAccess(**A**) and a Render(**R**) node. The root node is a Separator (**S**). Process flow (right): Each Render node [3][4][5] uses its own DataAccess map(grey) inside one DataAccess node [2] referring to one MultiData node [1]. Figure 5.1 shows a possible result of this scene graph were each Render node generates one of the three grids.*

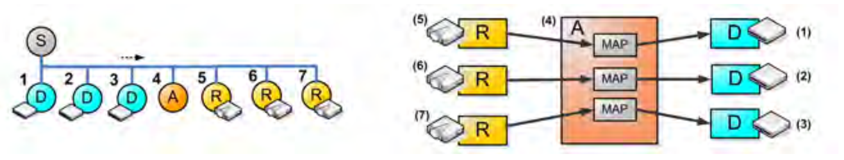

*Figure 5.4: CashFlow Scene Graph Example with several DataAccess nodes (left): Three MultiData(**D**), one DataAccess(**A**) and a Render(**R**) node. Process flow (right): Each Render node [5][6][7] uses its own DataAccess map(grey) inside one DataAccess node [4] referring to several MultiData nodes [1][2][3]. This scene graph may produce the same output as scene graph from figure 5.3.*

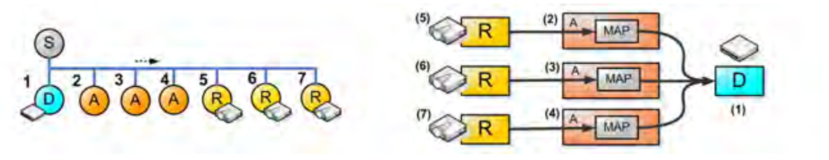

*Figure 5.5: CashFlow Scene Graph Example with several MultiData nodes (left): One MultiData(**D**), three DataAccess(**A**) and a Render(**R**) node. Process flow (right): Each Render node [5][6][7] links to a separate DataAccess node [2][3][4] and accesses the DataAccess map(grey) inside. All DataAccess map's (grey) refer to the same MultiData nodes [1]. Although this process flow differs from figure 5.3 and figure 5.4 the visual result may stay the same.*

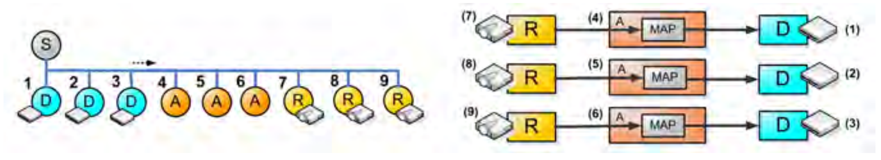

*Figure 5.6: CashFlow Scene Graph Example with several MultiData nodes (left): Three MultiData(**D**), three DataAccess(**A**) and a Render(**R**) node. Process flow (right): Each Render node [7][8][9] is bound to a separate DataAccess node [4][5][6] using the embedded DataAccess map(grey). Each DataAccess map(grey) refers to a separate MultiData nodes [1][2][3]. Although the four introduced scene graphs look quite different, they may produce the same visual results, because of the DataAccess node interface.*

5.2 CashFlow Scene Graph Examples

The complete *CashFlow* scene graph shown in figure 5.7 consists of (1) a loader importing data (2) a mapper creating the three girds and a renderer(3). This process flow was also introduced as the *CashFlow* Visualization Pipeline (see chapter 3.1 on page 38). Note, that figure 5.1 from page 101 can be produced be this scene graph.

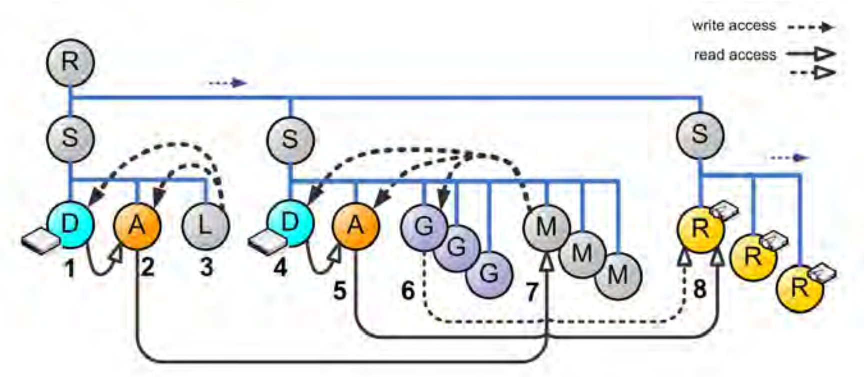

*Figure 5.7: Complete CashFlow Scene Graph: Consists of MultiData(**D**), DataAccess(**A**), Grid(**G**), Mapper(**M**) and Render(**R**) node as well as root(**R**) and separator(**S**) node.*
This scene graph corresponds to the visualization in figure 5.1 on page 101 and to the process flow diagram in figure 5.8.

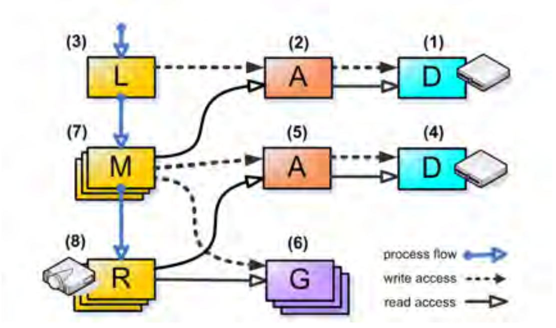

*Figure 5.8: CashFlow Scene Graph – Process Flow: Consists of MultiData(**D**), DataAccess(**A**), Grid(**G**), Mapper(**M**) and Render(**R**) node as well as Root(**R**) and Separator(**S**) node.*
This process flow diagram corresponds to the scene graph in figure 5.7 and to the visualization in figure 5.7 on page 106.

Let's take a closer look at the scene graph. The traversal starts at the root node (*R*) descending to the first MultiData node [1] passing the Separator node (*S*). Node [1] and the DataAccess node [2] register themselves to their corresponding elements (not shown in figure 5.7) and traversal proceeds to the Loader node [3]. This Loader node [3] imports data by adding a DataAccess map to the DataAccess node [2] and inserts the new data into MultiData node [1]. Again the process flow points into the opposite direction to the scene graph traversal as sketched in figure 5.8. The traversal of the scene graph is continued passing the second Separator node (**S**) and descending to MultiData node [4]. The nodes [4][5][6] are only empty space holders registering to their elements. Let's switch back to the process flow diagram in figure 5.8. The Mapper node [7] accesses the DataAccess node [2] and reads the data stored in node [1]. Using that data the Mapper node generates the a spherical grid by inserting a new DataAccess map to DataAccess node [5]. The generated DataAccess map points to node [4] were the data is stored. Also the type of grid is stored in the *Grid node* [6]. Since a *Grid node* can only store information for one grid we need three Grid nodes. The two other mapper nodes proceed the same way.

Finally the traversal reaches the Render node [8]. Each Render node first requests the grid data from DataAccess node [5] and receives a DataAccess map pointing to MultiData node [4]. In figure 5.7 this is symbolized by the arrow, that starts at node [4] passes node [5] and ends to node [8]. The Render node also needs the topology information stored in the Grid node [6]. Same goes for the other Render nodes that are linked to the other Grid nodes. The process flow diagram in figure 5.8 emphasizes this procedure also. The color map is generated on–the–fly by the Render nodes. This color map could also be created only once by another Mapper node, but this was avoided to keep figure 5.7 as simple as possible.

Applied Visualization Techniques

After resuming the base concepts of *CashFlow* some more evolved visualizations are discussed now in detail. We will extend the scene graph from figure 5.7 on page 106. First we will create the color map only once and link all render nodes to it. Second we will generate a height field from the raw data using another Mapper node and third we will apply the color map to the height field. Figure 5.9 shows the resulting image in top view while in figure 5.10 the scene is rendered in side view to emphasize the height fields.

The bottom row consists of three planes in spherical, polar and Cartesian coordinates with the color map generated from the sample values. The middle row shows the same coordinate systems without the color map but with a height field from the same data used from the color map. In the top row both height maps and color maps are combined. due to the used color map low values are mapped to green, medium values are mapped to yellow and high values are mapped to red.

In the last section we used a linear color mapping for regular grids in figure 5.10 and figure 5.9.

More examples with real data are presented in figure 5.11 on page 109. This is a 2D flow field passing a block moving from left to right. The used grid is a rectilinear grid, which is very dense around the block. In figure 5.12 color maps are applied to planes in 3D. The used data set is the space shuttle data set from NAS NASA [1]. Color is mapped to total energy.

[1]dataset from NASA at: http://www.nas.nasa.gov/Research/Datasets/data_sets.html [NAS05]

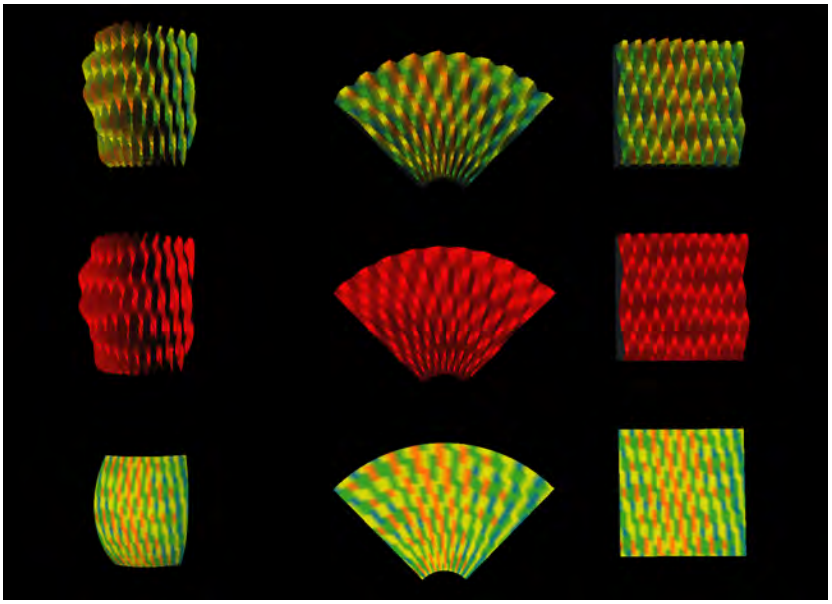

Figure 5.9: CashFlow Cartesian, polar and spherical grids I (top view)

