TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT INFORMATIK
INSTITUT FÜR SOFTWARE- UND MULTIMEDIATECHNIK
PROFESSUR FÜR COMPUTERGRAPHIK UND VISUALISIERUNG
PROF. DR. STEFAN GUMHOLD

Diplomarbeit

zur Erlangung des akademischen Grades
Diplomingenieur für Informationssystemtechnik

Visualisierung von Laser-Plasma-Simulationen

Lukas Zühl
(Geboren am 26. März 1985 in Dresden)

Betreuer:
Prof. Dr. rer. nat. Stefan Gumhold
Dr. Ulrich Schramm

Dresden, 6. März 2011

Aufgabenstellung

Die Abteilung Laser-Teilchenbeschleunigung am Forschungszentrum Dresden-Rossendorf untersucht experimentell und theoretisch die Wechselwirkung hochintensiver, ultrakurzer Laserimpulse mit Materie. Damit diese Wechselwirkungen in hoher zeitlicher Auflösung analysiert werden können, werden neben den Experimenten Simulationen auf Hochleistungsrechnern durchgeführt. Hierbei fallen große Datenmengen an, die meist zeitaufwendig aufbereitet und visualisiert werden, um die für den Anwender relevanten physikalischen Sachverhalte darzustellen.

Ziel dieser Diplomarbeit ist die Entwicklung eines Programms, das über eine intuitiv bedienbare graphische Benutzeroberfläche die Durchführung der Simulation sowie die Auswertung der Ergebnisse unterstützt. Insbesondere sollen dabei Methoden zur interaktiven Visualisierung von Partikeln (z. B. Elektronen, Ionen) und Feldern (z. B. elektromagnetische Felder) in zwei und drei Dimensionen implementiert werden. Die Daten hierfür können von verschiedenen Simulationscodes stammen und in unterschiedlichen Formaten vorliegen. Folgende Teilaufgaben sind zu bearbeiten:

- Analyse der vorhandenen Simulationen, Datenformate, und Arbeitsschritte:
  - Arbeitsschritte: Simulationsvorbereitung, Simulationsdurchführung, Auswertung der Ergebnisse
  - Simulationscodes: Picls, Illumination, PiConGPU
  - Daten: Partikel, Skalar- und Vektorfelder (2D und 3D)
  - Lokalität: lokal oder verteilt
- Entwicklung eines Programms, das die Vorbereitung und Durchführung der Simulationen über eine graphische Benutzeroberfläche unterstützt und die Ergebnisse der Simulationen visualisiert.
  - Implementierung von Schnittstellen zu den Ein- und Ausgabedaten. Diese sollen große Datenmengen handhaben können und für eine Online-Visualisierung erweiterbar sein.
  - Entwurf und Implementierung einer graphischen Benutzerschnittstelle (GUI).
  - Implementierung verschiedener Visualisierungsnetzwerke für die interaktive Visualisierung von Partikeln sowie Skalar- und Vektorfeldern.
  - Integration vorhandener Arbeiten zur stereoskopischen Visualisierung und 3D-Eingabegeräten.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Elektrotechnik und Informationstechnik eingereichte Arbeit zum Thema:

Visualisierung von Laser-Plasma-Simulationen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 6. März 2011

Lukas Zühl

Kurzfassung

Die Abteilung Laser-Teilchenbeschleunigung des Helmholtz-Zentrum Dresden-Rossendorf (HZDR) erforscht die Wechselwirkung hochintensiver, ultrakurzer Laserimpulse mit Materie. Numerische Simulationen dieses Prozesses stellen ein wichtiges Hilfsmittel bei der Analyse der physikalischen Prozesse in hoher räumlicher und zeitlicher Auflösung bereit. Diese Simulationen produzieren mehrere umfangreiche Datensätze, die für eine spätere Analyse und Visualisierung gespeichert werden. Allerdings liegt der Schwerpunkt existierender Visualisierungsbibliotheken und Werkzeuge auf der Visualisierung einzelner Datensätze. Im Rahmen dieser Arbeit wird ein Framework entwickelt, das eine vereinheitlichte Sicht auf alle Feld- und Teilchendaten einer Simulation bietet und einfache Methoden für die Abfrage, Filterung, Transformation und Abbildung von Daten in einen geometrischen Raum einer Visualisierung bereitstellt. Eine deklarative Sprache wurde entwickelt, die dem Nutzer ermöglicht mit einfachen Mitteln eine Visualisierung zu konfigurieren und Streu- und Liniendiagramme sowie Skalar- und Vektorfeldvisualisierungen zu erzeugen.

Abstract

The laser particle acceleration division at the Helmholtz-Zentrum Dresden-Rossendorf (HZDR) studies the interaction of ultra short, high intensity laser pulses with matter. Numerical simulations of this process are an important method available to analyse the physical processes at a high spacial and temporal resolution. These simulations produce several large datasets that are stored for subsequent analysis and visualization. However, the existing visualization libraries and tools usually focus on visualizing a single dataset at a time. This thesis presents a framework that provides a unified view on all the field and particle datasets produced by a simulation run and simple methods for querying, filtering, transforming, and mapping data to geometric space for visualization. A declarative language was developed to allow the user to easily configure a visualization and create scatterplots and lineplots as well as visualizations of scalar- and vectorfields.

Inhaltsverzeichnis

1 Einleitung

Teilchenbeschleuniger stellen ein wichtiges Werkzeug der Wissenschaft dar [Taj08]. Ihre Anwendung reicht von der Erforschung der Materie bis zu Anwendungen in der Medizin, in der sie z.B. als Röntgen- oder Ionenstrahlquellen für die Strahlentherapie dienen. Das Gebiet der Laser-Teilchenbeschleunigung erforscht, wie Elektronen und Ionen mit Hilfe von Laserimpulsen auf nahezu Lichtgeschwindigkeit beschleunigt werden können. Ziel dieser Forschung ist, kompakte und kostengünstige Teilchenbeschleuniger für die Wissenschaft und Medizin zu entwickeln.

Numerische Simulationen helfen bei der Analyse des Beschleunigungsprozesses und ermöglichen Parameterstudien für seine Optimierung. Da diese Simulationen sehr rechenaufwändig sind, werden sie meist auf Hochleistungsrechnern durchgeführt. Die dabei produzierten Datensätze sind sehr umfangreich und enthalten Teilchen- und Felddaten über viele Zeitschritte. Bestehende Visualisierungswerkzeuge ermöglichen meist nur die Visualisierung einzelner Datensätze, z.B. eines 3D Vektorfeldes zu einem bestimmten Zeitpunkt, und haben keine Kenntnisse über den Aufbau und die Zusammenhänge aller Daten einer Simulation. Eine flexible Visualisierung der Simulationsdaten ist somit nicht möglich oder mit einem hohen Aufwand für die Auswahl und Vorverarbeitung der Datensätze sowie der Konfiguration einer Visualisierung durch den Nutzer verbunden.

Ziel dieser Arbeit ist es, dem Nutzer eine einfache und flexible Visualisierung der gesamten Simulationsdaten zu ermöglichen.

Der nächste Abschnitt gibt zunächst einen Überblick über die Laser-Teilchenbeschleunigung. Danach wird der Particle-in-Cell (PIC)-Algorithmus eingeführt, der in vielen Simulationen der Plasmaphysik verwendet wird. Abschnitt 1.3 stellt existierende Anwendungen und Frameworks vor, die für die Visualisierung von Simulationsdaten geeignet sind. Abschnitt 1.4 geht schließlich auf die Ziele dieser Arbeit näher ein.

1.1 Laser-Teilchenbeschleunigung

Während konventionelle Teilchenbeschleuniger elektromagnetische Wellen mit Wellenlängen im Zentimeterbereich verwenden, um geladenen Teilchen zu beschleunigen, nutzen Laser-Teilchenbeschleuniger

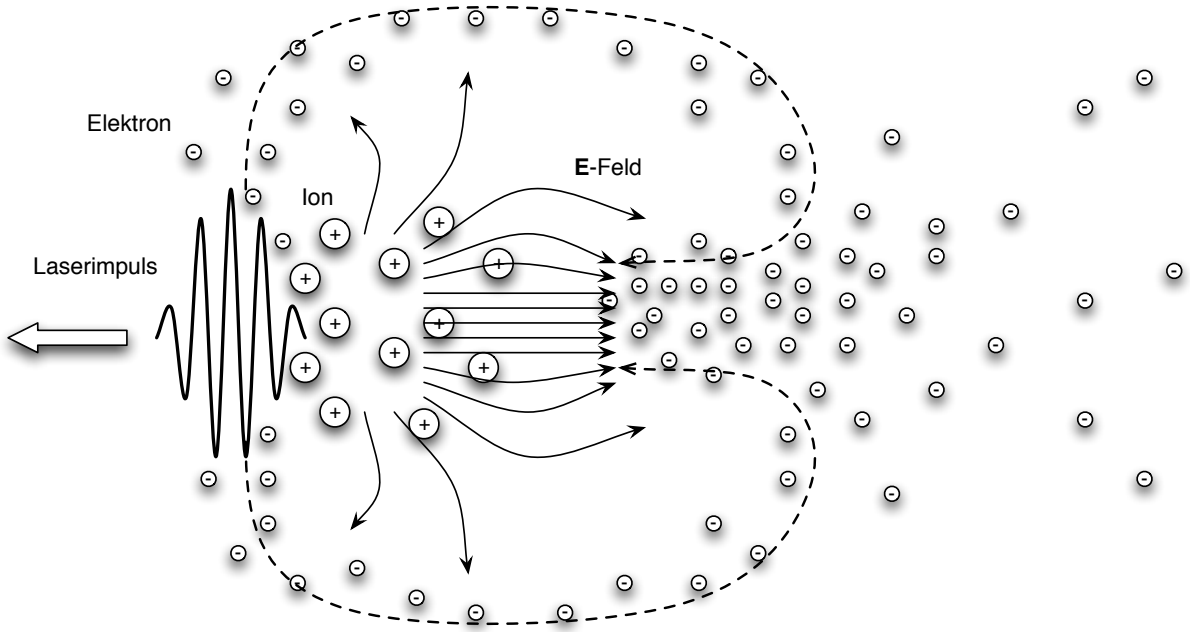

Abbildung 1.1: Prinzip der Laser-Teilchenbeschleunigung nach der LWFA Methode

elektromagnetische Wellen im Mikrometerbereich und darunter. Dies ermöglicht die Konstruktion von kompakteren und günstigeren Beschleunigeranlagen. Allerdings erfordert dies eine hochpräzise Kontrolle des Beschleunigungsprozesses und die Optimierung diverser Laser- und Plasmaparameter, wie der Impulsdauer oder der Plasmadichte [Taj08].

Die Laser-Teilchenbeschleunigung nutzt die sogenannte Laser Wakefield Acceleration (LWFA) Technik, um Elektronen auf nahezu Lichtgeschwindigkeit zu beschleunigen. Abbildung 1.1 veranschaulicht das Prinzip der LWFA. Es wird ein ultrakurzer, hochintensiver Laserimpuls in ein Gasplasma geschossen. In [Taj08] beträgt die Impulsdauer zum Beispiel $35\mathrm{fs}$ bei einer Leistung von bis zu $50\mathrm{TW}$. Die hohen elektrischen Felder des Laserimpulses trennen die negativ geladenen Elektronen von den positiv geladenen Ionen. Hinter dem Laserimpuls entsteht ein sogenanntes *bubble regime*, in dem sehr hohe elektrische Feldstärken erreicht werden. Dieses *wakefield* beschleunigt schließlich die Elektronen auf einer Strecke von wenigen Millimetern auf nahezu Lichtgeschwindigkeit. Dabei werden Energien von mehreren Gigaelektronenvolt erreicht.

1.2 Simulation: Particle-in-Cell Algorithmus

Die Prozesse der LWFA lassen sich mit Simulationscodes der Plasmaphysik numerisch simulieren. Ein weit verbreiteter Algorithmus aus diesem Bereich ist der Particle-in-Cell (PIC)-Algorithmus. Er dis-

kretisiert den simulierten Bereich durch ein 2D oder 3D Gitter, an dessen Knotenpunkten die Werte der elektrischen, magnetischen und Stromdichtefelder berechnet werden. Den Gitterzellen werden außerdem Makropartikel zugewiesen, die die Verteilung einer gewissen Anzahl an Elektronen oder Ionen modellieren. Während die Felder auf einem raumfesten Gitter diskretisiert sind, können sich die Teilchen frei im Raum bewegen und beliebige Positionen einnehmen. Aus dieser Teilchenbewegung lassen sich Ströme berechnen, die wiederum die Felder ändern. Der PIC-Algorithmus basiert auf den Maxwell-Gleichungen [Stö05]:

$$\begin{aligned}
\operatorname{div} \mathbf{B} &= 0 \\
\operatorname{div} \mathbf{D} &= \rho \\
\operatorname{rot} \mathbf{E} &= -\frac{\partial \mathbf{B}}{\partial t} \\
\operatorname{rot} \mathbf{H} &= \frac{\partial \mathbf{D}}{\partial t} + \mathbf{J}
\end{aligned}$$

Im Vakuum gilt außerdem:

$$\begin{aligned}
\mathbf{D} &= \epsilon_0 \cdot \mathbf{E} \\
\mathbf{B} &= \mu_0 \cdot \mathbf{H}.
\end{aligned}$$

Die Maxwell-Gleichungen beschreiben, dass ein Magnetfeld quellenfrei ist, dass Raumladungen die Quellen elektrischer Felder sind, dass zeitveränderliche Magnetfelder elektrische Wirbelfelder erzeugen, und dass Ströme und Verschiebeströme magnetische Wirbelfelder erzeugen. Die Wirkung der elektrischen und magnetischen Felder auf bewegte Ladungen wird durch die Lorentz-Kraft beschrieben:

$$\mathbf{F} = q \cdot (\mathbf{E} + \mathbf{v} \times \mathbf{B}).$$

Eine Iteration des PIC-Algorithmus umfasst folgende Schritte [BWH+10]:

1. Berechnung der Lorentzkraft $\mathbf{F}$, die, durch die elektrische Feldstärke $\mathbf{E}$ und magnetische Flussdichte $\mathbf{B}$, auf die Partikel ausgeübt wird.
2. Berechnung der neuen Position und Geschwindigkeit $\mathbf{v}$ der Partikel.
3. Berechnung der Stromdichte $\mathbf{J}$ aus der Bewegung der Partikel.
4. Berechnung der neuen $\mathbf{E}$ und $\mathbf{B}$ Feldstärken anhand der Maxwell-Gleichungen.