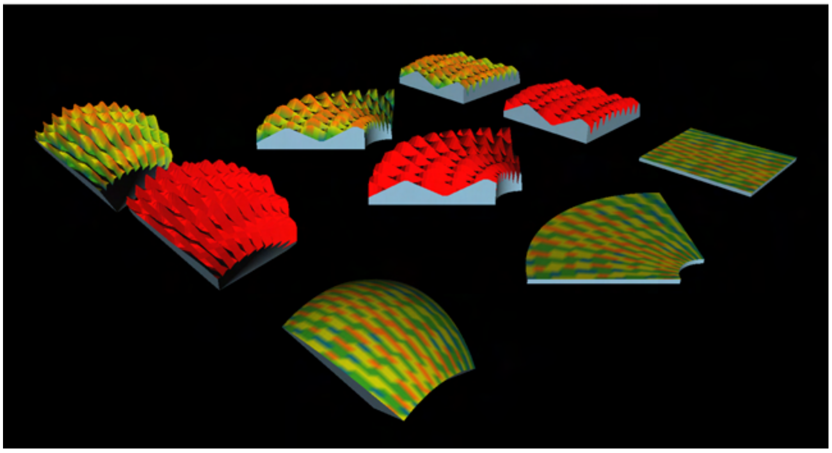

Figure 5.10: CashFlow Cartesian, polar and spherical grids II

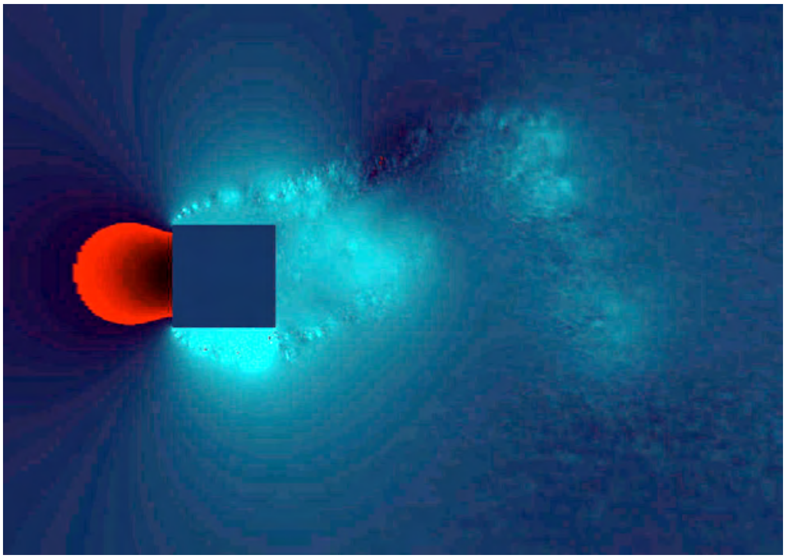

Figure 5.11: CashFlow Color map for 2D flow. Fluid flow from left to right. Color is mapped to pressure.

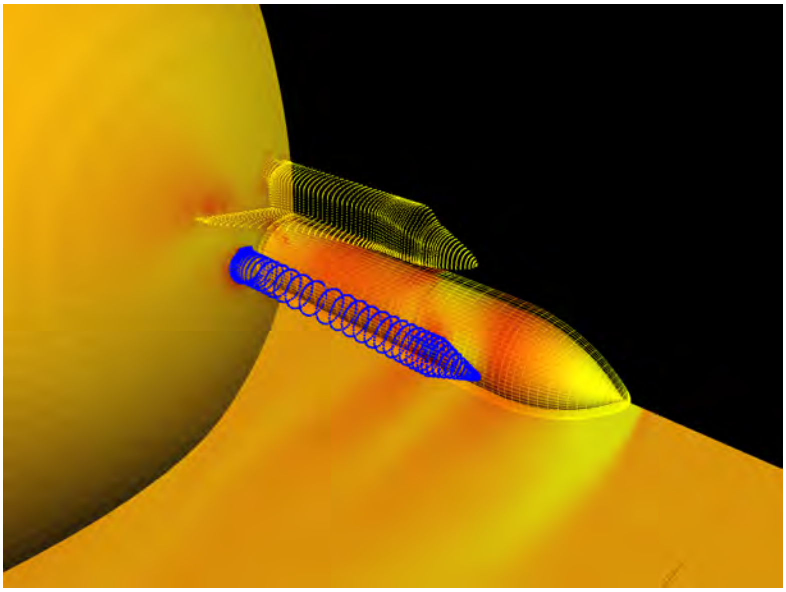

Figure 5.12: CashFlow Color Map in space shuttle data set. Color corresponds to total energy.

5.3 Curvilinear Grid Visualization

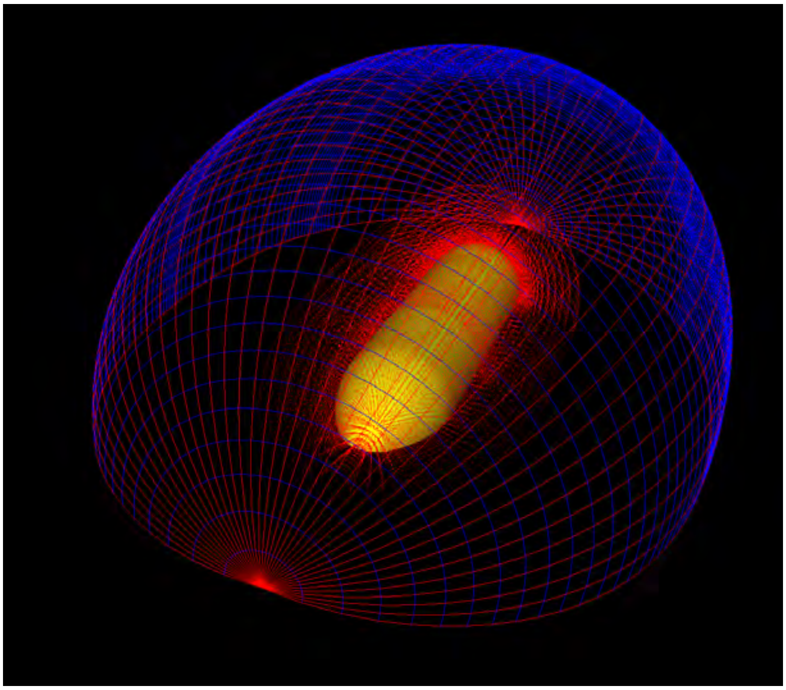

Figure 5.13: CashFlow Curvilinear Grid Visualization: Blue lines visualize the radial components of the cell while red lines show the axial components and their spacing. The most inner plane of the grid is shown in yellow.

A lot of CFD data sets are defined on curvilinear grids. Several algorithms have been published on fast rendering of curvilinear grids. Different to equally spaced Cartesian grids and rectilinear grids the shape of a curvilinear grid is not obvious. Thus visualizing the grid itself is an important topic as in figure 5.13 and figure 5.14.

Figure 5.13 shows a visualization of a hemisphere defined on a curvilinear grid. The most inner layer of this grid is rendered as yellow surface. This is the external tank of the space shuttle (see figure 5.16 on page 112). The high density of the inner layers is illustrated with a red point–cloud surrounding the external tank in yellow. Finally the most outer layer is visualized using radial concentric arches in blue and axial components of the grid in red.

Figure 5.14 focus on the space shuttle and its curvilinear grid. The red concentric lines define two planes perpendicular to the longitudinal axis.

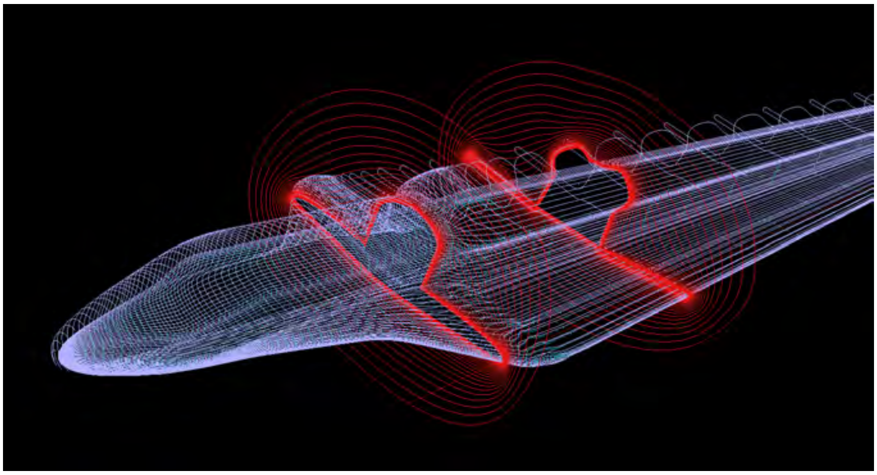

Figure 5.14: CashFlow Curvilinear Grid visualization II: Red lines show two concentric layers in the X–Y plane of the grid. Blue lines visualize radial and axial components in the X–Z plane of the grid.

5.4 Streamlines in CashFlow

Several rendering style for Streamlines are implemented in *CashFlow*. They can be divided into two groups:

- **Simple Streamlines**
  They are based on several interpolation points, which are connected to form the stream line. Generating simple streamlines the two interpolation points are connected using lines, tubes or illuminated lines

  - **Simple Streamlines**
  - **StreamTubes** *(see figure 5.15)*
  - **Illuminated Streamlines** *(see figure 5.16)*

- **Oriented Streamlines**
  In addition to the number of interpolation points creating the streamline each interpolation point defines a vector, indicating the vorticity of the flow. Using that additional information leads to two new basic rendering styles:

  - **StreamRibbons**
    Two interpolation points are connected by a rectangle. The additional vector defined at each interpolation point is used to create that rectangle.
  - **StreamBoxes**
    The additional vector defines the orientation for a box, that is extruded along the streamline. This is an extension to the StreamTubes style allowing perception of vorticity of the flow along the streamline.

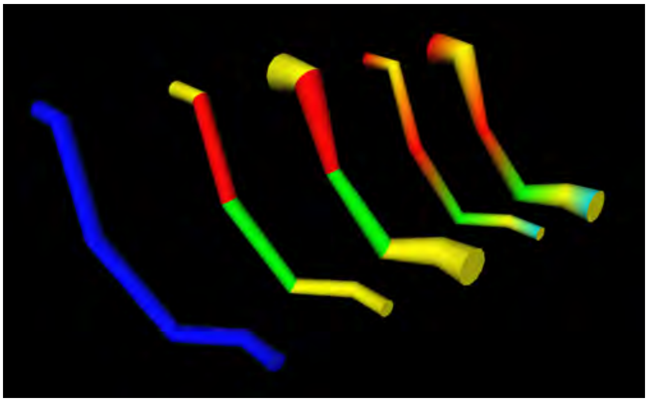

Figure 5.15: CashFlow StreamTube rendering styles: (left) constant tube diameter using one color. (middle left) constant tube diameter, scalar value mapped to color. (middle) same color map but tube diameter correspond to scalar value. (middle right) smooth color blending and constant tube diameter. (right) smooth color blending and varying tube diameter.