Zwei Simulationscodes, die diesen Algorithmus implementieren sind Illumination [GSM06] und PIConGPU [BWH+10]. Die Visualisierungsbeispiele in dieser Arbeit basieren auf Datensätzen einer Illumination-Simulation. Illumination kann im Gegensatz zu PIConGPU auch dreidimensionale Volumina simulieren, und somit für diese Arbeit geeignete 3D Datensätze bereitstellen, z.B. dreidimensionale Skalar- und Vektorfelder.

1.3 Vorhandene Visualisierungen

Der Prozess der Visualisierung wandelt Daten in Bilder um, die genaue Informationen über diese Daten vermitteln. Für die Erstellung von Visualisierungen existieren verschiedene Ansätze unterschiedlicher Flexibilität und Komplexität [SML06].

- *Visualisierungsanwendungen*: Sie bieten spezialisierte graphische Nutzerschnittstellen und sind meist auf eine bestimmte Anwendungsdomäne spezialisiert.
- *Visuelle Programmierung*: Der Nutzer konstruiert über graphische Eingaben ein Visualisierungsnetzwerk, das für seine Problemstellung geeignet ist.
- *Programmbibliotheken*: Sie erlauben die flexible Programmierung eigener Visualisierungsanwendungen. Dies setzt allerdings Programmierkenntnisse bei dem Nutzer voraus.

Im Folgenden werden einige Programme und Frameworks für die Visualisierung wissenschaftlicher Datensätze vorgestellt. Sie umfassen das Mathematikprogramm MatLab, die Frameworks VTK und Equalizer sowie das Programm ParaView.

Das Programm MatLab bietet umfangreiche Methoden für das technische und wissenschaftliche Rechnen [Mat11]. Es gibt zudem die Möglichkeit diverse 2D und 3D Diagramme zu erstellen. MatLab ist jedoch nicht auf die interaktive, dreidimensionale Visualisierung großer Datensätze spezialisiert. Für eine solche Aufgabe, wurden spezielle Frameworks entwickelt, mit denen performante, auf die wissenschaftliche Visualisierung spezialisierte Anwendung erstellt werden können. Allerdings erfordern diese Frameworks aufgrund ihrer Komplexität meist einen hohen Einarbeitungsaufwand und gute Programmierkenntnisse.

Das Visualization Toolkit (VTK) ist ein quelloffenes Framework für wissenschaftliche Visualisierung [VTK11b]. Es ermöglicht die Programmierung von Visualisierungsnetzwerken, die bestimmte Eingangsdatensätze in mehreren Stufen transformieren und schließlich darstellen können. VTK stellt verschiedene Datentypen und eine große Anzahl an Algorithmen bereit, die die Datensätze verarbeiten und neue Datensätze erzeugen können. Der Netzwerkansatz ermöglicht zudem die Verteilung der einzelnen Stufen auf mehrere Prozesse und die Parallelisierung vieler Visualisierungsalgorithmen.

Ein weiteres Framework, Equalizer, ist auf das flexible, parallele Rendering für OpenGL Anwendungen spezialisiert [EMP09]. Es dient vor allem als *middleware* für Anwendungen, die stark skalierbar sein müssen und Daten sowohl auf einer einzelnen Grafikkarte, als auch auf einem *cluster* von mehreren Prozessoren und Graphical Processing Units (GPUs) rendern. Im Gegensatz zu VTK bietet Equalizer jedoch keine große Auswahl an vorgefertigten Filtern für die Datentransformation und wissenschaftliche

Visualisierung an, sondern konzentriert sich überwiegend auf den parallelen Rendering-Prozess.

Die Anwendung ParaView [Par11] basiert auf VTK und stellt dem Nutzer eine grafische Benutzeroberfläche bereit, über die Visualisierungen erstellt und konfiguriert werden können. ParaView nutzt intensiv die Möglichkeiten der Parallelverarbeitung von VTK und ermöglicht so die Visualisierung großer Datenmengen.

In [Wil05] wird eine Grammatik spezifiziert, mit der Grafiken sehr flexibel beschrieben und erzeugt werden können. Sie wurde überwiegend für Grafiken aus dem Bereich der Statistik entwickelt. Für die Visualisierung von Simulationsdaten werden allerdings spezielle Verfahren, z.B. für die 3D Vektorfeldvisualisierung, benötigt und es müssen spezielle Datentypen, insbesondere Gittertypen, spezifiziert und effizient verarbeitet werden können. Ausserdem ist die effiziente Verarbeitung und Darstellung großer Datenmengen in der wissenschaftlichen Visualisierung entscheidend.

1.4 Ziele dieser Arbeit

Wie bereits in den vorigen Abschnitten erwähnt wurde, konzentrieren sich existierende Visualisierungslösungen auf die Visualisierung einzelner Datensätze. Die üblichen Anwendungen der wissenschaftlichen Visualisierung setzten Wissen über die Struktur und die Zusammenhänge der Daten voraus. Sie übernehmen meist das feste physikalische Modell der Datensätze direkt für die Visualisierung. Zum Beispiel werden die elektrischen oder magnetischen Felder meist mit den Methoden der Vektorfeldvisualisierung dargestellt und die Partikeldatensätze werden stets als Punktwolken im dreidimensionalen Raum visualisiert. Eine Visualisierung dient dem Wissenschaftler hier nur zur Darstellung bestimmter Ergebnisse oder um gewisse Parameter visuell zu bestimmen.

Ziel dieser Arbeit ist, die Struktur aller Simulationsdaten zunächst unabhängig von einer möglichen Visualisierung zu erfassen und dann die flexible Zuordnung von Datenfeldern zu den Achsen und Attributen einer Visualisierung zu ermöglichen. Dem Wissenschaftler soll so ermöglicht werden, über eine einfache und flexible Konfiguration der Darstellungen, beliebige physikalische Größen zueinander in Beziehung zu setzen, um neue Zusammenhänge entdecken und Modelle bilden zu können.

Für die Visualisierung sollen wenige Visualisierungsnetzwerke mit VTK implementiert werden, die möglichst viele Konfigurationen abdecken. In dieser Arbeit wird VTK verwendet, da es bereits eine große Auswahl an geeigneten Filtern für die Visualisierung der Simulationsdaten bietet. Dazu gehören unter anderem Filter für die Erzeugung von Glyphen, Isoflächen und Stromlinien, die sich jeweils für die Darstellung von Partikeln, Skalarfeldern und Vekorfeldern eignen. Desweiteren können eigene Filter implementiert und in einem VTK-Visualisierungsnetzwerk verwendet werden.

Auch der Ansatz aus [Wil05], Graphiken durch eine Grammatik zu beschreiben, findet in dieser Arbeit Verwendung. Über eine deklarative Sprache kann die Konfiguration einer Visualisierung sehr kompakt beschrieben werden. Eine solche Beschreibung muss nicht vom Nutzer selbst geschrieben werden, sondern kann auch durch eine graphische Benutzeroberfläche erzeugt werden. Dies würde auch Client-Server Anwendungen ermöglichen, in denen eine Konfiguration auf dem Client erstellt und über ein Netzwerk an den Server übertragen wird, der daraus eine Visualisierung erstellt.

Kapitel 2 führt die Simulationsdatensätze ein und gibt eine Einführung in VTK, seine Datenstrukturen und Algorithmen. Es werden erste Visualisierungen für die Partikel- und Felddaten vorgestellt. In Kapitel 3 wird eine Verallgemeinerung des Datenmodells und der Visualisierungen entwickelt. Die Ideen dieses Kapitels bilden den Kern der vorliegenden Arbeit. Auf der Basis der hier erstellten Modelle werden in Kapitel 4 neue Visualisierungen und flexible Konfigurationsmöglichkeiten implementiert. Kapitel 5 stellt schließlich einige Ansätze für die Erweiterung und Verbesserung der vorgestellten Implementierung vor.

2 Visualisierung einzelner Datensätze

Simulationen, die auf gängigen PIC-Codes wie Illumination oder PIConGPU basieren, erzeugen typischerweise pro Simulationsschritt einzelne Teilchen- und Felddatensätze. Dieses Kapitel stellt Methoden für die Visualisierung dieser einzelnen Datensätze vor und zeigt wie diese mit dem Visualization Toolkit implementiert werden. Die vollständige Simulation wird in Kapitel 3 behandelt.

Der folgende Abschnitt stellt anhand eines Beispieldatensatzes die gegebenen Daten und ihre Struktur vor. Abschnitt 2.2 gibt eine Einführung in VTK und trifft eine Auswahl geeigneter VTK-Datenstrukturen und Filter, die für die Visualisierung der Simulationsdaten geeignet sind. Die Partikel-, Skalarfeld- und Vektorfeldvisualisierungen in den Abschnitten 2.3 bis 2.5 stellen die Grundlage für die allgemeineren Methoden des nächsten Kapitels dar.

2.1 Beispiel-Datensatz

Dieser Abschnitt stellt einen Datensatz vor, der im weiteren Verlauf dieser Arbeit zur Veranschaulichung verwendet wird. Er stammt von einer Illumination Simulation und enthält Dateien zweier Formate, je nachdem, ob es sich um Teilchendaten, z.B. Elektronen und Ionen, oder Felddaten, z.B. elektrische und magnetische Felder, handelt.

Die Teilchendaten sind in Textdateien in der American Standard Code for Information Interchange (ASCII)-Kodierung gespeichert. Illumination erzeugt eine dieser Dateien für jeden Zeitschritt der Simulation, oder für eine Teilmenge der Zeitschritte, um die resultierende Datenmenge zu reduzieren. Jede dieser Dateien enthält eine Liste von Teilchen und deren Attribute, wobei jede Zeile ein Teilchen repräsentiert und die Attribute “`Position.X`”, “`Position.Y`”, “`Position.Z`”, “`Velocity.X`”, “`Velocity.Y`”, “`Velocity.Z`”, “`Energy`” und “`ID`” jeweils einer Spalte zugeordnet sind. Tabelle 2.1 gibt einen Überblick über die Attribute, deren Datentypen und Einheiten. Die Attributnamen sind in den Dateien selbst nicht spezifiziert, sondern so gewählt um innerhalb dieser Arbeit und in dem Programm die Spalten referenzieren zu können, siehe Abschnitt 3.1. Die “`ID`” wird benötigt um ein bestimmtes Teilchen identifizieren und im zeitlichen Verlauf verfolgen zu können, weil ihre Anzahl mit der Zeit variieren kann und deshalb ein Teilchen nicht implizit über die Zeilennummer identifiziert werden

Tabelle 2.1: Partikelattribute

Bezeichner	Datentyp	Einheit	Beschreibung
"Position.X"	double	µm	X-Position des Teilchens
"Position.Y"	double	µm	Y-Position des Teilchens
"Position.Z"	double	µm	Z-Position des Teilchens
"Velocity.X"	double	$\mathrm{cm} \cdot \mathrm{s}^{-1}$	X-Komponente der Geschwindigkeit des Teilchens
"Velocity.Y"	double	$\mathrm{cm} \cdot \mathrm{s}^{-1}$	Y-Komponente der Geschwindigkeit des Teilchens
"Velocity.Z"	double	$\mathrm{cm} \cdot \mathrm{s}^{-1}$	Z-Komponente der Geschwindigkeit des Teilchens
"Energy"	double	MeV	Energie des Teilchens
"ID"	unsigned	1	Teilchen ID

Tabelle 2.2: Feldattribute

Bezeichner	Datentyp	Einheit	Beschreibung
"E.X"	double	$\mathrm{statV/cm}$	X-Komponente des elektrischen Feldes
"E.Y"	double	$\mathrm{statV/cm}$	Y-Komponente des elektrischen Feldes
"E.Z"	double	$\mathrm{statV/cm}$	Z-Komponente des elektrischen Feldes

kann.

Des weiteren erzeugt Illumination während der Simulation Skalar- und Vektorfelddatensätze in einem Binärformat. Zu beachten ist, dass alle Attribute der Teilchendaten in einer Datei stehen, während jedes Skalarfeld und alle Komponenten der Vektorfelder in separaten Dateien stehen. Beispielsweise werden für das **E**-Feld

$$\mathbf{E} = E_x \cdot \mathbf{e}_x + E_y \cdot \mathbf{e}_y + E_z \cdot \mathbf{e}_z$$

pro Zeitschritt drei Dateien erzeugt, die jeweils die E_x, E_y und E_z Komponenten enthalten. Der Datensatz eines Vektorfeldes setzt sich dementsprechend aus den Datensätzen mehrerer Skalarfelder zusammen, die in der jeweiligen Anwendung zusammengefügt werden müssen. Tabelle 2.2 beschreibt die vorhandenen Feldattribute, ihre Datentypen und Einheiten[1].

In diesem Beispiel führt Illumination eine 3D Simulation durch. Hierbei wird der Raum in einem dreidimensionalen Gitter mit n_x, n_y und n_z Werten entlang der jeweiligen Achsen diskretisiert, an dessen Knotenpunkten die einzelnen Feldwerte berechnet werden. Jeder Feldwert $F(i_x, i_y, i_z)$, mit $0 \leq i_\alpha <$

[1]Hierbei gilt: $1\,\mathrm{statV/cm} = \mathrm{c}_0 \cdot 10^{-6} \cdot \mathrm{V/m}$, $\mathrm{c}_0 = 299\,792\,458\,\mathrm{m} \cdot \mathrm{s}^{-1}$.

$n_\alpha - 1, \alpha \in \{x, y, z\}$, wird in einem eindimensionalen Array der Größe

$$n = n_x \cdot n_y \cdot n_z$$

an der Position

$$i = i_x \cdot n_y \cdot n_z + i_y \cdot n_z + i_z$$

gespeichert. Illumination erstellt für alle Zeitschritte i_t der Simulation und jedes Skalarfeld bzw. jede Vektorfeldkomponente eine neue Datei, in die das entsprechende Array in binärer Kodierung geschrieben wird.

Aus den Indizes i_x, i_y, i_z und i_t lassen sich die Raumkoordinaten x, y und z sowie die Zeit t wie folgt berechnen:

$$x = o_x + s_x \cdot i_x$$

$$y = o_y + s_y \cdot i_y$$

$$z = o_z + s_z \cdot i_z$$

$$t = o_t + s_t \cdot i_t,$$

wobei o_x, o_y und o_z den Gitter-Ursprung darstellt und s_x, s_y und s_z den Gitterabstand (*origin* und *spacing*) in x-, y- und z-Richtung, siehe Abbildung 2.1. Analog dazu, geben die Werte o_t und s_t die verwendete Startzeit und Abtastrate der Simulation an. In diesem Beispiel haben sie folgende Werte:

$$o_x = 0\,\mu\text{m}, \quad o_y = 0\,\mu\text{m}, \quad o_z = 0\,\mu\text{m}, \quad o_t = 0\,\text{fs}$$

und

$$s_x = 0{,}181\,486\,310\,3\,\mu\text{m}$$

$$s_y = 0{,}181\,486\,310\,3\,\mu\text{m}$$

$$s_z = 0{,}045\,371\,577\,57\,\mu\text{m}$$

$$s_t = 0{,}100\,895\,528\,1\,\text{fs}.$$

Diese Werte werden bei der Konfiguration einer Simulation festgelegt.

Desweiteren existiert in Illumination die Option das Simulationsfenster ab einem bestimmten Zeitschritt $i_{t,\text{start}}$ mit dem Laserimpuls zu bewegen, um diesen über eine längere Zeit verfolgen zu können. Dadurch

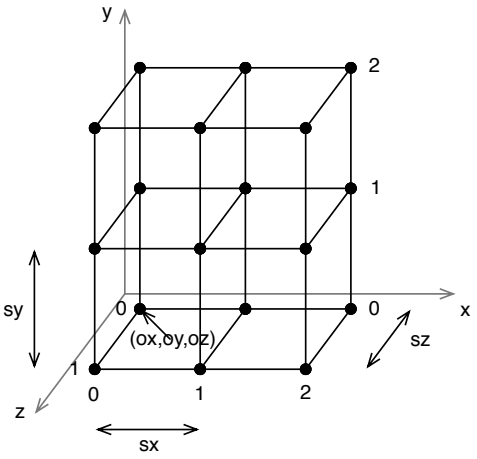

Abbildung 2.1: Struktur eines gleichmäßigen 3D Gitters

wird die z-Komponente des Gitterursprungs abhängig von dem Zeitschritt i_t:

$$o_z(i_t) = \begin{cases} 0\,\mu\text{m} & \text{für } i_t \leq i_{t,\text{start}}, \\ (i_t - i_{t,\text{start}}) \cdot z_{\text{out}} & \text{für } i_t > i_{t,\text{start}}. \end{cases}$$

$$i_{t,\text{start}} = 290$$

$$z_{\text{out}} = 0{,}483\,963\,\mu\text{m}$$

wobei z_{out} der Propagationsabstand zwischen zwei Ausgaben ist. Die Abhängigkeit der z-Position von zwei Indizes i_z und i_t ab einem bestimmten Zeitpunkt muss bei der Konfiguration der Visualisierungen beachtet werden.

2.2 VTK

Das Visualization Toolkit (VTK) ist ein frei verfügbares Softwaresystem für 3D Computergraphik, Bildverarbeitung und wissenschaftliche Visualisierung [VTK11b]. Hierbei handelt es sich um eine umfangreiche C++ Bibliothek, deren Klassen als Bausteine für komplexe Visualisierungsanwendungen dienen [SML96]. Schnittstellen zu anderen Programmiersprachen wie Tcl/Tk, Java, und Python sind ebenfalls vorhanden, werden aber in dieser Arbeit nicht benötigt.

Der Prozess der Visualisierung mit VTK umfasst die Transformation von Daten und deren Abbildung auf Graphikprimitive, die mit den Mitteln der Computergrafik dargestellt werden können. Hierfür werden zwei Modelle eingeführt: das Visualisierungsmodell und das Graphikmodell.

Das Visualisierungsmodell beschreibt den Datenfluss einer Visualisierung. Basierend auf dem *Pipes-*

and-Filters Architekturmuster [SH09], implementiert VTK dieses Modell als ein Netzwerk von Filtern, die jeweils ihre Eingangsdatensätze transformieren, um neue Ausgangsdatensätze zu erzeugen [VTK04]. Hierauf wird in den nächsten Abschnitten detaillierter eingegangen.

Das Graphikmodell abstrahiert Objekte der 3D Computergraphik, damit die Visualisierungsanwendung unabhängig von der zugrundeliegenden Graphikbibliothek und dem Fenstersystem programmiert werden kann. Es umfasst u.a. folgende Klassen:

- `vtkMapper`: enthält die geometrische Definition eines oder mehrerer 3D Objekte.
- `vtkActor`: repräsentiert ein Object der 3D Szene.
- `vtkProperty`: enthält Attribute eines `vtkActor`s, z.B. die Farbe oder Textur.
- `vtkTransform`: 4×4 Transformationsmatrix eines `vtkActor`s.
- `vtkLight`: beleuchtet die 3D Szene.
- `vtkCamera`: positioniert den Beobachter in der 3D Szene.
- `vtkRenderer`: rendert die 3D Szene.
- `vtkRenderWindow`: stellt das Ergebnis dar.
- `vtkRenderWindowInteractor`: ermöglicht die interaktive Steuerung der Kamera durch den Nutzer.

Die Klasse `vtkMapper` stellt das Bindeglied zwischen dem Visualisierungsmodell und dem Graphikmodell dar. Eine Instanz dieser Klasse, bzw. einer ihrer Unterklassen, bildet einen (transformierten) Datensatz auf die Graphikprimitive der zugrundeliegenden Graphikbibliothek ab. Dies können beispielsweise Punkte, Linien und Dreiecke in OpenGL [Ope11] sein.

2.2.1 Datentypen

Während der Visualisierung tauschen die Filter verschiedene Datenobjekte untereinander aus: Einerseits Metadaten, die Informationen über die Datensätze, Filter und Abfragen enthalten und als Schlüssel-Wert-Paare in Instanzen der Klasse `vtkInformation` gespeichert sind; andererseits Instanzen der Klasse `vtkDataObject` oder einer Spezialisierung davon, die die eigentlichen, zu visualisierenden Daten enthalten. Intern verwendet `vtkDataObject` eine Instanz von `vtkFieldData` um die Daten zu speichern. Hierbei handelt es sich um eine Tabellenstruktur, in der verschiedene Datenfelder eines Datensatzes spaltenweise organisiert sind und deren Werte in den einzelnen Zeilen gespeichert werden. Die Spalten sind als *arrays* implementiert und können durch einen Namen identifiziert werden. Ihre Elemente sind n-Tupel und können sowohl skalare als auch vektorielle Werte speichern, wobei n für ein

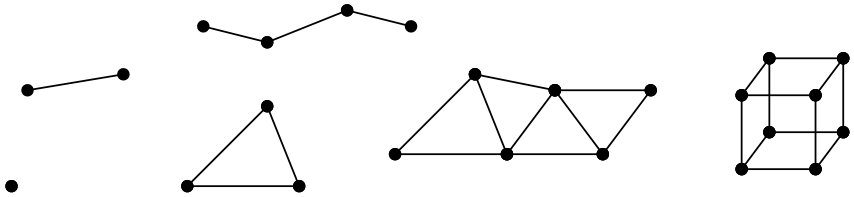

Abbildung 2.2: Struktur einiger Zelltypen aus VTK: *vertex, line, polyline, triangle, triangle strip, voxel*

gegebenes Array konstant ist. In diese Arrays müssen letztendlich die Simulationsdaten, die visualisiert werden sollen, geschrieben werden.

Die Klasse `vtkDataSet` und ihre Unterklassen erweitern `vtkDataObject` um Struktur- und Attributinformationen. Die Struktur umfasst Topologie und Geometrie, implementiert als Zell- und 3D Punktdaten. Zellen stellen eine topologische Beziehung zwischen den Punkten eines Datensatzes her. Sie ermöglichen u.a. die lokale Interpolation der Daten und die Berechnung von Gradienten. In dieser Arbeit sind zur Darstellung der Teilchen, Trajektorien und Skalar- sowie Vektorfelder insbesondere die (Poly-)Vertex-, (Poly-)Line- und Voxel-Zellen relevant. Abbildung 2.2 veranschaulicht die Voxel- und einige polygonale Zellen.

Zellen und Punkte müssen nicht immer explizit gespeichert werden. Wenn ein Datensatz eine reguläre Struktur hat, besteht eine einfache mathematische Beziehung zwischen den Punkten und Zellen. Dies ermöglicht effizientere Speicherstrukturen und Algorithmen als die allgemeineren, irregulären Datensätze es zulassen. VTK bietet u.a. folgende Spezialisierungen von `vtkDataSet`:

- `vtkImageData`: hat eine reguläre Topologie und Geometrie. Es besteht aus $n_x \cdot n_y \cdot n_z$ Punkten mit gleichmäßigem Abstand und $(n_x - 1) \cdot (n_y - 1) \cdot (n_z - 1)$ Voxel-Zellen.
- `vtkRectilinearGrid`: hat eine reguläre Topologie und teilweise reguläre Geometrie. Die Punktpositionen $p(i_x, i_y, i_z)$ werden aus drei Arrays $x[i_x]$, $y[i_y]$ und $z[i_z]$ bestimmt.
- `vtkStructuredGrid`: hat eine reguläre Topologie und irreguläre Geometrie. Die Punktpositionen werden in einem Array $p[i_x, i_y, i_z]$ explizit angegeben.
- `vtkUnstructuredGrid`: hat eine irreguläre Topologie und Geometrie. Wird aus beliebigen Zelltypen und Punkten erzeugt.
- `vtkPolyData`: stellt polygonale Daten (*vertices, lines, polygons, triangle strips*) bereit, wie sie z.B. auch in Graphikbibliotheken wie OpenGL vorkommen [OSW+08].
- `vtkTemporalDataSet`: ist eine Unterklasse von `vtkCompositeDataSet` die mehrere Zeitschritte eines Datensatzes speichert.

Relevant für diese Arbeit sind `vtkImageData` zur Speicherung der Gitterdaten der Simulation, z.B. der $\mathbf{E}$- und $\mathbf{B}$-Felder, sowie `vtkPolyData` für die Partikeldaten. Obwohl während der Simulation Daten für viele Zeitschritte erzeugt werden, wird `vtkTemporalData` in dieser Arbeit nicht verwendet. Abschnitt 3.1 geht hierauf im Detail ein.

VTK-Datensätze enthalten außerdem Attribute, die sowohl den Punkten als auch den Zellen zugeordnet werden können. Die Attribute umfassen u.a. Skalarwerte, Vektorwerte, Normalen, Texturkoordinaten, Tensoren und IDs. Bei der Partikelvisualisierung kann z.B. die Energie eines Teilchens als Skalarattribut und die Geschwindigkeit als Vektorattribut gespeichert werden. Die Gitterdatensätze speichern meist nur die Attribute der Gitterknoten, da die Punkt- und Zelldaten implizit aus dem gleichmäßigen Gitter hervorgehen.

2.2.2 Filter

Ein Visualisierungsnetzwerk in VTK ist ein gerichteter, azyklischer[2] Graph aus untereinander verbundenen Filtern. Diese Filter erzeugen oder transformieren die Daten der oben vorgestellten Datensätze. Quellen (*source objects*) sind Filter mit einem oder mehreren Ausgängen, aber keinen Eingängen. Sie erzeugen Datensätze prozedural oder lesen sie von einer externen Quelle ein, z.B. aus einer Datei. Senken (*sinks, mappers*) sind Filter mit einem oder mehreren Eingängen aber keinem Ausgang. Sie wandeln die Ergebnisdaten in Graphikprimitive um (`vtkMapper`) oder schreiben sie z.B. in eine Datei.