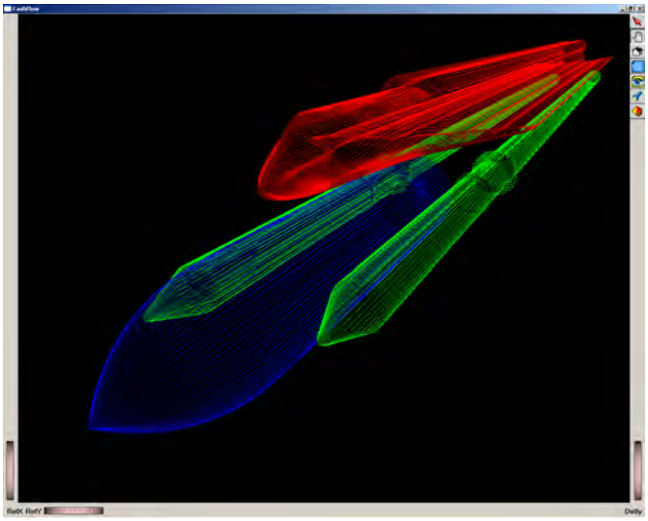

Figure 5.16: CashFlow Illuminated Streamlines. Space shuttle data set containing solid rocket booster in green, external tank in blue and space shuttle in red. Visualization of three different curvilinear grids.

5.5 Grid Iterator Examples

As proposed in section 4.5.2 on page 83 *Grid Iterators* can decouple the visualization algorithm from the grid representation. The advantage of that approach is, that the algorithm can be applied to several grids without modifications. Also a new type of grid can be accessed by several algorithms by implementing the grid interface for the new grid only. The disadvantage of that additional software layer is, that the execution speed of the algorithms is decreased compared to algorithms using raw access to the grids, taking properties of the grid into account.

Two examples, figure 5.17 and figure 5.18 shows the use of grid iterators. The first image in figure 5.17 show the space shuttle data set rendered two times. The rear yellow part was created using two cutting planes and the from green part shows the inverse selection visualizing the rest of the plane. Note, that the data set is defined on a curvilinear grid. The visualization was created by iterating over the grid.

The second example in figure 5.18 shows iso–volumes below a certain iso–value rendered in red using the cuberille rendering style [LC85]. The image in the middle of figure 5.18 shows the inverted selection from the above image in red. The lowest image illustrates the inverted selection again using a wire–frame visualization combined with a point–cloud rendering of all inner cells.

As future work we plan to port the marching cubes algorithm by [LC87] to *Cash-Flow* creating a *marching cubes grid iterator*. Figure 5.19 shows a human skull generated with marching cubes rendered as IndexedFaceSet using an Open Source implementation.

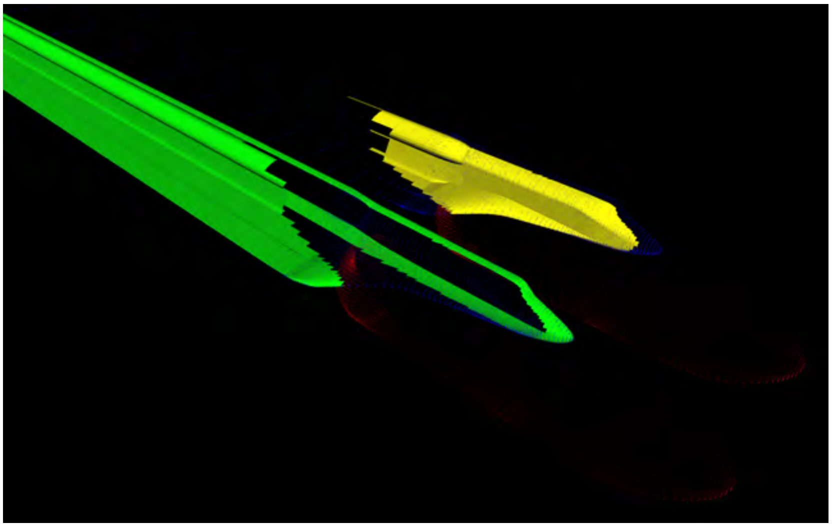

Figure 5.17: CashFlow Cutting Planes

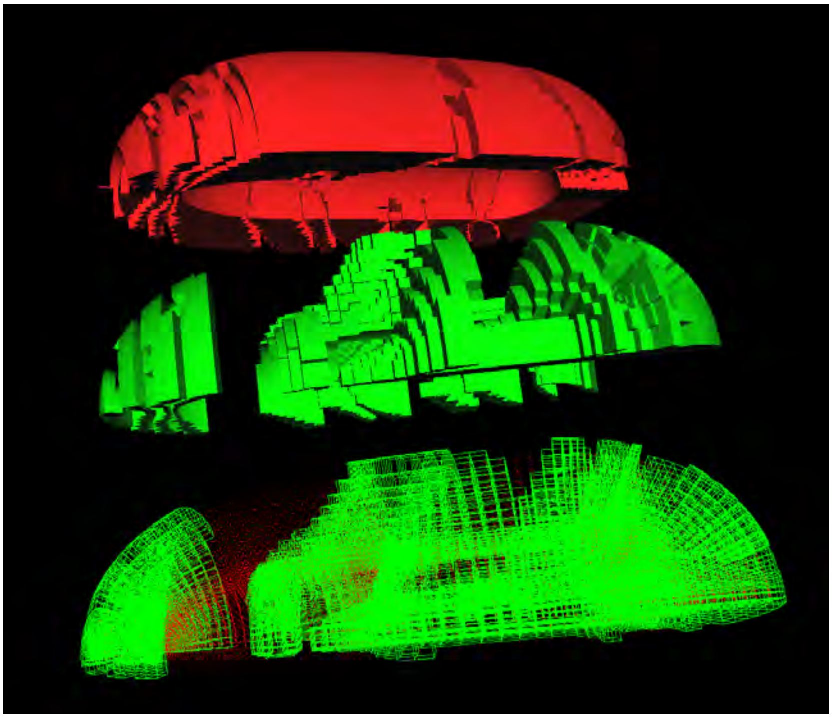

Figure 5.18: CashFlow Marching cube rendered with cuberille [LC85].

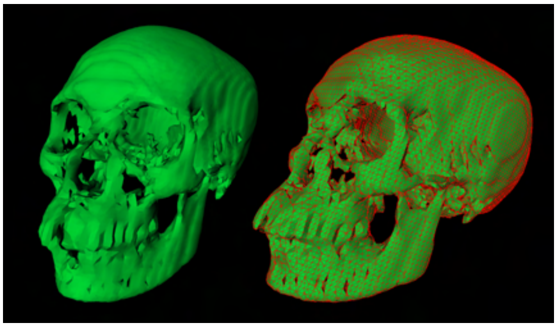

Figure 5.19: CashFlow Marching Cubes: A open source marching cubes implementation was used to generate this skull rendered as IndexedFaceSet.

5.6 Example for Parameterization

When dealing with data it is essential to be able to compare data sets. Independent of the origin of the data frequently the need arises to scale the data or shift the neutral axis or in general re–parameterize the data.

CashFlow supports three basic types of interpolation namely (1)constant, (2)linear and (3) cubic interpolation [AH02a]. Several examples are combined in figure 5.20. Note, that the elevation is used to visualize the mapping. One color map generated from the raw data is applied to all other visualizations.

(raw) The raw data is used as input for all other renderings (a)–(e). Based on the raw data a color map was created mapping blue and green to low regions and yellow and red to high regions.

(a) shows the color map generated from the raw data.

(b) Inverse mapping. Low input values are mapped to high output values and vice versa. Height field (b) is and inversion of height field (c).

(c) Combination of cubical and constant parameterization. Cubical mapping is applied to low values shown in blue and green while all high values in yellow and red are mapped to a constant value creating a plateau. Due to the cubic interpolation the slope in the blue and green regions differ from the original data labeled [raw] in figure 5.20.

(d)(e) Cubic mapping with different parameters. In (d) the peak is exaggerated and the slope at that point is quite steep compared to the original data [raw]. In (e) the incline at the peak is horizontal and smooth compared to the original data.

While in figure 5.20 the same color map was used for all visualizations contrariwise figure 5.21 shows color map's created of each elevation set. To keep consistency the elevation set are labeled corresponding to figure 5.20. Comparing the color distribution of the elevation sets in figure 5.21 gives a good clue on how the original values labeled as [raw] are redistributed. It also eases the comparison of heights, because equal heights have equal colors in figure 5.21. Especially the distribution of blue in low regions and red in high regions emphasize the results from the different mappings. In order to focus on the color distribution the surrounding boxes were disabled. Figure 5.21(f) shows an additional mapping with a steep slope for low values and a horizontal angle at the peak for high values.

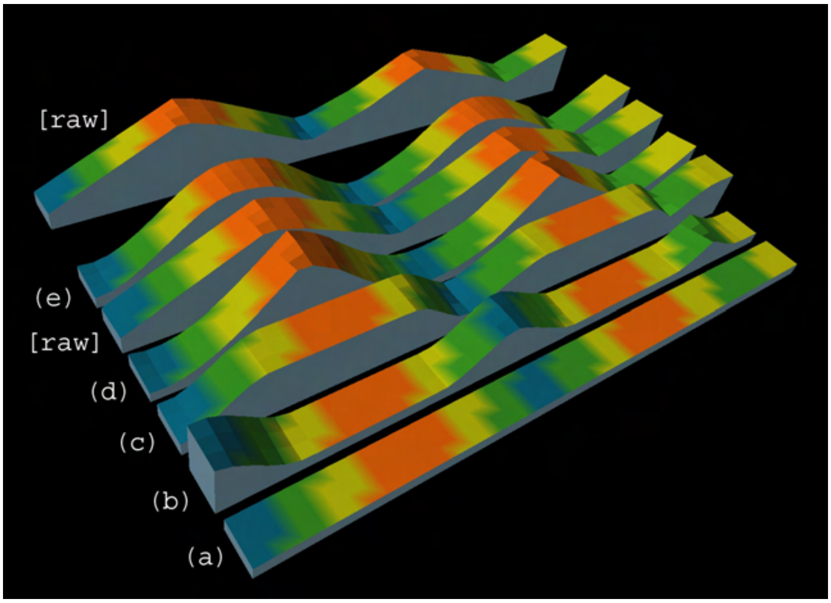

Figure 5.20: CashFlow Constant, linear and cubic parameterization: A color map created from the raw data was applied to all surfaces.

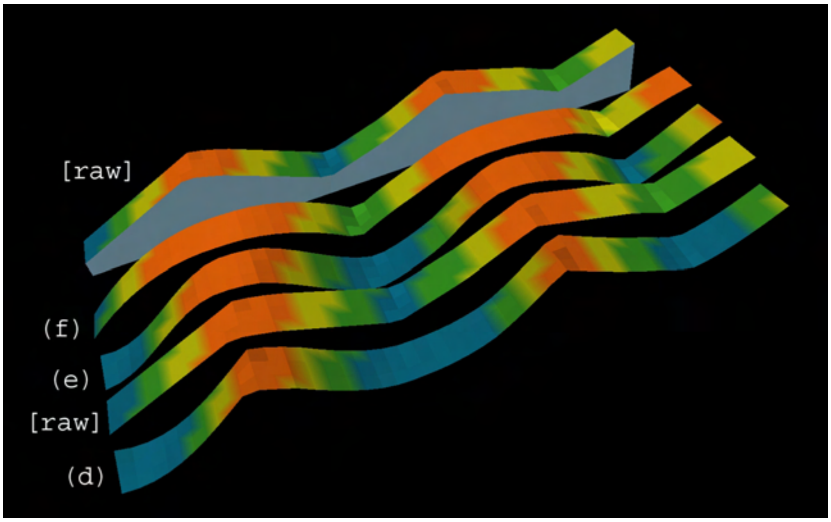

Figure 5.21: CashFlow Constant, linear and cubic parameterization: A color map was created for each height field. Letters indicate corresponding data sets in this figure and figure 5.20. The color map in this figure is bound to the actual height. This image also shows that the surrounding grey box can be disabled.

Figure 5.22 shows an example of a 2d histogram sharing the data with the height field on the right. Histograms are able to analyze scalar data from DataAccess node's.

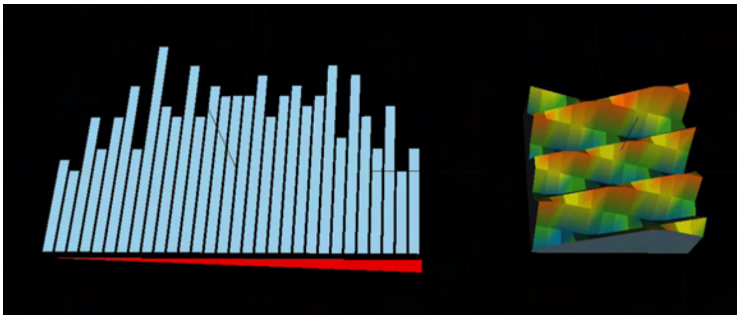

Figure 5.22: CashFlow 2D Histogram

5.7 Polygonal Surfaces & Textured Surfaces

Figure 5.23 shows polygonal rendering of curvilinear grids. The left image contains the rocket booster in yellow with an overlayed point rendering of grid points in the next layer. Right behind it the external Tank ins rendered in blue using flat–shading. The green band visualizes the deformation of the curvilinear grid in axial direction enclosing the space shuttle. The space shuttle is rendered twice, on one hand using Gouraud–shading[Gou71] in combination with a point rendering of the other half of the space shuttle. Unfortunately OpenGL still does not support Phong–shading[Pho73][BW86] although the necessary vertex normals are available. This limitation of OpenGL is bypassed using shaders in general.

On the other hand using flat–shading on top of the first rendered shuttle. The right image in figure 5.23 shows the same scene from another viewing angle with different rendering parameters. From top to bottom the first shuttle (top) is textured with a dominant green PLIC texture. The middle rendering shows Gouraud–shading and a grid visualization for the right side of the shuttle. The grid visualization consists of axial lines rendered in blue and radial lines rendered in yellow. The illustration at the bottom is similar to the left image, except for the shading style used for the space shuttle in yellow. In the right image the space shuttle at the bottom is flat–shaded while in the left image the lower space shuttle is rendered with Gouraud–shading. Figure 5.24 and figure 5.25 show curvilinear grid visualizations enhancing the shape of the grids.

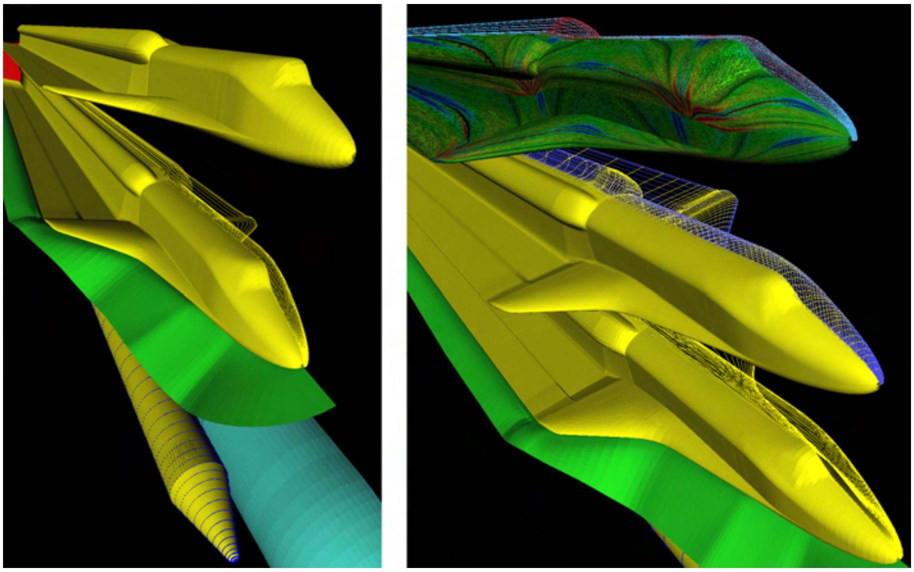

Figure 5.23: CashFlow flat–shaded & Gouraud–shaded surfaces [Gou71](left) Same scene with textured surfaces and grid visualization (right).

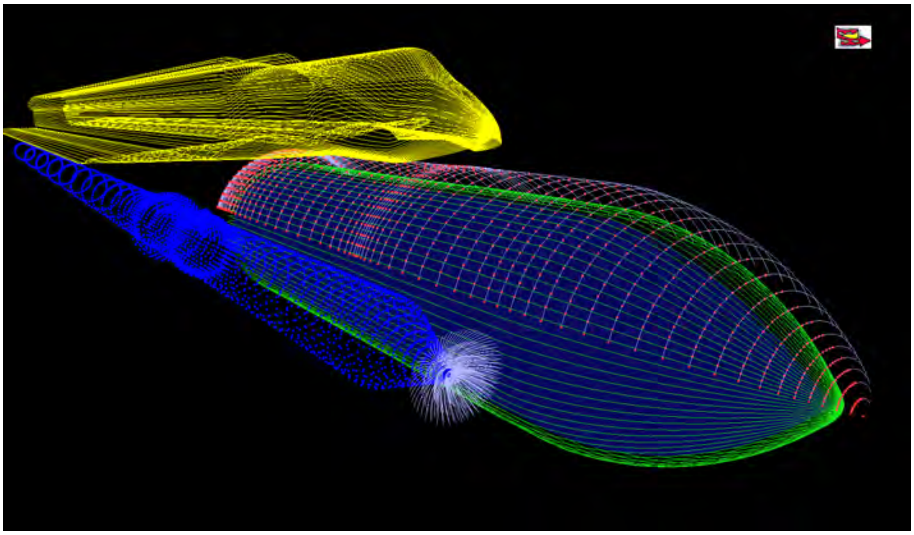

Figure 5.24: CashFlow Grids I: Fusion of several grids using different rendering styles.

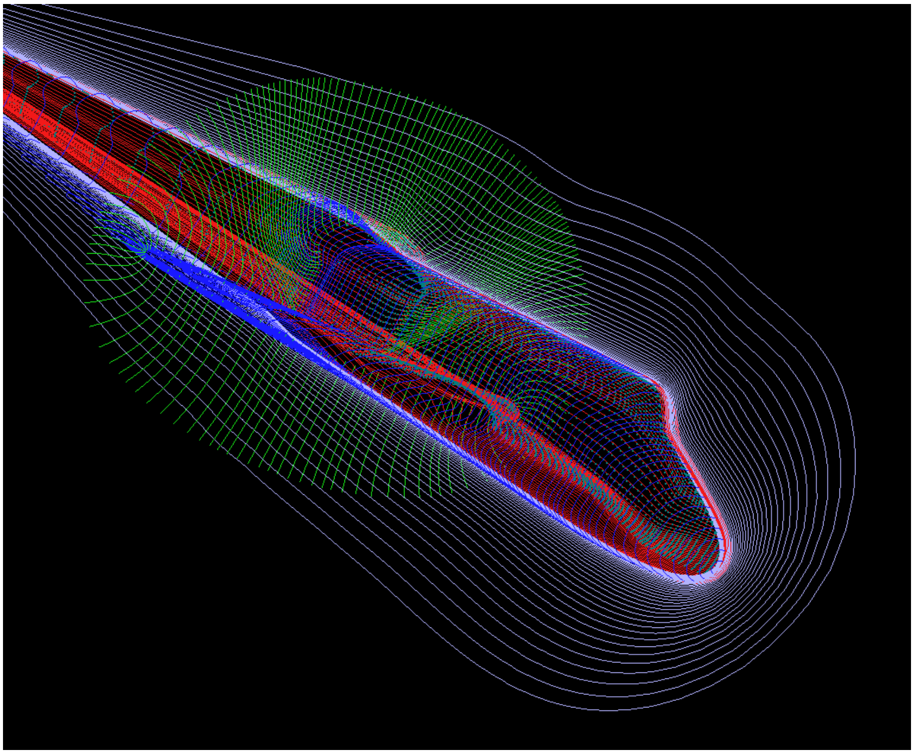

Figure 5.25: CashFlow Grids II: Cyan lines show the X–Z plane using concentric lines while the green lines show the X–Y pane using radial lines. The shuttle in the center is rendered with concentric blue lines in the X–Y plane and red lines in the X–Z plane.

Appendix A

Field connection & TGS MeshVis

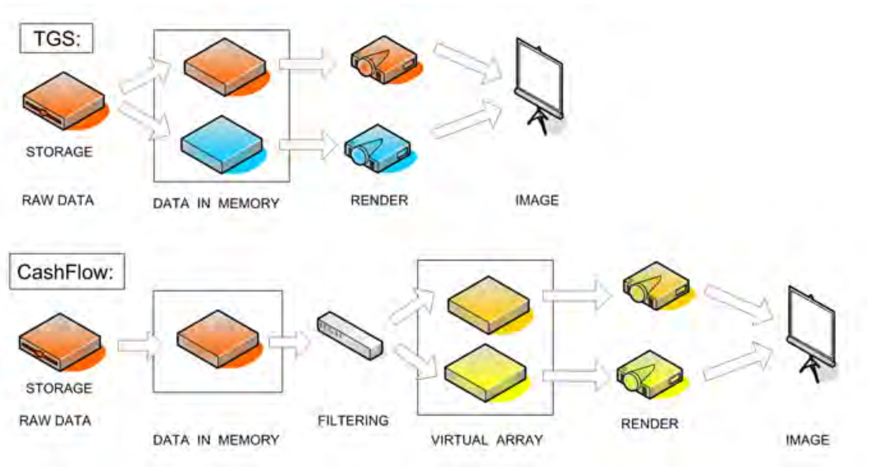

Figure A.1: CashFlow: Virtual Array vs. TGS Meshvis

This is a brief analysis of TGS MeshVis[1] in comparison to the *CashFlow* framework[2].

Each type of grid is assigned to one SoField. Each of these *SoField*s can be rendered by at least one *SoNode*. The *SoField* contains all the information on:

- **type of grid**
- **geometry data**

[1] TGS©MeshVis® formally know as TGS©DataMaster®

[2] "CashFlow - A Visualization Framework for 3D Flow Data" publish in May 2005 at Vienna University of Technology. All rights reserved.