Da es verschiedene Strategien gibt ein solches Netzwerk auszuführen, verwendet VTK das *strategy*-Entwurfsmuster [GHJV94], um die Strategie unabhängig von dem Algorithmus eines Filters austauschen zu können. Der Algorithmus eines Filters wird in eine Unterklasse von `vtkAlgorithm` implementiert und die Ausführungsstrategie in einer Unterklasse von `vtkExecutive`. Folgende Unterklassen sind verfügbar [VTK11a]:

- `vtkExecutive`
- `vtkDemandDrivenPipeline`
- `vtkStreamingDemandDrivenPipeline`
- `vtkCompositeDataPipeline`.

Ob sich ein vorangehender Filter aktualisiert hat und die Pipeline erneut ausgeführt werden muss, wird anhand eines Zeitstempels ermittelt (*modified time*). In einem ersten Schritt, *update pass*, wird ermittelt, welche Filter aktualisiert werden müssen, und in einem weiteren Schritt, *request pass*, werden die entsprechenden Berechnungen und der Datenaustausch durchgeführt. Die `vtkStreamingDemand-`

[2] ab VTK Version 5.0 sind keine Zyklen mehr zulässig.

`DrivenPipeline` ermöglicht das *streaming* von großen Datenmengen durch das Netzwerk [ALS+00] indem immer nur Teilmengen eines Datensatzes angefordert und verarbeitet werden (*extends, pieces, time steps*). Die `vtkCompositeDataPipeline` verarbeitet iterativ mehrere Blöcke oder Zeitschritte eines `vtkCompositeDataSet`s.

VTK bietet eine Vielzahl von Algorithmen, die Datensätze unterschiedlichen Typs transformieren und neue Datensätze erstellen können. Die Basisklasse aller Algorithmen ist `vtkAlgorithm`. Sie stellt eine allgemeine Schnittstelle für die Verarbeitung von Datensätzen in einem Visualisierungsnetzwerk bereit. Spezialisierte Klassen sind für unterschiedliche Datentypen verfügbar. Hierbei ist der Typ des zurückgegebenen Datensatzes entscheidend für die Benennung der entsprechenden Basisklasse. Beispielsweise verarbeitet der Algorithmus `vtkMarchingCubes` Volumendatensätze, z.B. `vtkImageData`, erbt aber von `vtkPolyDataAlgorithm`, weil er Isoflächen als polygonalen Datensatz erzeugt. Im Rahmen dieser Arbeit werden überwiegend Algorithmen der Klassen `vtkDataObjectAlgorithm`, `vtkDataSetAlgorithm`, `vtkPolyDataAlgorithm` und `vtkImageDataAlgorithm` verwendet.

2.3 Partikelvisualisierung

Um die Teilchendaten aus Abschnitt 2.1 zu visualisieren, müssen diese zunächst in entsprechende VTK-Datenstrukturen überführt werden. Anhand dieser Daten kann ein Visualisierungsnetzwerk eine 3D Darstellung erzeugen.

Da die Simulationsdaten nicht in einem VTK-Dateiformat vorliegen, muss ein neuer Quellfilter implementiert werden, der die Daten lädt und den nachfolgenden Filtern in geeigneten VTK-Datenstrukturen bereitstellt. Die Positionen der Teilchen sind explizit gegeben – es handelt sich um unstrukturierte Punktdaten die sich als *vertices* eines polygonalen Datensatzes speichern lassen. Die Werte von "`Position.X`", "`Position.Y`" und "`Position.Z`" werden den drei Komponenten von `vtkPointData` zugeordnet. Da es sich bei diesem Datensatz ausschließlich um Punktdaten handelt und keine Linien, Polygone oder Dreiecksstreifen erzeugt werden, müssen die *vertex*-Zellen nicht explizit erzeugt werden. Es wird implizit angenommen, dass jeder Eintrag in dem Punkte-Array ein Vertex darstellt.

Neben Positionsdaten sind weitere Attribute der Teilchen gegeben. Ein Array von Dreitupeln speichert die Werte von "`Velocity.X`", "`Velocity.Y`" und "`Velocity.Z`" und wird dem *vectors*-Attribut zugeordnet. Die "`Energy`"-Werte werden in ein weiteres Array geladen und dem *scalars*-Attribut zugeordnet.

Der Ladefilter ist eine Spezialisierung von `vtkPolyDataAlgorithm` und bildet die Quelle des Vi-

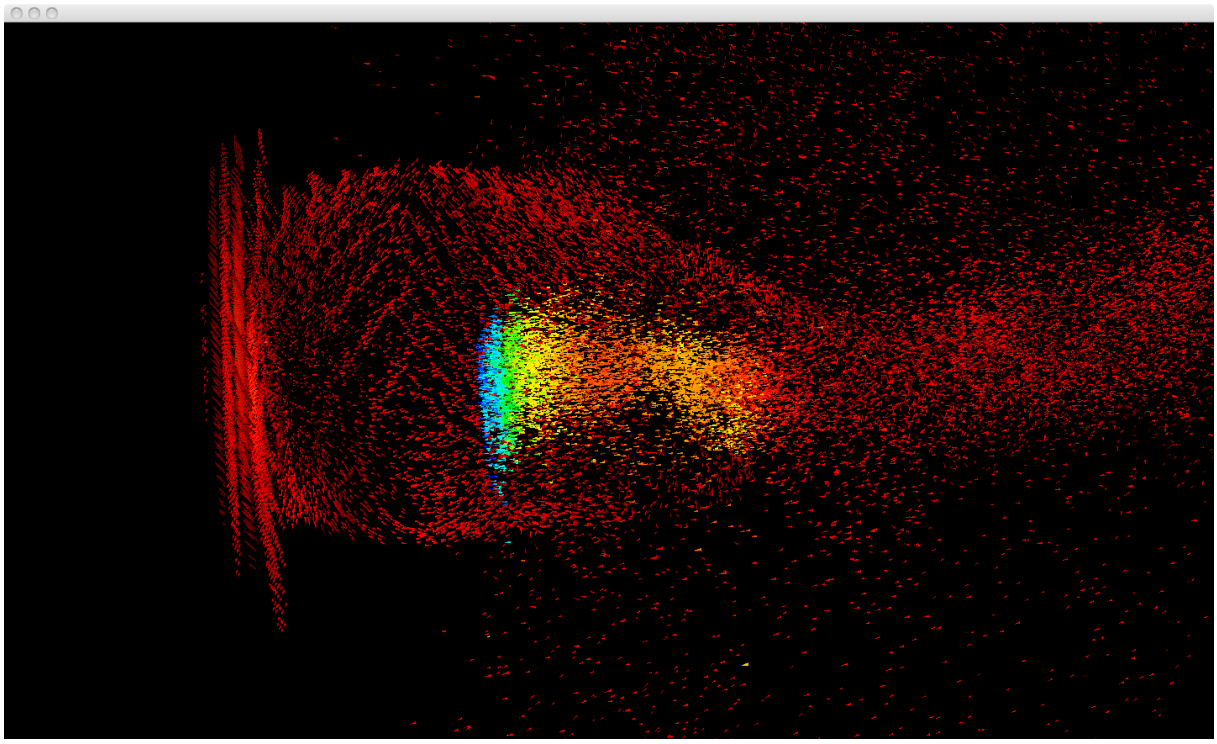

Abbildung 2.3: Die Visualisierung der Partikeldaten mit gerichteten Glyphen

sualisierungsnetzwerks. Der `vtkPolyData` Datensatz, den er erzeugt, kann von weiteren Filtern verarbeitet werden. Da die Teilchenanzahl in jeder Datei sehr hoch sein kann (ca. 900 000 Teilchen), ist eine Reduktion des Datensatzes wünschenswert, sowohl um Speicherplatz und Rechenzeit zu verringern, als auch um die Übersichtlichkeit zu erhöhen. Hierfür bietet VTK die Filter `vtkMaskPoints` und `vtkThresholdPoints`. Mit `SetOnRatio(`n`)` reicht `vtkMaskPoints` nur jeden n-ten Punkt des Eingangsdatensatzes an den Ausgangsdatensatz weiter. Die Option `SetRandomModeOn()` stellt sicher, dass die Auswahl zufällig ist und im Erwartungswert jeder n-te Punkt angezeigt wird. Bei dem Filter `vtkThresholdPoints` kann sowohl eine untere als auch eine obere Grenze für die Skalarwerte (hier "`Energy`") angegeben werden. Punkte, deren Skalarattribut sich außerhalb dieser Grenzen befinden, werden verworfen.

Abbildung 2.3 stellt den Teilchendatensatz des 100. Simulationsschritts dar. Die Teilchen werden durch Glyphen repräsentiert und nach der Energie (Skalarattribut) eingefärbt. Rote Glyphen repräsentieren die niedrigste Energie und blaue die höchste. Die Glyphen lassen sich mit dem `vtkGlyph3D`-Filter erzeugen, der an jeden Punkt eines Eingangsdatensatzes ein gegebenes Objekt (hier ein `vtkConeSource` mit der Seitenanzahl 3) erzeugt und anhand des Vektorattributs ausrichtet. Dieser Filter erzeugt einen polygonalen Datensatz, der mit dem `vtkPolyDataMapper` auf Graphikprimitive umgesetzt wird. Abbildung 2.4 stellt das Visualisierungsmodell dieses Netzwerkes dar. Abschnitt 4.1.2 verallgemeinert

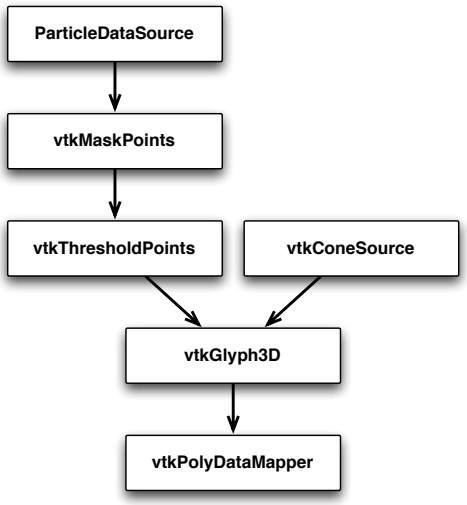

Abbildung 2.4: Das Visualisierungsnetzwerk zur Darstellung der Partikel

diese Darstellung auf Punktdaten eines n-dimensionalen Raumes.

2.4 Skalarfeldvisualisierung

Das gleichmäßige Gitter auf dem die skalaren Feldwerte, z.B. "`E.X`", gespeichert werden, entspricht exakt dem Datenformat von `vtkImageData`. Es handelt sich um ein 3D Array bei dem die Positionsdaten implizit aus den Array-Indizes i, j, k und den *origin* und *spacing* Werten berechnet werden. Eine Klasse, die von `vtkImageAlgorithm` abgeleitet ist, lädt die Daten in das *scalars*-Array von `vtkImageData` und ist die Quelle des Visualisierungsnetzwerks.

Es gibt mehrere Möglichkeiten ein solches 3D Skalarfeld zu visualisieren, u.a. Isoflächen, 2D Schnitte, und Volumenrendering [SML06]. Im Rahmen dieser Arbeit werden Isoflächen für die Skalarfeldvisualisierungen genutzt. Eine Isofläche eines dreidimensionalen Skalarfeldes $F : \mathbb{R}^3 \rightarrow \mathbb{R}$ besteht aus allen Punkten $p \in \mathbb{R}^3$ dieses Feldes, für die gilt: $F(p) = c$, mit c konstant (Isowert). Der Filter `vtkImageMarchingCubes` verwendet den *marching cubes*-Algorithmus [LC87], um aus dem `vtkImageData` Eingansdatensatz diese Flächen zu approximieren und als `vtkPolyData` bereitzustellen. Die Flächennormalen, die für die Beleuchtungsberechnung nötig sind, berechnet der Filter `vtkPolyDataNormals`. Der `vtkPolyDataMapper` bildet das Ergebnis auf entsprechende Graphikprimitive ab. Abbildung 2.5 zeigt das Visualisierungsmodell dieses Netzwerks. Abbildung 2.6 zeigt die Visualisierung des "`E.X`"-Feldes aus dem gegebenen Datensatz aus Abschnitt 2.1 zum Simulationsschritt 100 an.

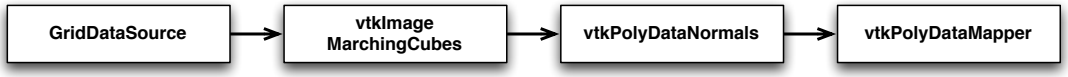

Abbildung 2.5: Das Visualisierungsnetzwerk zur Darstellung eines Skalarfeldes mit Isoflächen

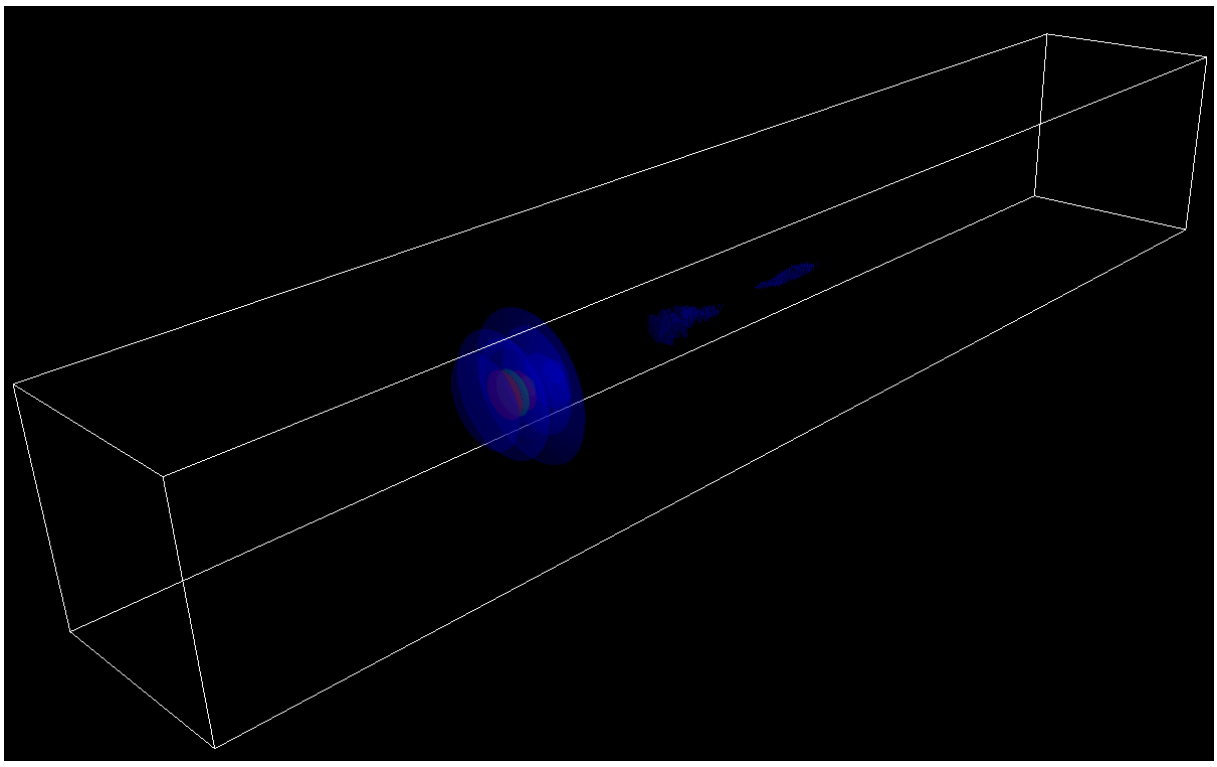

Abbildung 2.6: Die Visualisierung des Skalarfeldes mit Isoflächen

2.5 Vektorfeldvisualisierung

Wie in Abschnitt 2.1 eingeführt wurde, werden Vektorfelder komponentenweise gespeichert. Es kann für jede Feldkomponente die selbe Ladeklasse genutzt werden, die das Skalarfeld des vorigen Abschnitts lädt. Wie in Abbildung 2.7 zu sehen ist, lassen sich drei separate Skalarfelder durch den Filter `vtkImageAppendComponents` zu einem Feld mit drei Komponenten zusammenfügen. Der Filter `vtkAssignAttribute` weist dieses Feld dem *vectors*-Attribut zu.

Die folgenden Abschnitte stellen zwei Methoden vor, um 3D-Vektorfelder zu visualisieren: sogenannte *hedgehogs* und Stromlinien.

2.5.1 Hedgehogs

Hedgehogs sind ein Spezialfall der Glyphendarstellung, die an jedem Punkt des Eingangsdatensatzes eine Linie erzeugt und entlang des Vektorattributs ausrichtet. Die vielen Linien, die so erzeugt werden und wie Stacheln aussehen, geben diesem Verfahren seinen Namen: *hedgehog* (Igel).

Dieses Verfahren liefert gute visuelle Ergebnisse, wenn die Länge der unterschiedlichen Vektorpfeile sich nicht wesentlich unterscheidet und sie sich nicht stark überschneiden. Bei den vorliegenden $\mathbf{E}$-Feldern kann sich der Betrag jedoch um mehrere Größenordnungen unterscheiden, was zu der Darstellung in Abbildung 2.8 führt. Mit dem Filter `vtkArrayCalculator` lassen sich neue Vektorwerte für die Länge der Linien berechen. Außerdem lässt sich ein Skalarwert, z.B. die euklidische Norm des Vektors, hinzufügen, der für die Farbe verwendet werden kann. Somit zeigen nach einer Normierung aller Vektoren, mit

$$\mathbf{v}_{\text{neu}} = \begin{cases} \frac{\mathbf{v}}{\|\mathbf{v}\|} & \text{für } \|\mathbf{v}\| \neq 0 \\ 0 & \text{sonst} \end{cases}$$

die Linien in Richtung des Vektorfeldes, wobei deren Farbe den Betrag des Feldes repräsentiert. Eine weitere visuelle Verbesserung kann erreicht werden, wenn für die Normalisierung eine stetige Funktion verwendet wird, für die außerdem gilt, dass $\|\mathbf{v}_{\text{neu}}\| \to 0$, wenn $\|\mathbf{v}\| \to 0$. Eine nichtlineare Skalierung der Vektoren mit einer monoton steigenden Funktion stellt sicher, dass sehr kleine Feldstärken auch durch kürzere Linien dargestellt werden. Eine mögliche Funktion, die diese Anforderungen erfüllt, ist $\log(1 + \|\mathbf{v}\|)$. Die neuen Vektorwerte ergeben sich aus

$$\mathbf{v}_{\text{neu}} = \begin{cases} \log(1 + \|\mathbf{v}\|) \cdot \frac{\mathbf{v}}{\|\mathbf{v}\|} & \text{für } \|\mathbf{v}\| \neq 0 \\ 0 & \text{sonst} \end{cases}$$

Die Filter `vtkMaskPoints` und `vtkThresholdPoints` geben, wie bei den Partikeln in Abschnitt

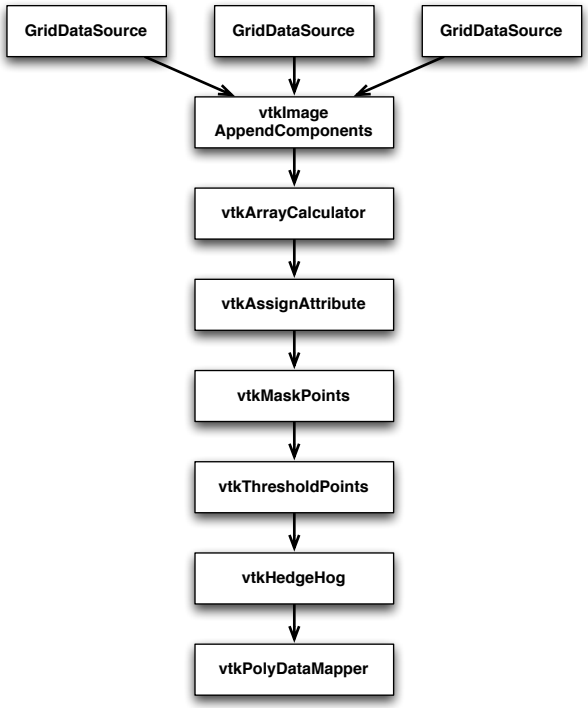

Abbildung 2.7: Das Visualisierungsnetzwerk zur Visualisierung des Vektorfeldes mit *hedgehogs*

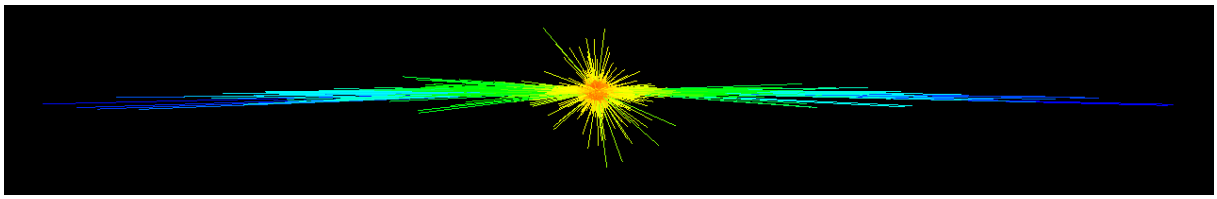

Abbildung 2.8: Die Visualisierung des Vektorfeldes mit *hedgehogs* ohne Skalierung

2.3, die Möglichkeit den Datensatz zu reduzieren. Abbildung 2.9 stellt das Ergebnis für den Beispieldatensatz dar.

2.5.2 Stromlinien

Eine weitere Möglichkeit dreidimensionale Vektorfelder zu visualisieren stellen Stromlinien dar. "Stromlinien folgen in einem gegebenen Zeitpunkt den Geschwindigkeitsvektoren, d.h., eine an eine Stromlinie gelegte Tangente gibt die Strömungsrichtung in diesem Punkt an" [Stö05]. Für ein Vektorfeld $\mathbf{V}$, eine Stromlinie s und einem Punkt s_0 gilt zu einem gegebenen Zeitpunkt t:

$$s(0) = s_0$$
$$\frac{\partial s(r)}{\partial r} = \mathbf{V}(s(r), t).$$

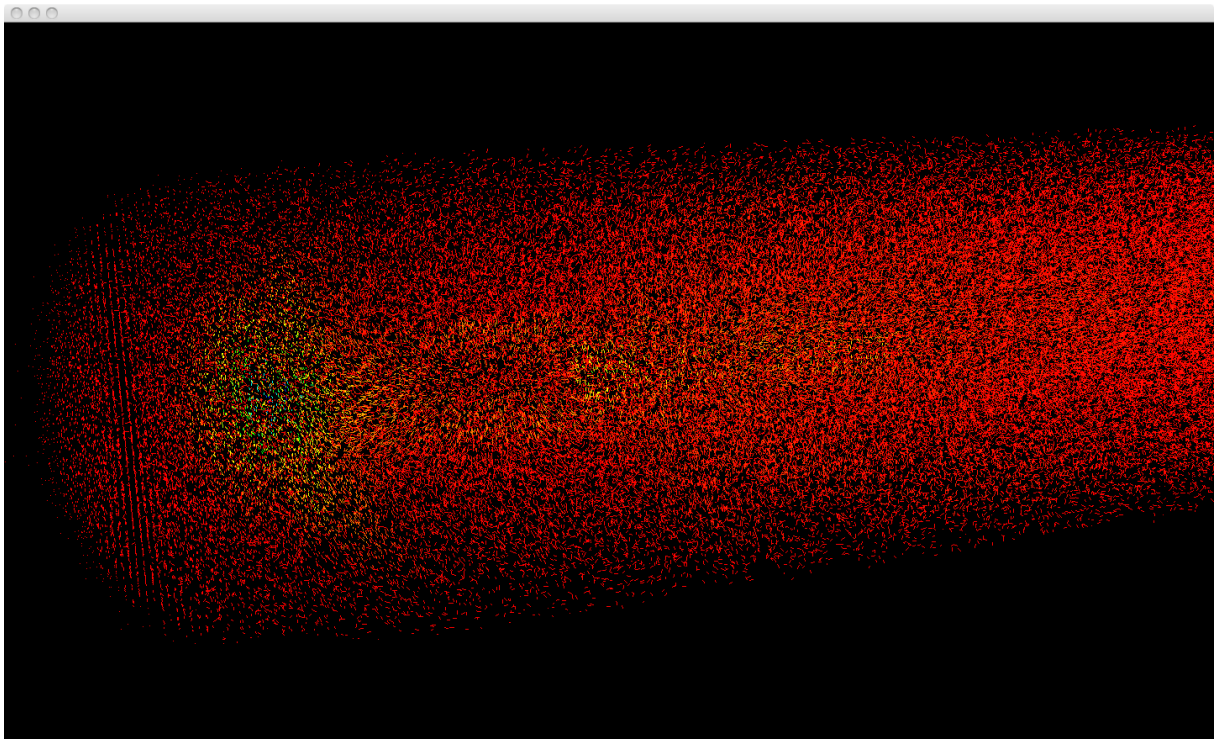

Abbildung 2.9: Die Visualisierung des Vektorfeldes mit *hedgehogs* mit nichtlinearer Skalierung

Stromlinien lassen sich, ausgehend von einem Startpunkt s_0, durch Integration des Vektorfeldes berechnen.

$$s(r) = s_0 + \int_0^r \mathbf{V}(s(\rho), t)d\rho$$

In VTK lassen sich Stromlinien mit dem Filter `vtkStreamTracer` erzeugen. Dieser Filter benötigt zwei Eingangsdatensätze: ein Vektorfeld und eine Menge von *seed points*, Startpunkte für die Integration der Stromlinien. In diesem Beispiel werden diese Punkte direkt aus dem Gitter generiert, indem `vtkMaskPoints` einen Großteil der Gitterpunkte maskiert und durchschnittlich jeden n-ten Punkt weiterreicht. Abbildung 2.10 zeigt das Visualisierungsnetzwerk hierfür.

Bei der zufälligen Platzierung der *seed points* und der Berechnung der Stromlinien muss zwischen der Anzahl und Länge der Linien und dem benötigten Speicherplatz und Rechenzeit abgewogen werden. Bei wenigen langen Linien kann es passieren, dass wichtige topologische Merkmale des Vektorfeldes nicht getroffen werden. Viele kurze Linien decken das Feld gut ab, aber heben Quellen und Senken nicht genügend hervor, da nicht genug Linien zusammenfließen können. Sehr viele lange Stromlinien hingegen, führen zu einem hohen Speicher und Rechenaufwand und einer geringen Übersichtlichkeit.