- **topology**
- **additional scalar and vector data**

This is the classic C++ approach using an abstract super–class and sub–classes implementing the abstract interface.

The advantages are the following:

- **classical C++ approach**
  Heavily uses abstract classes and derive specialized classes.
- **simple tree-structure**
  The different types of grids are organized in a simple tree–structure.

The disadvantages are the following:

- **large effort**
  Caused by huge number of *SoField* classes and *SoNode* classes. Each type of grid is represented by one SoField. Each *SoField* has at least one rendering node to visualize it.
- **inflexible**
  Each time raw data should be rendered in another grid, a new copy in a new field has to be created. For instance a carpet plot based on cartesian grid and carpet plot based on cylindrical–polar grid showing the same scalar values must be stored in two separated fields.
- **tight coupling**
  of topology data and other data like scalar data and vector data per grid point.
- **huge class tree**
  due to the reason that each grid has its own *SoField* and *SoNode*.
  - **Improvement:** data storage should only depend on the raw type of data like *float*, *integer*, *string* $\rightarrow$ much smaller amount of classes $\rightarrow$ smaller API, which is easier to understand.

Danksagung

Vielen Dank an meine Schwester Nina und meine Eltern für ihre Unterstützung meiner Arbeit. Auch meiner Oma möchte ich an dieser Stelle herzlich Danken für ihre unermüdliche Aktivität und ihre Unterstützung in schweren Zeiten. Meine innigste Dankbarkeit geht an dich, Elisabeth, dafür, daß ich heute der sein kann, der ich bin.

Außerdem will ich mich bei meinem Betreuer, Dieter Schmalstieg, herzlich dafür bedanken, daß er dieser Arbeit ermöglicht hat und durch seine zahllosen Verbesserungsvorschläge maßgeblich zu dieser Arbeit beigetragen hat. Ohne seinen Einsatz wäre es nicht möglich gewesen, diese Arbeit so abzuschließen. Danke auch an Gerhard Reitmayr für seine aktive Mithilfe am Softwaredesign von CashFlow. Ebenfalls bedanken will ich mich beim Meister Gröller und Helwig Hauser, für deren Eröffnung einer Neuen Welt namens "Visualisierung".

Ich will mich auch bei meinen Freunden bedanken, die mich während der Studienzeit begleitet haben und diese Zeit zu einer unvergeßlichen gemacht haben. Ohne euch wäre es weder lustig noch möglich gewesen. Bei Tamer für die endlosen fruchtbaren Diskussionen über JAVA, C++ und übers Leben. Bei Tom Theußl, für die gemeinsame Zeit diesseits und jenseits des großen Teiches. Bei Ali für die angeregten Gespräche über die Faszination der Mathematik und des Bieres. Bei Sascha, der die Wogen der See immer zu schiffen wußte und mir diese Leidenschaft näher gebracht hat. Bei Erich für seine direkte und fokussierte Art die Dinge anzugehen. Bei Ben für die anregenden Gersprãche über Graphen und allerlei Theorien. Bei Martin für die tolle Zeit abseits der Uni. Bei Frenk für seine Begeisterungsfähigkeit und dafür, daß er die Dinge einfach g'macht hat, anstatt stundenlang d'rüber zu reden. Bei Gottfried, der gezeigt hat, was schon Magelan erfahren hat, nämlich das Reisen bildet. Bei Christopher für's gemeinsame sinnlose Dreschen eines weißen Balls. Bei Gabriel für die Entdeckung der Kreativität und seine Expressivität. Bei Gerhard für sein Organisationstalent und daß er mein Interesse für Medizin in der Informatik geweckt hat. Bei Omar der mich daran erinnert hat, daß man die Sachen auch verkaufen und vermarkten sollte. Bei Maresa, die immer den kürzesten Weg kannte. Bei Andi für seine Unterstützung wider den dunklen Seiten der Informatik. Bei Christian und Günther für die zahllosen gemeinsamen Abende und für ihr Takeda Ryu Budo.

Die Wichtigste zum Schluß. Danke Isabell, dafür daß du immer an mich geglaubt hast und mich immer unterstützt hast, wo und wie du nur irgendwie konntest.

List of Figures

Bibliography

[3D] Java 3D. *Java 3D© by SUN©*. http://java.sun.com/products/java–media/3D/.

[ACT98] Cornelius W. van Overveld Alexandru C. Telea. *An Object–Oriented Interactive System for Scientific Simulations: Design and Applications*, pages 207–220. publ: Springer, Heidelberg, 1998.

[AH02a] Tomas Akenine–Möller and Eric Haines. *Cubic Hermit Interpolation, in Realtime Rendering*, volume 1 ed *1*, chapter Cubic Hermit Interpolation, pages 492–ff. publ: A K Peters, 2002. ISBN156881–182–9.

[AH02b] Tomas Akenine–Möller and Eric Haines, editors. *Real–Time Rendering*. publ: AK Peters, 2002.

[AHB87] B. Freeman–Benson et al. Alan H. Borning, R. Duisberg. *Constraint hierarchies.* In *OOPSLA proceedings 1987*, pages 48–60. ACM. 1987.

[AHH02] Tomas Akenine–Möller, Haines, and Eric Haines. *Realtime Rendering*. publ: A K Peters, 2nd edition, 2002. ISBN 156881–182–9.

[Aki92a] Allen Akin. *Analysis of OpenGL vs. PEX, Analysis of PEX 5.1 and OpenGL 1.0.* In *SIGGRAPH1992*. SGI. 1992.

[Aki92b] Allen Akin. *OpenGL® by SGI©* . Silicon Graphics Inc., 1992. release of OpenGL 1.0.

[Ali] Alias. *Alias Wavefront Systems Corp.* http://www.alias.com.

[AT95] Greg Abram and Lloyd Treinish. *An Extended Data–Flow Architecture for Data Analysis and Visualization.* In *proceedings of IEEE Visualization 1995*, page 263. IEEE Computer Society, Washington, DC, USA, 1995.

[Aut] Autodesk. *AutoDesk Inc.* http://www.autodesk.com/.

[AVS92] AVS. *Advanced Visualization System.* Advanced Visual Systems Inc; 1992. AVS Express see UPSON1989AVS, http://www.avs.com/.

[BHH99] Gunther Webery Bjoern Heckel and Bernd Hamann. *Construction of Vector Field Hierarchies*. In *proceedings of IEEE Visualization 1999*, pages 19–26, Universitĺat Kaiserslautern, University of California. 1999.

[BJ97] Wilfrid Lefer Bruno Jobard. *Creating Evenly–Spaced Streamlines of Arbitrary Density*. In *proceedings of IEEE Visualization 1997*. EuroGraphics Workshop. 1997.

[BJH00] Gordon Erlebacher Bruno Jobard and M. Yousuff Hussaini. *Hardware–Accelerated Texture Advection For Unsteady Flow Visualization*. In *proceedings of IEEE Visualization 2000*, pages 155–162, Florida State University. 2000.

[BL92] Nancy S. Collins Bruce Lucas, Gregory D. Abram, David A. Epstein, Donna L. Gresh, Kevin P. McAuliffe. *An architecture for a scientific visualization system*. In *proceedings of IEEE Visualization 1992*, pages 107–114, Los Alamitos, CA, USA. IEEE Computer Society Press. 1992.

[BLC01] Guillaume Caumon Bruno Lèvy and Stèphane Conreaux. *Circular incident edge lists: a data structure for rendering complex unstructured grids*. In *proceedings of IEEE Visualization 2001*, pages 159–166, Vandoeuvre, France. University of Kaiserslautern, IEEE. 2001.

[Bor79] Alan H. Borning. *Thinglab– A constraint oriented simulation laboratory*. PhD thesis, Stanford University, 1979. successor of Sketchpad, http://www.2share.com/thinglab/ThingLabReport.zip.

[Bor81] Alan H. Borning. *The Programming Language Aspects of ThingLab, a Constraint–Oriented Simulation Laboratory*. In *ACM Transactions on Programming Languages and Systems*, volume 3 ed *4*, pages 353–387. successor of Sketchpad, Okt 1981.

[Bur94] M. M. Burnett. *Visual object–oriented programming, concepts and environments*. publ: Prentice Hall, 1994.

[BW86] Gary Bishop and David M. Weimer. *Fast Phong shading*. In *proceedings of Siggraph 1986*, pages 103–106, New York, NY, USA. ACM. 1986.

[Cal91] Brian Calvert. *Interactive Analysis of Multidimensional Data*. Master's thesis, University of Illinois Department of Computer Science, 1991. Masters Thesis.

[CC91] I. Currington and M. Coutant. *AVS – A Flexible Interactive Distributed Environment for Scientific Visualization Applications*. In *Second EuroGraphics Workshop on Visualization in Scientific Computing*. April 1991.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*. In *proceedings of VVS 1994*, pages 91–98, New York, NY, USA. ACM. 1994.

[CEI94] CEI. *http://www.ceintl.com/.* Computational Engineering International, Inc., 1994. CFD software.

[CG04] Tobias Salzbrunn Christoph Garth, Xavier Tricoche, Tom Bobach, Gerik Scheuermann. *Surface Techniques for Vortex Visualization.* In *proceedings of VisSym 2004*, pages 155–164. 2004.

[Chi02a] Ed H. Chi. *Expressiveness of the Data Flow and Data State Models in Visualization Systems.* In *Advanced Visual Interfaces Conference*, pages 375–378, Trento, Italy. 2002.

[Chi02b] Ed H. Chi. *A Framework for Visualizing Information.* publ: Kluwer Academic Publishers, Netherlands, July 2002. 176 pages, http://www2.parc.com/istl/projects/uir/pubs/items/UIR–2002–03–Waterson–AVI–WebQuilt.pdf.

[Chi03] Yi–Jen Chiang. *Out–of–Core Isosurface Extraction of Time–Varying Fields over Irregular Grids.* In *proceedings of IEEE Visualization 2003*, pages 217–224, Polytechnic University USA. 2003.

[CL93] Brian Cabral and Leith Casey Leedom. *Imaging Vector Fields Using Line Integral Convolution.* In James T. Kajiya New York, editor, *proceedings of Siggraph 1993*, pages 263–Ű269. LIC, 1993.

[CM92] Roger Crawfis and Nelson Max. *Direct Volume Visualization of Three Dimensional Vector Fields.* In *proceedings of the 1992 Workshop on Volume Visualization*, pages 55Ű–60. ACM. 1992.

[CMPS96] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. *Optimal isosurface extraction from irregular volume data.* In *proceedings of VVS 1996*, pages 31–38, Piscataway, NJ, USA. IEEE Press. 1996.

[Coi00] Coin3D. *Coin3D Open Inventor library*, 2000. Coin3D, Open Inventor library by Systems In Motion (SIM),Coin 1.0 released in 2000, http://www.coin3d.org/.