Außerdem ist die Wahl des Integrationsverfahrens entscheidend für die Qualität der Darstellung. Die Standardeinstellung des `vtkStreamTracer` ist das Runge-Kutta-Verfahren 2. Ordnung, das eine feste

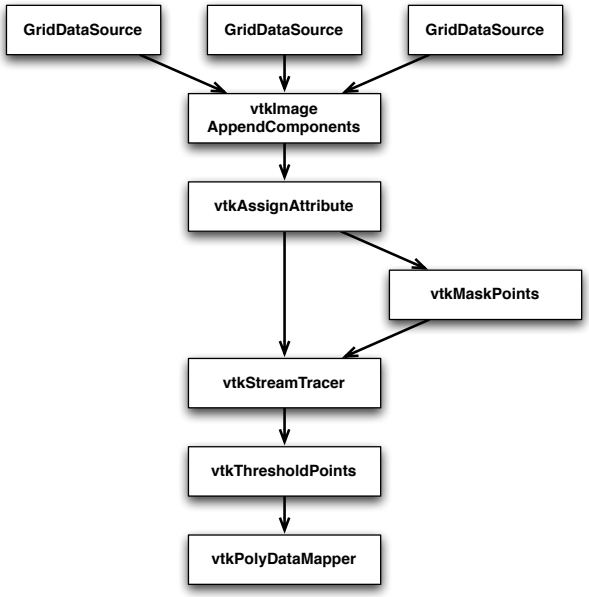

Abbildung 2.10: Das Visualisierungsnetzwerk zur Visualisierung des Vektorfeldes mit Stromlinien

Schrittweite für die Integration verwendet. Dieses kann jedoch, insbesondere bei Senken, zu sichtbaren Artefakten führen. Das Runge-Kutta-45-Verfahren hat eine adaptive Schrittweite und eignet sich deshalb besser für die Visualisierung unbekannter Vektorfeld-Datensätze. Nachteil dieses Verfahrens ist jedoch der höhere Rechenaufwand.

Es gibt weitere Methoden zeitveränderliche Vektorfelder mit charakteristischen Linien zu visualisieren: Bahnlinien, Streichlinien und Zeitlinien. Bahnlinien sind die Trajektorien von masselosen Partikeln in einen Flussfeld über der Zeit, Streichlinien verbinden alle Partikel, die zu einem beliebigen Zeitpunkt durch einen bestimmten Punkt geflossen sind, und Zeitlinien verbinden eine Auswahl an Partikeln und verfolgen diese Line über der Zeit. Hier wird jedoch davon ausgegangen, dass es sich bei dem Vektorfeld um ein Geschwindigkeitsfeld handelt, in dem sich die Teilchen tangential zu dem Vektorfeld zu dem jeweiligen Zeitpunkt bewegen. Die Pfade der geladenen Teilchen ergeben sich jedoch aus einem komplexen Zusammenspiel der $\mathbf{E}$-, $\mathbf{B}$- und $\mathbf{J}$-Felder und werden in der Simulation berechnet, die zu jedem Zeitschritt die Positionen der Partikel abspeichert. Die Bahnlinien der Teilchen werden also nicht während der Visualisierung aus einem Vektorfeld berechnet, sondern aus den Partikeldatensätzen extrahiert, siehe Abschnitt 4.1.4. Die Vektorfelder selbst ($\mathbf{E}$-, $\mathbf{B}$- und $\mathbf{J}$-Felder) werden mit Stromlinien dargestellt. Abbildung 2.11 zeigt die Stromlinien des $\mathbf{E}$-Feldes zum Zeitschritt 100. Abbildung 2.12 zeigt dasselbe Feld nachdem Feldwerte geringen Betrags herausgefiltert wurden.

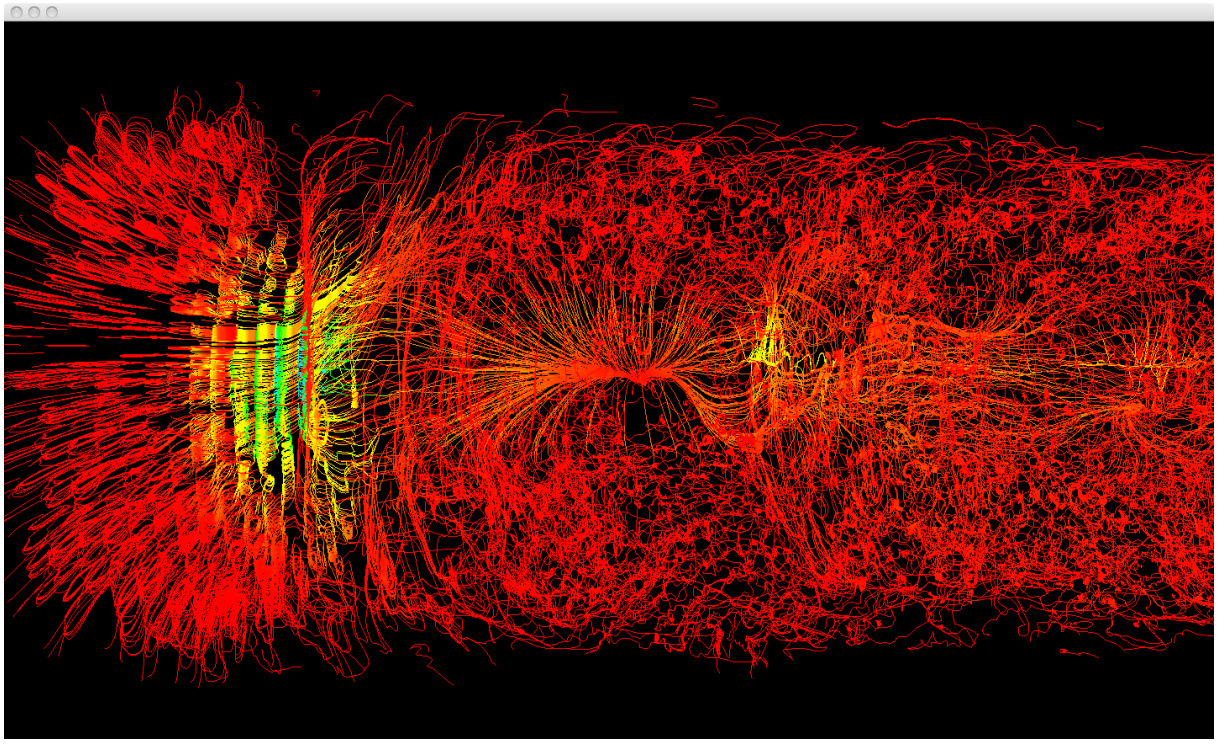

Abbildung 2.11: Visualisierung eines Vektorfeldes mit Stromlinien

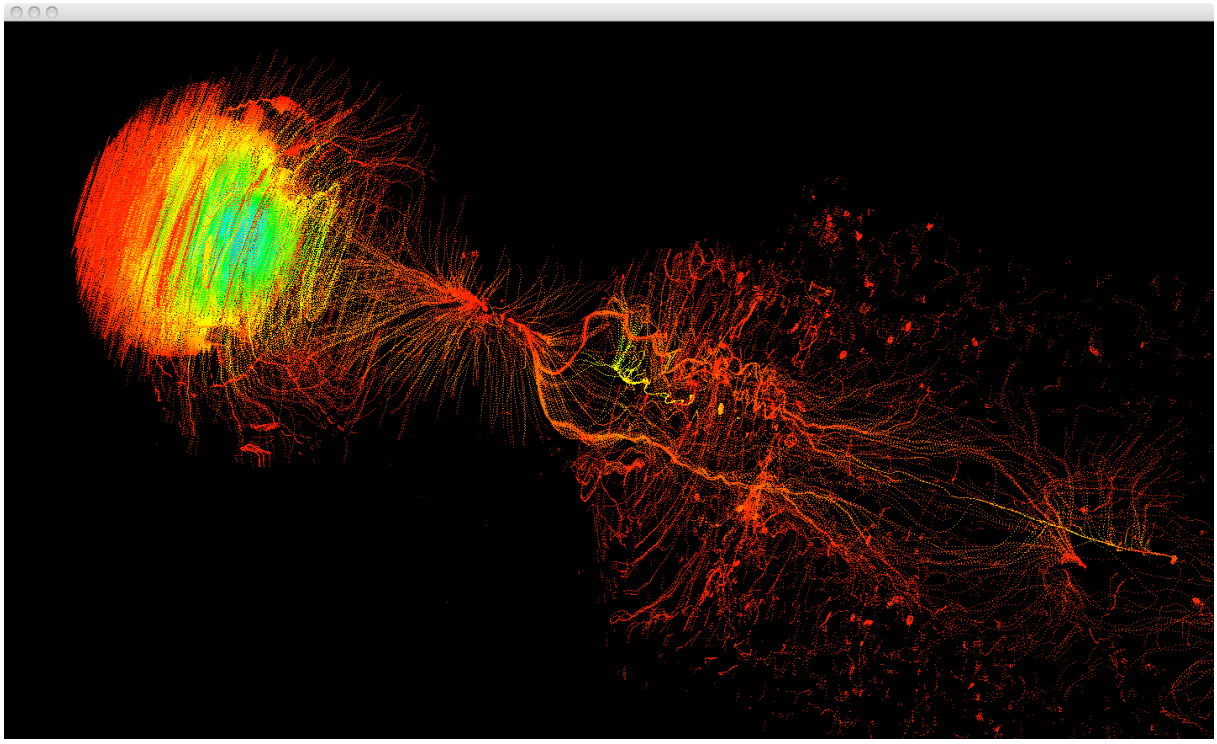

Abbildung 2.12: Visualisierung eines Vektorfeldes mit Stromlinien und Threshold-Filter

3 Verallgemeinerung

Das vorangegangene Kapitel hat alle Dateien einer Simulation als separate Datensätze betrachtet und Visualisierungsnetzwerke vorgestellt. Diese Netzwerke sind jeweils auf die Darstellung von Partikeldaten oder Skalar- und Vektorfeldern spezialisiert. VTK ist auf die Visualisierung von 2D und 3D Datensätzen spezialisiert und bietet keine direkte Unterstützung für die Beschreibung und Visualisierung von Datensätzen beliebiger Dimension und Struktur. Ziel dieses Kapitels ist zunächst die Struktur der Simulationsdaten allgemein und unabhängig von den VTK Datenstrukturen zu beschreiben. Auf Basis dieser Struktur können unterschiedliche Darstellungen erzeugt werden, indem der Nutzer bestimmte Datenfelder auswählt und den Achsen und Attributen einer Visualisierung zuordnet.

Abschnitt 3.1 führt ein Datenmodell ein, das alle Simulationsdaten unabhängig von den zugrundeliegenden Simulationsdateien erfasst. Abschnitt 3.2 betrachtet die unterschiedlichen Visualisierungstypen, die sich aus der Zuordnung bestimmter Felder der Simulationsdaten zu den Achsen und Attributen einer Visualisierung ergeben.

3.1 Datenmodell

Die Daten für jeden Zeitschritt der Simulation werden in separaten Dateien abgespeichert. Die bisherigen Visualisierungen haben sich auf die Darstellung dieser einzelnen Zeitschritte der Simulation konzentriert. Dazu wurden in einem Visualisierungsnetzwerk ein bis drei Ladefilter instanziiert – einer pro Quelldatei – die die Datensätze der Dateien laden und der Visualisierung zur Verfügung stellen. Ein allgemeines Modell der Simulationsdaten abstrahiert diese Sicht. Es nimmt Anleihe an dem relationalen Datenbankmodell, das Daten in Tabellen darstellt und die physikalische Datenspeicherung kapselt. Anwender beschäftigen sich somit allein mit der logischen Struktur der Daten [Gei09]. Im Rahmen dieser Arbeit wird dieses Modell um topologische Beziehungen erweitert.

3.1.1 Tabellensicht

Die Daten der Simulation werden in zwei Tabellen zusammengefasst: eine für die Partikeldaten, "`Particles`", und eine für die Gitterdaten, "`Fields`". Im Gegensatz zu der dateiorientierten Sicht aus Kapitel 2, fasst der Partikeldatensatz sämtliche Teilchen zu jedem Zeitschritt in einer Tabelle zusammen und der Gitterdatensatz alle Skalar- und Vektorfelder der Simulation. Hierbei handelt es sich lediglich um eine logische Sicht auf die vorhandenen Simulationsdaten anhand derer später eine Auswahl für eine konkrete Visualisierung getroffen werden kann. Eine geometrische Interpretation der Daten erfolgt an dieser Stelle nicht, sondern ist Gegenstand von Abschnitt 3.2.

Ein *Tupel* ist eine Zeile dieser Tabelle und ein *Feld* ist eine Spalte. Jedes Feld wird durch einen eindeutigen Namen identifiziert. Ein Eintrag in der Tabelle ist ein *Wert*. Die *Domäne* eines Feldes ist die Menge aller möglichen Werte dieses Feldes. Da es sich um numerische Simulationsdaten handelt, ist die Domäne entweder eine Untermenge der natürlichen Zahlen oder der reellen Zahlen. Die zugrundeliegenden Datentypen sind integrale oder Fließkommatypen (`int`, `unsigned int`, `float`, `double`, usw.). Der *Wertebereich* eines Feldes ist die Menge aller tatsächlich enthaltenen Werte dieses Feldes. Da viele Filter des Visualisierungsnetzwerkes nur den minimalen Wert v_{min} und maximalen Wert v_{max} eines Feldes benötigen, wird der Wertebereich folgend durch $[v_{\text{min}}; v_{\text{max}}]$ gekennzeichnet.

Die Tabelle enthält sowohl Indizes, die für die Spezifikation von Zellen und Gittern benötigt werden, als auch die eigentlichen Simulationsdaten, z.B. Position, Zeit und andere physikalische Größen. Diese Werte können explizit gespeichert oder implizit ermittelt werden. Jedes Tupel einer Tabelle wird durch einen *Primärschlüssel* eindeutig identifiziert, der aus den Werten eines oder mehrerer Felder integralen Typs besteht. Der Primärschlüssel eines Tupels des Partikeldatensatzes besteht aus dem Zeitindex i_t und dem Attribut "`ID`". Der Primärschlüssel eines Tupels des Gitterdatensatzes aus den Indizes i_t, i_x, i_y und i_z. Primärschlüssel werden benötigt, um bestimmte Tupel aus einer Tabelle explizit abfragen zu können.