[CRE99] C. Teitzel C. Rezk–Salama, P. Hastreiter and Thomas Ertl. *Interactive exploration of volume line integral convolution (LIC) based on 3D texture mapping.* In *proceedings of IEEE Visualization 1999*, pages 233–240, Universitay of Erlangen. 1999.

[CUL89] David Kamins Craig Upson, Thomas Jr Faulhaber and David H. Laidlaw. *The Application Visualization System: A Computational Environment for Scientific Visualization.* In *IEEE Computer Graphics and Applications*, volume 9 ed *4*, pages 30–42. other authors: David Schlegel, Jefrey Vroom, Robert Gurwitz and Andries van Dam AVS system paper, July 1989.

[Des] Coin Designer. *coin designer, a visual programming editor for coin.* Open source project. http://coindesigner.sourceforge.net/.

[DHL01] J. Scott Davidson David H. Laidlaw, R. M. Kirby, Timothy S. Miller, Marco da Silva, William H. Warren and Michael Tarr. *Quantitative comparative evaluation of 2D vector field visualization method.* In *proceedings of IEEE Visualization 2001,*, pages 143–150, Brown University, Road Iland. 2001.

[dM] 3ds Max. *3DStudio max by AutoDesk Inc.©* . http://www.autodesk.com/.

[dMG93] Vicki de Mey and Simon Gibbs. *A MultiMedia Component Kit: Experiences with visual Composition of Applications.* In *proceedings ACM Multimedia*, pages 291 – 200. ACM. 1993.

[Dor03] Alois Dornhofer. *A Discrete Fourier Transform Pair for Arbitrary Sampling Geometries with Applications to Frequency Domain Volume Rendering on the Body–Centered Cubic Lattice.* Master's thesis, Technical University of Vienna and School of Computing Science, Simon Fraser University, Vancouver, Canada,, March 2003.

[Dov95] Don Dovey. *Vector Plots for Irregular Grids.* In *proceedings of Visualization 1995*, pages 248Ű–253, Atlanta, Georgia. IEEE. Oct 1995.

[DQNJ02] Ronald Fedkiw Duc Quang Nguyen and Henrik Wann Jensen. *Physically based modeling and animation of fire.* In *proceedings of Siggraph 2002*, pages 721–728, New York, NY, USA. ACM. 2002.

[dv95] Willem C. deLeeuw and Jarke J. vanWijk. *Enhanced Spot Noise for Vector Field Visualization.* In *proceedings Visualization 1995*, pages 233–239. IEEE. Oct 1995.

[dv99] Willem C. deLeeuw and Robert vanLiere. *Collapsing Flow Topology Using Area Metrics.* In *proceedings of IEEE Visualization 1999*, pages 413–416, Center for Mathematics and Computer Science. 1999.

[DVA96] *DVA, visual programming interface for VTK.* Apricot Software, 1996. http://www.data–visualization–software.com.

[DX91] Open DX. *Open Visualization Data Explorer© by IBM®*, 1991. open sourece project sponsered by IBM Inc., http://www.opendx.org/.

[Dye90] D. Scott Dyer. *Visualization: A Dataflow Toolkit for Visualization.* In *IEEE Computer Graphics and Applications*, volume 10 ed *4*, pages 60–69, Los Alamitos, CA, USA. IEEE Computer Society Press. 1990.

[Eng01] *High–Quality Pre–Integrated Volume Rendering Using Hardware–Accelerated Pixel Shading*, 2001.

[Exp71] IRIX Explorer. *IRIX Explorer® by SGI©.* Numerical Algorithms Group, 1971. Dr Steve Jenkins, The Numerical Algorithms Group Ltd., Oxford UK, http://www.nag.co.uk/.

[EYSK94] David S. Ebert, Roni Yagel, Jim Scott, and Yair Kurzion. *Volume rendering methods for computational fluid dynamics visualization.* In *proceedings of IEEE Visualization 1994*, pages 232–239, Los Alamitos, CA, USA. IEEE Computer Society Press. 1994.

[Fak] FakeSpaceBoom. *Fakespace Labs Inc.* Fakespace Labs Inc. http://www.fakespacelabs.com.

[FF88] Leila De Floriani and Bianca Falcidieno. *A hierarchical boundary model for solid object representation. ACM Trans. Graph.*, 7(1):42–60, 1988.

[FH94] Jean M. Favre and James Hahn. *An Object Oriented Design for the Visualization of Multi–Variable Data Objects.* In *proceedings of IEEE Visualization 1994*, pages 318–325. 1994.

[Fie97] FieldView. *FieldView.* Intelligent Light, 1997. CFD post–processing software, http://www.ilight.com/.

[FLvD$^+$00] Andrew S. Forsberg, David H. Laidlaw, Andries van Dam, Robert M. Kirby, George E. Karniadakis, and Jonathan L. Elion. *Immersive virtual reality for visualizing flow through an artery.* In *proceedings of IEEE Visualization 2000*, pages 457–460, Los Alamitos, CA, USA. IEEE Computer Society Press. 2000.

[Gam95] Erich Gamma. *DesignPatters.* publ: Addison–Wesley, 1995. ISBN 0–201–63361–2.

[Gam96] Erich Gamma. *Entwurfsmuster.* publ: Addison–Wesley, 1996. ISBN 3–827–32199–9.

[Gay96] Jonathan Gay. *Macromedia Flash.* Macromedia Inc., December 1996. FutureSplash sold to macromedia and became Macromedia Flash 1.0, http://www.macromedia.com/software/flash/.

[GKM03] Tolga Tasdizen Gordon Kindlmann, Ross Whitaker and Torsten Möller. *Curvature–Based Transfer Functions for Direct Volume Rendering: Methods and Applications.* In *proceedings Visualization 2003*, pages 513–520. 2003.

[GL05] Markus Grabner and Robert S. Laramee. *Image Space Advection on Graphics Hardware.* In *proceedings of SCCG2005*, pages 75–82. 2005.

[Gou71] Henri Gouraud. *Continuous shading of curved surfaces.* In *IEEE Transactions on Computers*, volume 20 ed 6, pages 623–628. IEEE. 1971.

[GPL85] GNU GPL. *GNU General Public License (GPL) by Free Software Foundation (FSF)*, 1985. Open Source Projects, http://www.gnu.org/.

[GR95a] Gary A. Glatzmaier and Paul H. Roberts. *A three–dimensional self–consistent computer simulation of a geomagnetic field reversal.* In *Nature, 377*, pages 203–209. 1995, http://www.es.ucsc.edu/ glatz/geodynamo.html.

[GR95b] Gary A. Glatzmaier and P.H. Roberts. *A three–dimensional convective dynamo solution with rotating and finitely conducting inner core and mantle.* In *Phys. Earth Planet. Inter.*, pages 63–75. 1995, http://www.es.ucsc.edu/ glatz/geodynamo.html.

[GS01] Hans Hagen Gerik Scheuermann, Tom Bobach, Karim Mahrous, Bernd Hamann, Kenneth I. Joy and Wolfgang Kollmann. *A tetrahedra–based stream surface algorithm.* In *proceedings of IEEE Visualization 2001*, pages 505–508. 2001.

[GSLS03] Udeepta D. Bordoloi Guo Shi Li and Han Wei Shen. *A Topology Simplification Method for 2D Vector Fields.* In *proceedings of IEEE Visualization 2003*, pages 241–248. Ohio State University, IEEE. 2003.

[HD90] Karl–Heinz Küttner Heinrich Dubbel, Wolfgang Beitz, editor. *DUBBEL Taschenbuch für den Maschinenbau.* publ: Springer, 17th edition, 1990. 17th edition, german.

[HDD94] Karl–Heinz Küttner Heinrich Dubbel, Wolfgang Beitz and B. J. Davies, editors. *DUBBEL Handbook of Mechanical Engineering.* publ: Springer, 18th edition, 1994. 18th edition, english.

[HGPR99] B. Henne H. G. Pagendarm and M. Rütten. *Case Study: Detecting vortical phenomena in vector data by medium–scale correlation.* In *proceedings of IEEE Visualization 1999*, pages 409–412, DLR Göttingen, Germany. 1999.

[Hil91] Daniel D. Hils. *Datavis: a visual programming language for scientific visualization.* In *CSC '91: Proceedings of the 19th annual conference on Computer Science*, pages 439–448, New York, NY, USA. ACM. 1991.

[HLG98] Helmut Doleisch Helwig Löffelmann and Eduard Gröller. *Visualizing dynamic systems near critical points.* In *proceedings SCCG 1998*, pages 175–184. Helwig Hauser, 1998.

[HLS03] Mathieu Desbrun Haeyoung Lee and Peter Schröder. *Progressive encoding of complex isosurfaces.* In *proceedings SIGGRAPH2003*, pages 471–476. ACM. 2003.

[HM76] Peter Henderson and Jr. James H. Morris. *A lazy evaluator.* In *proceedings of the 3rd ACM SIGACT–SIGPLAN*, pages 95–103, Atlanta, Georgia, USA. January 19–21 1976.

[HM03] Helwig Hauser and Matej Mlejnek. *Interactive Volume Visualization of Complex Flow Semantics.* In *proceedings of VMV 2003*. 2003.

[HP97] Hans – Christian Hege and Konrad Polthier. *Visualization and Mathematics*. publ: Springer, 1997. ISBN 3–540–61269–6.

[HTS03] Hans–Christian Hege Holger Theissel, Tino Weinkauf and Hans–Peter Seidel. *Saddle Connectors – An Approach to Visualizing the Topological Skeleton of Complex 3D Vector Fields*. In *proceedings of IEEE Visualization 2003*, pages 225–232. IEEE. 2003.

[HWB96] Ken Brodlie Helen Wright and Martin Brown. *The Dataflow Visualization Pipeline as a Problem Solving Environment*. publ: Springer, 1996.

[IG97] Victoria L. Interrante and Chester Grosch. *Strategies for effective visualizing 3D flowwith volume LIC*. In *proceedings of Visualization 1997*. IEEE. 1997.

[Ing78] Daniel H. H. Ingalls. *The Smalltalk–76 programming system design and implementation*. In *proceedings of the 5th ACM SIGACT–SIGPLAN*, pages 9–16, New York, NY, USA. ACM. 1978.

[Inv92] Open Inventor. *OpenInventor® by SGI©*, 1992. scene graph library, open source project, http://www.sgi.com/products/software/inventor/.

[IWC$^+$88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. *Fabrik: a visual programming environment*. In *OOPSLA proceedings 1988*, volume 23 ed *11*, pages 176–190, New York, NY, USA. ACM. 1988.

[JD92] Pat Hanrahan John Danskin. *Fast algorithm for volume ray tracing*. In *proceedings of ACM volume visualization 1992*, pages 91–98. ACM. 1992.

[KBG03] Bruce Walter Kavita Bala and Donald P. Greenberg. *Comining Edges and points for interactive high–quality rendering*. In *proceedings SIGGRAPH2003*, pages 631–640. ACM. 2003.