3.1.2 Zellen und Gitter

Abschnitt 2.2.1 hat Zellen eingeführt, die eine topologische Beziehung zwischen den Tupeln eines Datensatzes herstellen. Ein Knoten ist eine Zelle, der genau ein Tupel der Tabelle zugeordnet ist. Eine Kante ist eine Zelle (a, b), die zwei Knoten a und b verbindet. Höherdimensionale Zellen lassen sich nach [HM04] aus dem Kreuzprodukt von Zellen niedriger Dimension erstellen. Das Kreuzprodukt zweier Knoten a und b ist ein neuer Knoten ab. Das Kreuzprodukt $c \times d$ zweier Zellen $c = (c_1, ..., c_n)$ und $d = (d_1, ..., d_m)$ der Dimensionen $\dim(c)$ und $\dim(d)$ ist eine Zelle

$$e = (c_1d_1, c_1d_2, ..., c_1d_m, c_2d_1, c_2d_2, ..., c_2d_m, ..., c_nd_1, c_nd_2, ..., c_nd_m)$$

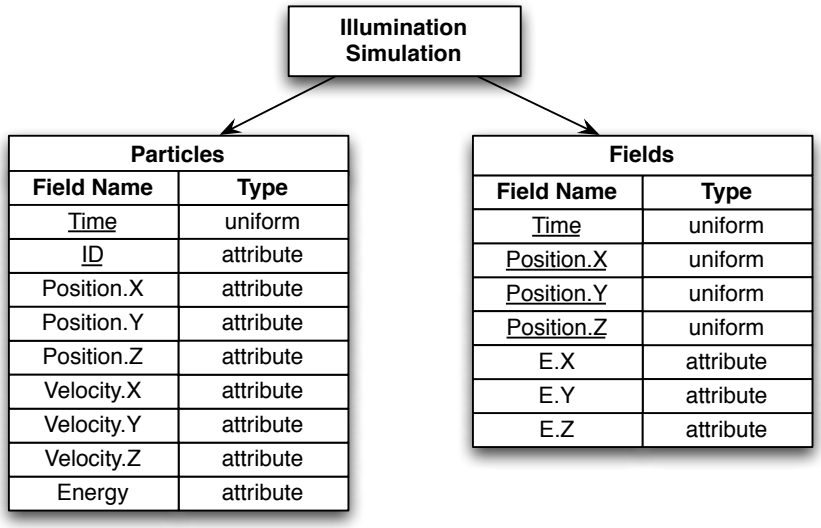

Abbildung 3.1: Datensätze und Feldtypen einer Illumination-Simulation

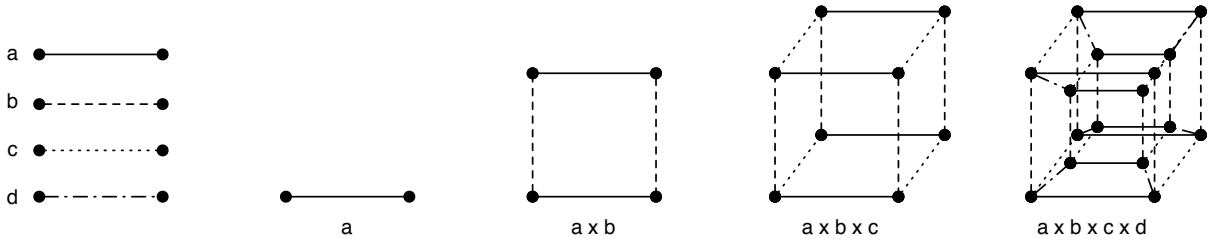

Abbildung 3.2: Konstruktion einer 4D Hypercube-Zelle

der Dimension $\dim(e) = \dim(c) + \dim(d)$. Abbildung 3.2 veranschaulicht wie sich eine 4D-Hypercube[1] Zelle aus dem Kreuzprodukt von vier 1D-Zellen konstruieren lässt.

Zellen dienen unter anderem der lokalen Interpolation von Werten, die den Knoten einer Zelle zugeordnet sind. Anhand einer Interpolationsfunktion lassen sich aus einer Zelle z und einem Interpolationsvector α die Feldwerte im inneren einer Zelle berechnen. Beispiele für Interpolationsfunktionen sind die lineare Interpolation für eine 1D-Zelle

$$m = \alpha \cdot m_1 + (1 - \alpha) \cdot m_2$$

die bilineare Interpolation für Viereckszellen

$$m = \alpha_1 \cdot (\alpha_2 \cdot m_1 + (1 - \alpha_2) \cdot m_2) + (1 - \alpha_1) \cdot (\alpha_2 \cdot m_3 + (1 - \alpha_2) \cdot m_4)$$

oder baryzentrische Interpolation für Dreieckszellen [AMHH08]

$$m = \alpha_1 \cdot m_1 + \alpha_2 \cdot m_2 + \alpha_3 \cdot m_3, \quad \text{mit } \alpha_1 + \alpha_2 + \alpha_3 = 1.$$

[1] Ein Hypercube ist eine Verallgemeinerung eines dreidimensionalen Kubus auf n-dimensionale Räume.

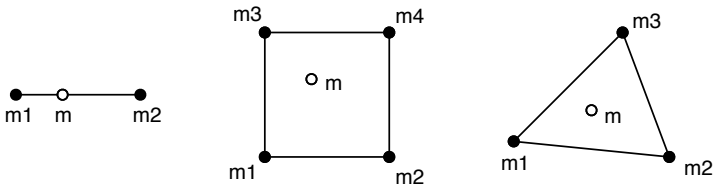

Abbildung 3.3: Berechnung eines Merkmales m aus den Werten m_i an den Knoten einer Zelle

Hierbei sind m_i die Werte eines bestimmten Felds am Knoten i der Zelle und α_i die Interpolationsparameter, siehe Abbildung 3.3.

Felder der Tabelle, die nicht zu der Indexmenge gehören, sind Teil des Beobachtungsraumes oder des Merkmalsraumes. Aus einem Punkt des Beobachtungsraumes lässt sich maximal eine Zelle z und ein zugehöriger Interpolationsvektor α bestimmen, d.h. Zellen dürfen sich im Beobachtungsraum nicht überschneiden. Der Beobachtungsraum des Partikeldatensatzes besteht aus der Zeit t und der Partikel ID. Der Beobachtungsraum des Gitterdatensatzes aus der Zeit t und den Positionen x, y und z. Die Unterscheidung zwischen Beobachtungsraum und Merkmalsraum spielt eine wichtige Rolle bei der Wahl einer geeigneten Visualisierung, siehe Abschnitt 3.2.

Gitter sind nach [HM04] eine Sequenz von Mengen von Zellen $[G_0, G_1, ..., G_d]$, wobei jede Menge G_i Zellen der Dimension i enthält. Das Kreuzprodukt $A \otimes B$ zweier Gitter $A = [A_0, ..., A_a]$ und $B = [B_0, ..., B_b]$ ist ein Gitter $G = [G_0, ..., G_d]$ mit

$$G_k = \bigcup_{j=0}^{k} A_j \times B_{k-j}, \quad \text{mit } 0 \leq k \leq a + b$$

Abbildung 3.4 zeigt verschiedene zweidimensionale Gitter in einem zweidimensionalen Beobachtungsraum. Der Unterschied zwischen gleichmäßigen (a), geradlinigen (b) und strukturierten Gittern (c) ergibt sich durch ihre Einbettung in den Beobachtungsraum. Die Gitter (a) und (b) lassen sich aus dem Kreuzprodukt zweier eindimensionaler Gitter, die jeweils bereits in einem eindimensionalen Beobachtungsraum eingebettet sind, konstruieren. Strukturierte (c) und unstrukturierte (d) n-dimensionale Gitter können allgemein nicht aus dem Kreuzprodukt von Gittern niedrigerer Dimensionen erzeugt werden, sondern werden direkt in einen n-dimensionalen Beobachtungsraum eingebettet. Aus einem gegebenen Punkt des Beobachtungsraumes, der in der Abbildung durch ein Kreuz markiert ist, lässt sich maximal eine zugehörige Zelle eines Gitters bestimmen.

Das Kreuzprodukt zweier gleichmäßiger Gitter der Dimensionen n und m erzeugt ein neues gleichmäßiges Gitter der Dimension $n + m$. Die Zellen eines n-dimensionalen gleichmäßigen Gitters sind n-dimensionale Hypercubes. Der Gitterdatensatz der vorliegenden Simulation besteht aus einem vier-

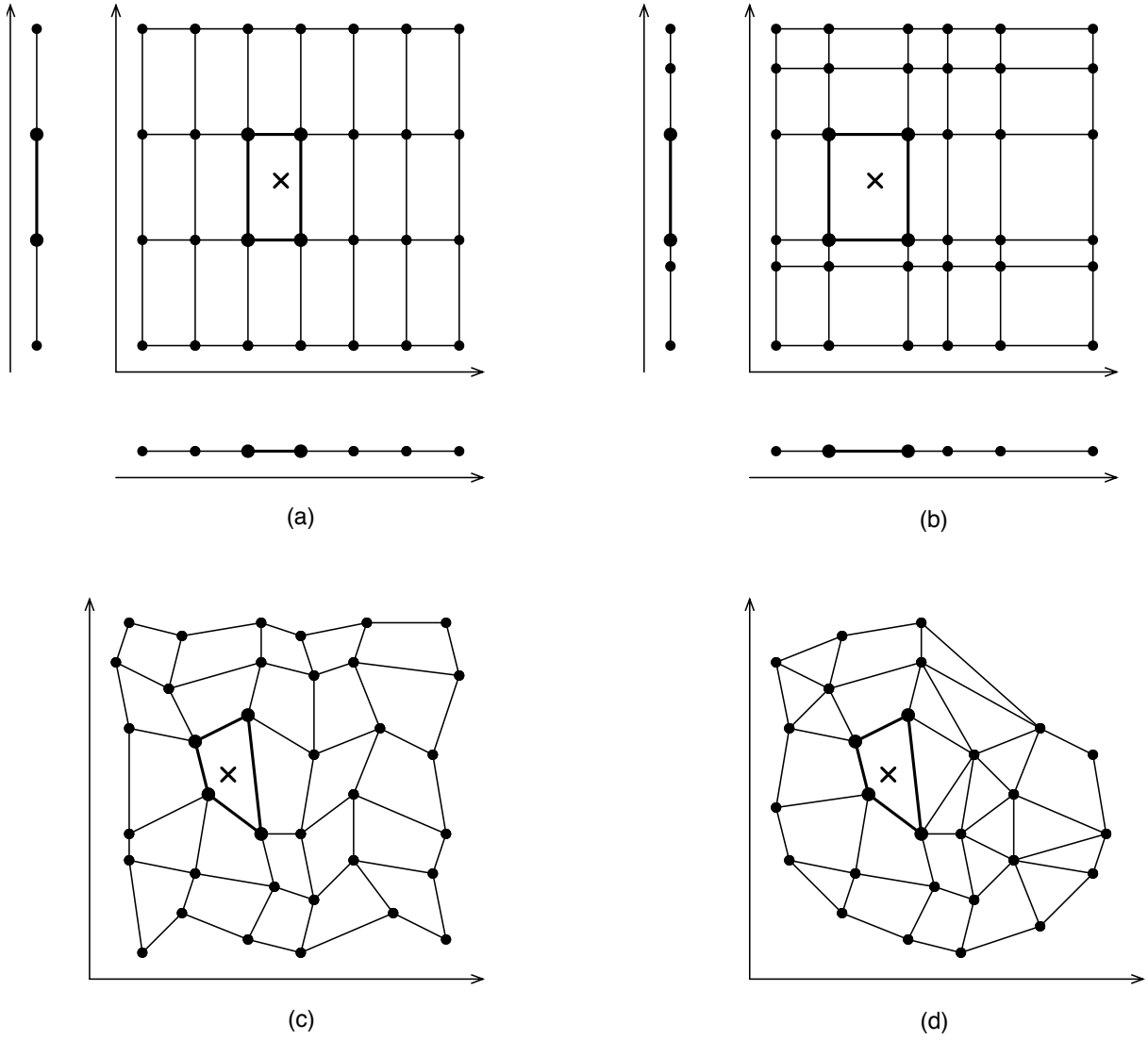

Abbildung 3.4: Verschiedene 2D Gitter und ihre geometrische Einbettung: (a) gleichmäßiges Gitter, (b) geradliniges Gitter, (c) strukturiertes Gitter und (d) unstrukturiertes Gitter. (a) und (b) lassen sich aus dem Kreuzprodukt zweier 1D-Gitter erzeugen.

dimensionalen Gitter dessen Knoten über die Indizes i_x, i_y, i_z und i_t referenziert werden. Ein $n-s$-dimensionaler Schnitt durch dieses Gitter bedeutet, dass s Indizes konstant gehalten werden. Abbildung 3.5 zeigt die Möglichkeiten auf, ein $[0,1]\times[0,1]\times[0,1]\times[0,1]$ Gitter, das aus einer 4D-Hypercube-Zelle besteht, um jeweils eine Dimension zu reduzieren.

Die VTK Gittertypen sind allesamt dreidimensionale Gitter in einem dreidimensionalen Beobachtungsraum. Es ist zum Beispiel nicht möglich ein zweidimensionales $n\times m$ Gitter per Kreuzprodukt aus einem eindimensionalen gleichmäßigen Gitter mit Knotenanzahl n, unter Angabe von *origin* (o_1) und *spacing* (s_1), und einem eindimensionalen geradlinigen Gitter mit Knotenanzahl m, unter Angabe einer geordneten Liste von 1D-Punkten ($p_2(j)$), zu erzeugen. In diesem Fall muss ein $n\times m\times 1$ `vtkStructuredGrid` und ein Array von 3D-Punkten $p(i,j,k) = (o_1 + i \cdot s_1, p_2(j), c)$ erstellt werden, mit c konstant.