[KEE99] Rüdiger Westermann Klaus Engel and Thomas Ertl. *Isosurface extractions techniques for web–based volume visualization*. In *proceedings of IEEE Visualization 1999*, pages 139–146, University of Stuttgart, University of Computersience Utah. 1999.

[KMC99] Torsten Möller Kalus Mueller and Roger Crawfis. *Splatting without the blur*. In *proceedings IEEE Visualization 1999*, pages 363–370. IEEE. October 1999.

[KSK01] Barry Lazos Kurt Severance, Paul Brewster and Daniel Keefe. *Wind tunnel data fusion and immersive visualization: A case study*. In *proceedings of IEEE Visualization 2001*, pages 505–508. NASA, Brown University, IEEE. 2001.

[LB02] Robert S. Laramee and R. Daniel Bergeron. *An Isosurface Continuity Algorithm for Super Adaptive Resolution Data.* In *proceedings of CGI Advances in Modelling, Animation, and Rendering*, pages 215–237, Bradford, UK. July 2002.

[LC85] R.A. Reynolds L. Chen, G.T. Herman, J. K. Udupa. *Surface shading in the cuberille environment.* In *IEEE Computer Graphics and Applications*, volume 12(3), pages 33–43. IEEE. cube rille rendering style, 1985.

[LC87] William E. Lorensen and Harvey E. Cline. *Marching cubes: A high resolution 3D surface construction algorithm.* In *proceedings Computer Graphics and Interactive Techniques*, pages 163–169. 1987.

[Lev88] Marc Levoy. *Display of Surfaces from Volume Data.* In *IEEE Computer Graphics and Applications*, volume 8 ed *3*. IEEE. 1988.

[Lev90] Mark Levoy. *Efficient Ray Tracing of Volume Data.* In *Transactions on Graphics*, volume 9 ed *3*, pages 245–261. ACM. 1990.

[Lev92] Marc Levoy. *Volume rendering using the Fourier projection–slice theorem.* In *Proceedings of the conference on Graphics interface '92*, pages 61–69, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. 1992.

[LG90] et al L. Gelberg. *Visualization Techniques for Structured and Unstructured Scientific Data.* In *Course Notes, Siggraph 1990 Course "State of the Art in Data Visualization"*. 1990.

[LL94] Philippe Lacroute and Marc Levoy. *Fast volume rendering using a shear–warp factorization of the viewing transformation.* In *proceedings of Siggraph 1994*, pages 451–458, New York, NY, USA. ACM. 1994.

[Löf98] Helwig Löffelmann. *Visualizing Local Properties and Characteristic Structures of Dynamical Systems.* PhD thesis, Technical University of Vienna, 1998.

[Mac] Macromedia. *Macromedia Inc.©*. http://www.macromedia.com.

[Mal93] Tom Malzbender. *Fourier volume rendering.* In *ACM Transactions on Graphics*, volume 12 ed *3*, pages 233–250. July 1993.

[Mat05] MatLab. *MatLab ® by MathWorks Inc.* MathWorks Inc.©, 1994–2005. http://www.mathworks.com/.

[Max86] Nelson Max. *Light diffusion through clouds and haze.* In *Computer Vision, Graphics and Image Processing*, volume 33, pages 280–292. 1986.

[Max95] Nelson Max. *Optical Models for Direct Volume Rendering.* In *IEEE Educational Activities Department*, volume 1 ed *2*, pages 99–108, Piscataway, NJ, USA. IEEE Educational Activities Department. 1995.

[MAY] MAYA. *MAYA© by ALIAS©*. Modeling software, support motion–capture data, http://www.alias.com/maya/.

[Mes] MeshVis. *MeshVis by TGS©*. former DataMaster, visualization of grids, http://www.tgs.com/pro_div/DataViz_main.htm.

[MHH03] C. Berger Markus Hadwiger and H. Hauser. *High–Quality Two–Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware*. In *proceedings of IEEE Visualization 2003*, pages 301–308. IEEE. 2003.

[MJHL02] Thorsten Scheuermann Mark J. Harris, Greg Coombe and Anselmo Lastra. *Physically–based visual simulation on graphics hardware*. In *proceedings of ACM Siggraph/EuroGraphics HWWS 2002*, pages 109–118, Aire–la–Ville, Switzerland, Switzerland. EuroGraphics Association. 2002.

[Mle03] Matej Mlejnek. *Modelling the Visualization Mapping for Volumetric Flow Visualization*. Master's thesis, Technical University of Vienna, VRVis, 2003. http://www.vrvis.at/vis/resources/DA–MMlejnek/.

[Mor01] Patrick Moran. *Field Model: An Object–Oriented Data Model for Fields*. Technical report, NASA, 2001. open source C++ Template library, http://www.nas.nasa.gov/News/Techreports/2001/PDF/nas–01–006.pdf.

[Mor03] Patrick Moran. *Field Model C++ specialized template library*, 2003. http://field–model.sourceforge.net/.

[Motay] Systems In Motion. *Systems In Motion AS*. Systems In Motion AS, founded 1994 in Trondheim, Norway. http://www.sim.no.

[MSE99] Wolf Bartelheimer Martin Schulz, Franz Reck and Thomas Ertl. *Interactive visualization of fluid dynamics simulations in locally refined cartesian grids*. In *proceedings of IEEE Visualization 1999*, pages 413–416. University of Stuttgard, University of Erlangen, IEEE. 1999.

[mSK98] Han Wei Shen and D. L. Kao. *A new line integral convolution algorithm for visualizing time–varying flow fields*. In *IEEE Trans. Visualization Computer Graphics*, volume 4 Nr. 2, page 98Ű108. 1998.

[MTHG03] Oliver Mattausch, Thomas Theußl, Helwig Hauser, and Eduard Gröller. *Strategies for interactive exploration of 3D flow using evenly–spaced illuminated streamlines*. In *proceedings of SCCG 2003*, pages 213–222, New York, NY, USA. ACM. 2003.

[MvR96] James D. Murray and William van Ryper. *Encyclopedia of Graphics File Formats*. publ: O'Reilly + Ass., 2nd edition, 1996. MOELLER2002 page 444 ref[577].

[MZH96] Detlev Stalling Malte Zöckler and Hans–Christian Hege. *Interactive viasulatization of 3D–vector fields using illuminated streamlines*. In *proceedings IEEE Visualization 1996*, pages 107–113. IEEE. 1996.

[NAS] NAS NASA. *NAS System division Office, part of NASA*. http://www.nas.nasa.gov/.

[NAS05] NAS NASA. *Sample Datasets from NAS NASA*, 2005. http://www.nas.nasa.gov/Research/Datasets/datasets.html.

[Nie03] Gregory M. Nielson. *MC*: Star Functions for Marching Cubes*. In *proceedings of IEEE Visualization 2003*, pages 59–66, Arizona State University. 2003.

[NRF03] Willi Geiger Nick Rasmussen, Duc Quang Nguyen and Ronald Fedkiw. *Smoke simulation for large scale phenomena*. In *proceedings SIGGRAPH2003*, pages 703–707. ACM. 2003.

[OWD$^+$96] Upul Obeysekare, Chas Williams, Jim Durbin, Larry Rosenblum, Robert Rosenberg, Fernando Grinstein, Ravi Ramamurthi, Alexandra Landsberg, and William Sandberg. *Virtual workbench – a non–immersive virtual environment for visualizing and interacting with 3D objects for scientific visualization*. In *proceedings of IEEE Visualization 1996*, pages 345–ff., Los Alamitos, CA, USA. IEEE Computer Society Press. 1996.

[PAD03] David Cohen–Steiner Pierre Alliez and Oliver Devillers. *Anisotropic polygonal Remeshing*. In *proceedings SIGGRAPH2003*, pages 485–493. ACM. Bruno Lèvy and Mathieu Desbrun, 2003.

[Pas04] Valerio Pascucci. *Isosurface Computation Made Simple: HArdware acceleration, adaptive refinement and tetrahedral stripping*. In *proceedings of VisSym 2004*, pages 293–300. GPU–based isosurface extration, 2004, http://www.pascucci.org/pdfpapers/vissym2004.pdf.

[Per85] Ken Perlin. *An image synthesizer*. In *proceedings of Siggraph 1985*, pages 287–296, New York, NY, USA. ACM. 1985.

[Pho73] Bui Tuong Phong. *Illumination for computer–generated images*. PhD thesis, University of Utah,Salt Lake City, 1973. Phong shading.

[PL90] D. Parker and Y. Lin. *The Application Visualization System for Finite Element Analysis*. In *Banff Conference on FEA*. May 1990.

[PN01] Ken Perlin and Fabrice Neyret. *Flow Noise*. In *proceedings of Siggraph Technical Sketches and Applications*, page 187. Aug 2001, http://www–imagis.imag.fr/Publications/2001/PN01.

[RAEM94] William Ribarsky, Eric Ayers, John Eble, and Sougata Mukherjea. *Glyphmaker: Creating Customized Visualizations of Complex Data*. *Computer*, 27(7):57–64, 1994.

[RBHC91] B. Lucas R. B. Haber and N. Collins. *A data model for scientific visualization with provisions for regular and irregular grids*. In *proceedings of IEEE Visualization 1991*, pages 298 – 305. 1991.

[RBJ01] Bernd Hamann Ralph Bruckschen, Falko Kuester and Kenneth I. Joy. *Real–time Out–of–Core Visualization of Particle Traces*. In *IEEE Symposium in Parallel and Large–Data Visualization and Graphics 2001*, pages 45–50, University of California. IEEE. Oct 2001.

[RMKL99] H. Marmanis R. M. Kirby and D. H. Laidlaw. *Visulazing multivalued data from 2D incopressible flows using concepts from painting*. In *proceedings of IEEE Visualization 1999*, pages 333–340, Brown University. 1999.

[RP99] Peter Rechenberg and Gustav Pomberger, editors. *Informatik Handbuch*. publ: Carl Hanser Verlag Wien, 2nd edition, 1999. german.

[RSL04] Jurgen Schneider Robert S. Laramee, Daniel Weiskopf, Helwig Hauser. *Investigating Swirl and Tumble Flow with a Comparison of Visualization Techniques*. In *proceedings of IEEE Visualization 2004*, pages 51–58, Washington, DC, USA. IEEE Computer Society. 2004.

[RSLH05] Markus Hadwiger Robert S. Laramee and Helwig Hauser. *Design and Implementation of Geometric and Texture–Based Flow Visualization Techniques*. In *proceedings of SCCG2005*, pages 67–74. 2005.

[RWE00] Christopher Johnson Rüdiger Westermann and Thomas Ertl. *A Level–Set Method for Flow Visualization*. In *proceedings IEEE Visualization 2000*, pages 147–154. Aachen and Stuttgart, Germany, Utah, IEEE. 2000.

[Sad99] Ari Sadarjoen. *Extraction and Visualization of Geometries in Fluid Flow Fields*. PhD thesis, Delft University of Technology, 1999.

[Sam90] H. Samet. *The design and analyisis of spacial data structures*. publ: Addison–Wesley, 1990. B–Rep list.

[Sch96a] S. Schiffer. *Visuelle Programmierung – Potential und Grenzen*. publ: Oldenburg Verlag Wien, 1996. german, in "Beherschung von Informationssystemen".

[Sch96b] Dieter Schmalstieg. *Studierstube*, 1996. http://www.studierstube.org/.

[Sch97a] Dieter Schmalstieg. *The remote rendering pipeline*. PhD thesis, Technical University of Vienna, 1997.

[Sch97b] William J. Schroeder, editor. *The Visualization Toolkit User's Guide*. publ: Kitware, Inc. publishers, Dezember 1997. ISBN 1–930934–08–4, VTK Users Guide.

[Sch98] Stefan Schiffer. *Visuelle Programmierung – Grundlagen und Einsatzmöglichkeiten*. publ: Addison–Wesley, 1998. german.

[SCI02] SCIRun, 2002. SCIRun: A Scientific Computing Problem Solving Environment, http://software.sci.utah.edu/scirun.html.

[SE04] Simon Stegmaier and Thomas Ertl. *A Graphics Hardware–Based Vortex Detection and Visualization System.* In *proceedings of IEEE Visualization 2004*, pages 195–202, Washington, DC, USA. IEEE Computer Society. 2004.

[SGI] SGI. *SGI®*. Silicon Graphics Inc. http://www.sgi.com.

[SK97] Han–Wei Shen and David L. Kao. *UFLIC: A Line Integral Convolution Algorithm For Visualizing Unsteady Flows.* In Roni Yagel and Hans Hagen, editors, *Visualization '97*, pages 317–323. IEEE. 1997.

[SKCM99] Ben Shneiderman Stuart K. Card, Jock D. Mackinlay and Jock D. McKinley. *Readings in Information Visualization.* publ: Morgan Kaufmann, 1999. ISBN 155860533–9.

[SM00] Heidrun Schumann and Wolfgang Müller. *Visualisierung, Grundlagen und allgemeine Methoden.* publ: Springer, 2000.

[Sta99] Jos Stam. *Stable fluids.* Technical report, proceedings of Siggraph 1999, New York, NY, USA, 1999.

[Sta03] Jos Stam. *Flows on surfaces of arbitrary topology.* Technical Report 3, Alias Wavefront Systems Corp, New York, NY, USA, 2003.

[Sut63] Ivan Sutherland. *A man–machine graphical graphical communication system in interactive computer graphics.* In *Spring Joint Computer Conference Proceedings.* Sketchpad, on TX–2 mainframe at MIT's Lincoln Labs in 1962. It allows him to make engineering drawings with a light pen., 1963.

[TB96] G. Turk and D. Banks. *Image–guided streamline placement.* In *proceedings of Siggraph 1996*, pages 453–460. ACM. 1996.

[TCCG99] Daniel S. Katz Thomas C. Clune and Gary A. Glatzmaier. *Advances in Modeling the Generation of the Geomagnetic Field by the Use of Massively Parallel Computers and Profound Optimization.* In *proceedings of the SIAM Conference 1999*, San Antonio, Texas, USA. March 1999.

[Tec81] TecPlot. *TecPlot CFD analysis software*, Jan 1981. CFD software, http://www.tecplot.com/.

[TGS] TGS. *TGS Mercury Computer Systems Inc.* http://www.tgs.com/.

[TJW02] Scott Schaefer Tao Ju, Frank Losasso and Joe Warren. *Dual Contouring of Hermit Data.* In *proceedings SIGGRAPH2002*, pages 339–346, Rise University. ACM. 2002.

[Tog02] Together. *Borland® Together®*. Borland Inc., Jan 2002. UML diagram tool, visual programming, http://www.borland.com/together/.

[Too00] Scalable Visualization Toolkits. *Scalable Visualization Toolkits*, 2000. http://vistools.npaci.edu/.

[TTS03] Ch. Rössl T. Theisel and H.–P. Seidel. *Saddle Connectors – An Approach to Visualizing the Topological Skeleton of Complex 3D Vector Fields*. In *proceedings of IEEE Visualization 2003*, Saarbrücken, Germany. IEEE. 2003.

[Tut01] IEEE. *IEEE Visualization Tutorial Nr. 1 on "Visualization Systems"*, 2001.

[Tv99] Alexandru C. Telea and Jarke J. vanWijk. *VISSION: An Object Oriented Dataflow System for Simulation and Visualization*. In *In porceedings of VisSym 1999*. IEEE TCVG Symposium on Visualization. 1999.

[TvW99] Alexandru Telea and Jarke J. van Wijk. *Simplified representation of vector fields*. In *proceedings of IEEE Visualization 1999*, pages 35–42. IEEE. Eindhoven University of Technology, 1999.

[VBL$^+$03] Fabien Vivodtzev, Georges-Pierre Bonneau, Lars Linsen, Bernd Hamann, Kenneth I. Joy, and Bruno A. Olshausen. *Hierarchical isosurface segmentation based on discrete curvature*. In *VISSYM '03: Proceedings of the symposium on Data visualization 2003*, pages 249–258, Aire–la–Ville, Switzerland, Switzerland. EuroGraphics Association. 2003.

[Vol10] SIM Voleon. *SIM VOLEON Volume rendering package, extension to Coin3D Open Inventor*. System In Motion AS, 2004 release 1.0. open source project, http://www.sim.no/products/SIM_Voleon/.

[VRM] VRML. *VRML, Virtual Reality Modeling Language*. published by SGI.

[Vro94] Jeffrey Vroom. *Object–Oriented Application Development with AVS/Express*. In *proceedings of AVS 1994 Conference*. 1994.

[Vro95] Jeffrey Vroom. *AVS Express: A New Programming Paradigm*. In *proceedings of AVS 1995 Conference*. 1995.

[VTK96] VTK. *VTK The Visualization Toolkit*, 1996. by William J. Schroeder, Ken Martin and Bill Lorensen,publisher by Kitware Inc., http://www.kitware.com.

[VVP99] David Kao Vivek Varma and Alex Pang. *PLIC: Bridging the gap between streamlines and LIC*. In *proceedings of IEEE Visualization 1999*, pages 341–348. NASA Ames Research Center, IEEE. 1999.

[VVP00] David Kao Vivek Verma and Alex Pang. *A Flow–guided Streamline Seeding Strategy*. In *proceedings of IEEE Visualization 2000*, pages 163–170. Brown University, IEEE. 2000.

[vW91] Jarke J. van Wijk. *Spot Noise -Ű Texture Syntheses for Data Visualization*. In *proceedings of Siggraph 1991*, volume 25 Nr.4, pages 309–318. ACM. In Computer Graphics, 1991.

[vW02] Jarke J. van Wijk. *Image based flow visualization*. In *proceedings SIGGRAPH2002*, pages 745–754. Technical University of Eindhoven. 2002.

[vW03] Jarke J. van Wijk. *Image Based Flow Visualization for Curved Surfaces*. In *proceedings of IEEE Visualization 2003*, pages 123–130, Technische Universiteit Eindhoven. 2003.

[vWT01] Jarke J. van Wijk and Alexandru Telea. *Enridged contour maps*. In *proceedings of IEEE Visualization 2001*, pages 69–74, Eindhoven University of Technology. 2001.

[Wad84] Philip Lee Wadler. *Listlessness is better than laziness: Lazy evaluation and garbage collection at compile–time*. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 45–52, New York, NY, USA. ACM. 1984.

[Wal95] Theo van Walsum. *Selective visualization on curvilinear grids*. PhD thesis, Delft University of Technology, 1995.

[Wer93] Josie Wernecke. *The Inventor Mentor: Extending Open Inventor*. publ: Addison–Wesley, 2nd edition edition, November 1993.

[Wer94] Josie Wernecke. *The Inventor Toolmaker*. publ: Addison–Wesley, 2nd edition edition, April 1994.

[Wes90] Lee Westoven. *Footprint evaluation for volume rendering*. In *proceedings of Siggraph 1990*, pages 367–376, New York, NY, USA. ACM. original paper for splatting, 1990.

[WG97] Rainer Wegenkittl and Eduard Gröller. *Fast oriented line integral convolution for vector field visualization via the Internet*. In *proceedings of IEEE Visualization 1997*, pages 309–316, Los Alamitos, CA, USA. IEEE Computer Society Press. 1997.

[WJSL96] Kenneth M. Martin William J. Schroeder and William E. Lorensen. *The design and implementation of an object–oriented toolkit for 3D graphics and visualization*. In *proceedings of IEEE Visualization 1996*, pages 93–ff. paper for VTK, 1996.

[WJSL97] Ken Martin William J. Schroeder and Bill Lorensen. *The Visualization Toolkit: An Object–Oriented Approach to 3–D Graphics*. publ: Kitware, Inc. publishers, Dezember 1997. ISBN 1–930934–07–6, VTK Users Guide.

[WJSV92] G. D. Montanaro William J. Schroeder, W. E. Lorensen and C. R. Volpe. *VISAGE: an object–oriented scientific visualization system.* In *proceedings of IEEE Visualization 1992*, pages 219–226, Los Alamitos, CA, USA. IEEE Computer Society Press. 1992.

[WR00] Michel A. Westenberg and J. B. T. M. Roerdink. *X–Ray Volume Rendering by Hierarchical Wavelet Splatting.* In *Machine Graphics and Vision,*, volume 9(1/2), pages 307–314. 2000.

[XTDS$^+$04] Gordon Kindlmann Xavier Tricoche, Christoph Garth, Eduard Deines, Gerik Scheuermann, Markus Ruetten, and Charles Hansen. *Visualization of Intricate Flow Structures for Vortex Breakdown Analysis.* In *proceedings of IEEE Visualization 2004*, pages 187–194, Washington, DC, USA. IEEE Computer Society. 2004.

[XTH00] Gerik Scheuermann Xavier Tricoche and Hans Hagen. *A Topology Simplification Method for 2D Vector Fields.* In *proceedings of IEEE Visualization 2000*, pages 359–366, Germany. University of Kaiserslautern, IEEE. 2000.

[XTH01] Gerik Scheuermann Xavier Tricoche and Hans Hagen. *Continouse topotlogy simpification of planar vector fields.* In *proceedings of IEEE Visualization 2001*, pages 159–166, Germany. University of Kaiserslautern, IEEE. 2001.

[XZC04] Daqing Xue, Caixia Zhang, and Roger Crawfis. *Rendering Implicit Flow Volumes.* In *proceedings of IEEE Visualization 2004*, pages 99–106, Washington, DC, USA. IEEE Computer Society. 2004.

[ZC02] Caixia Zhang and Roger Crawfis. *Volumetric shadows using splatting.* In *proceedings of IEEE Visualization ualization 2002*, pages 85–92, Washington, DC, USA. IEEE Computer Society. 2002.

Index