Eine Ausnahme bildet `vtkTemporalDataSet` [BGM+07]. Es kombiniert ein beliebiges VTK-Gitter per Kreuzprodukt mit einem 1D-Gitter und ermöglicht Abfragen von Zeitschritten über eine Menge von Zeitwerten aus dem Beobachtungsraum, nicht über einen Zeitindex. Diese neuen Abfragemöglichkeiten und Gittertypen sind jedoch nur auf eine zusätzliche (Zeit-) Dimension beschränkt. Eine Verallgemeinerung auf Datensätze beliebiger Dimensionen existiert in VTK nicht.

Der einzige VTK-Visualisierungsalgorithmus, der wirklich einen vierdimensionalen Datensatz benötigt, ist `vtkTemporalStreamTracer`. Dieser erzeugt Bahnlinien aus einem zeitveränderlichen Vektorfeld. Abschnitt 2.5 diskutierte bereits, warum dieser Algorithmus für die vorliegenden elektromagnetischen Felder nicht geeignet ist. Im Rahmen dieser Arbeit werden nur Visualisierung in drei Dimensionen verwendet. Die Dimensionsreduktion und Filterung des Datensatzes erfolgt in einer separaten Stufe vor der Visualisierung. Deshalb wird `vtkTemporalDataSet` in dieser Arbeit nicht verwendet.

In den folgenden Abschnitten beschreibt der Begriff *uniform*-Feld eine Dimension eines gleichmäßigen Gitters. Er umfasst demnach sowohl die fortlaufenden Indizes als auch deren zugeordneten Werte des Beobachtungsraumes. Alle anderen Felder eines Datensatzes werden als Attributfelder bezeichnet.

3.2 Visualisierungsmodell

Im vorigen Abschnitt wurden der Beobachtungsraum und der Merkmalsraum eingeführt, die jeweils aus *uniform*- und Attributfeldern bestehen. Unterschiedliche Visualisierungen ergeben sich, je nachdem ob den Achsen Felder des Beobachtungs- oder des Merkmalsraumes zugewiesen werden und ob Skalar- oder Vektorattribute gesetzt werden.

Folgende Bezeichner benennen die Attribute einer Visualisierung: "`Axis.X`", "`Axis.Y`" und "`Axis.Z`" bezeichnen die Achsen, "`Scalar`", "`Vector.X`", "`Vector.Y`" und "`Vector.Z`" die Skalar- und

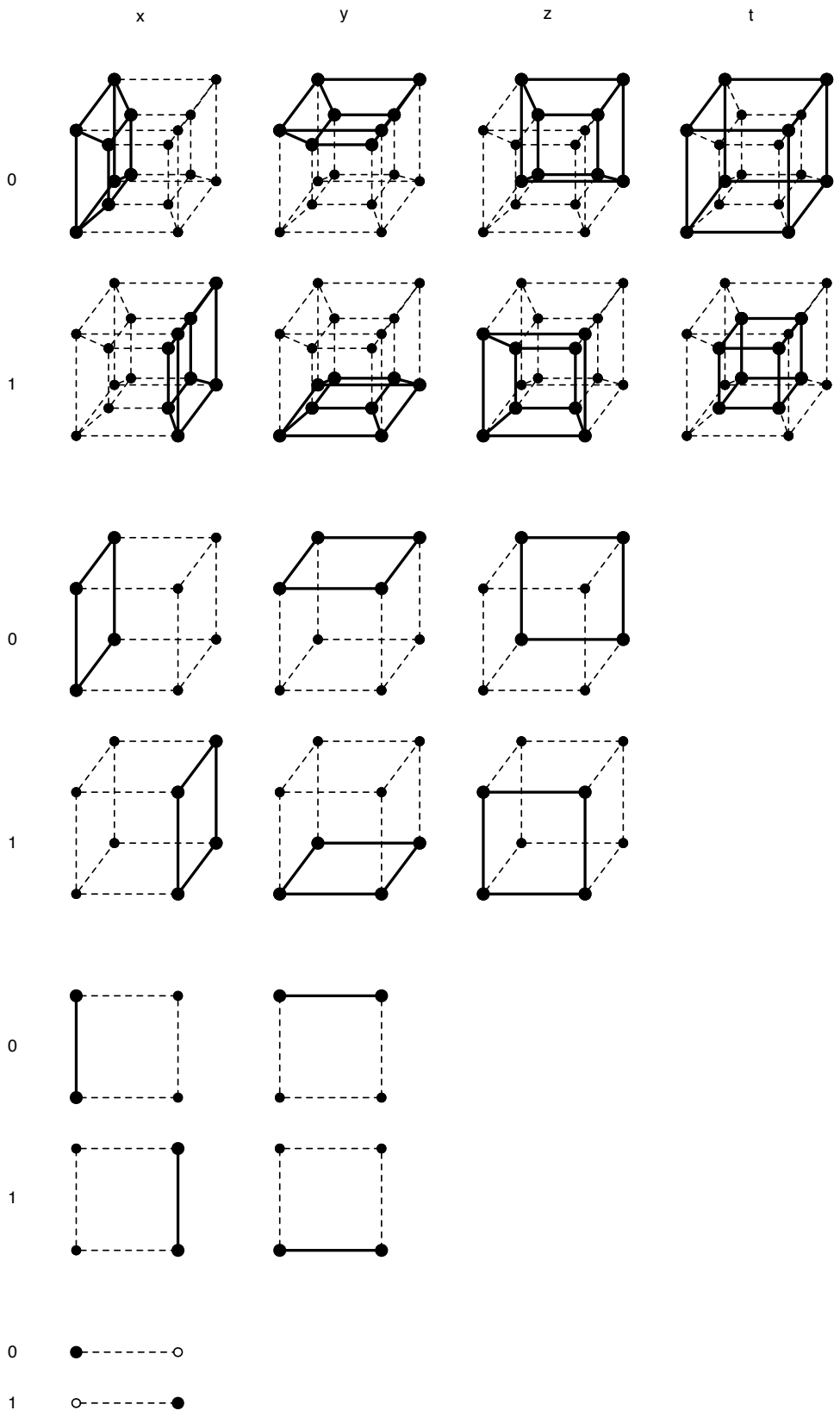

Abbildung 3.5: Subzellen der 4D, 3D, 2D und 1D-Hypercube Zellen.

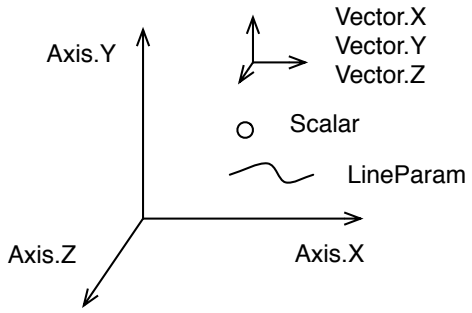

Abbildung 3.6: Schematische Darstellung der Achsen und Attribute einer Visualisierung

Vektorattribute, und "`LineParam`" – ein neues Attribut auf das unten näher eingegangen wird. "`LineParam`" muss ein Datenfeld von Typ *uniform* erhalten. Den Achsen, Skalar- und Vektorattributen können Datenfelder beliebigen Typs zugewiesen werden. Abbildung 3.6 stellt die Attribute der Visualisierungen schematisch dar.

Die topologischen Beziehungen des Datensatzes können für eine Visualisierung genutzt werden, wenn allen drei Achsen jeweils ein Feld des Beobachtungsraumes zugeordnet wird. In diesem Fall kann direkt ein dreidimensionaler VTK-Gitterdatensatz erstellt werden, z.B. `vtkImageData` oder `vtkStructuredGrid`. Zwar könnte man auch einen `vtkStructuredGrid` Datensatz erzeugen, dessen Punkte aus Feldern des Merkmalsraumes zusammengesetzt sind, allerdings erfüllt dieser im Allgemeinen nicht die Forderung vieler Visualisierungsalgorithmen, dass sich aus einem beliebigen Punkt maximal eine zugehörige Zelle bestimmen lässt, d.h. dass sich die Zellen nicht überlappen [SML06]. Der Algorithmus zur Berechnung von Stromlinien nutzt beispielsweise einen Gitterdatensatz, berechnet die Linien schrittweise im Beobachtungsraum, und setzt voraus, dass an jedem Punkt dieses Raumes maximal ein Vektorwert existiert.

Um die Skalar- und Vektorfeldvisualisierungen aus Kapitel 2 zu erzeugen, werden den Achsen die Felder "`Position.X`", "`Position.Y`" und "`Position.Z`" des Gitterdatensatzes zugeordnet. Eine Vektorfeldvisualisierung ergibt sich, wenn die Vektorattribute und optional das Skalarattribut gesetzt werden, eine Skalarfeldvisualisierung, wenn das Skalarattribut gesetzt ist, die Vektorattribute jedoch nicht. Abbildung 3.7 zeigt schematisch die Zuordnung für das Skalarfeld aus Abbildung 2.6. Zu beachten ist, dass der Eingangsdatensatz vierdimensional ist, die Visualisierung aber dreidimensional. Deshalb muss bei einem n-dimensionalen Datensatz ein dreidimensionaler Schnitt ausgewählt werden, indem für die $n-3$ verbleibenden *uniform*-Felder jeweils ein Schnittpunkt gewählt wird. In dem gegebenen Beispiel ist dies ein bestimmter Zeitschritt der Simulation.

Wird den Achsen mindestens ein Datenfeld aus dem Merkmalsraum zugeordnet, erzeugt das Programm

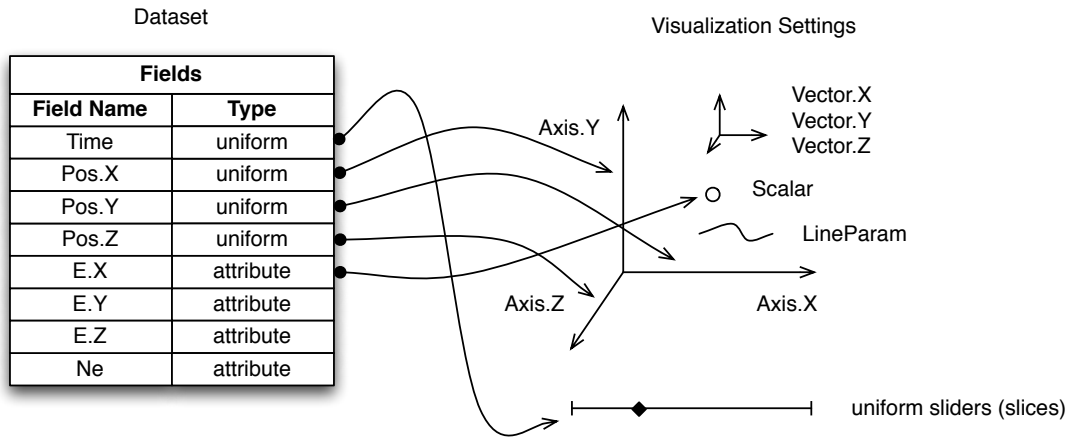

Abbildung 3.7: Beispielzuweisung für ein 3D Skalarfeld

keinen Gitterdatensatz, sondern einen Punktdatensatz, dessen Koordinaten sich aus den Werten der entsprechenden Felder ergeben. Dies ist zum Beispiel bei der Partikelvisualisierung in Abschnitt 2.3 der Fall. Hier wird ein Streudiagramm erzeugt auf das Abschnitt 4.1.2 näher eingeht. Die Skalar- und Vektorattribute sind optional und können beispielsweise die Farbe und Orientierung von Glyphen festlegen. Zu beachten ist, dass hier die Punkte aus einem n-dimensionalen Beobachtungsraum in dem dreidimensionalen Merkmalsraum dargestellt werden können.

Eine Erweiterung dieser Visualisierung ist möglich, indem die topologischen Informationen des ursprünglichen Datensatzes verwendet werden, um bestimmte Punkte durch Linien zu verbinden. Hierzu dient das Attribut "`LineParam`". Ein oder mehrere Felder des *uniform*-Typs lassen sich diesem Parameter zuordnen, um Trajektorien und Drahtgitter (*wireframes*) entsprechender Dimension zu erzeugen. Beispielsweise kann das Feld "`Time`" des Partikeldatensatzes verwendet werden, um die Bahnlinien der Partikel zu visualisieren.

Desweiteren sind spezielle Visualisierungen für gemischte Zuweisungen von mindestens einem Feld des Beobachtungsraumes und mindestens einem Feld des Merkmalsraumes zu den Achsen denkbar, z.B. ein Höhenfeld (*heightmap*) aus zwei Feldern des Beobachtungsraumes und einem Feld des Merkmalsraumes. Diese werden allerdings im Rahmen dieser Arbeit nicht weiter behandelt.

Abbildung 3.8 fasst die möglichen Visualisierung noch einmal zusammen. Dabei steht u für ein *uniform*-Feld des Beobachtungsraumes und m für ein Feld des Merkmalsraumes. Der Schwerpunkt dieser Arbeit liegt auf den dreidimensionalen Visualisierungen. Die folgenden Abschnitte zeigen wie Skalar- und Vektorfeldvisualisierungen sowie der Streu- und Liniendiagramme mit VTK aus den Simulationsdaten erzeugt werden können.

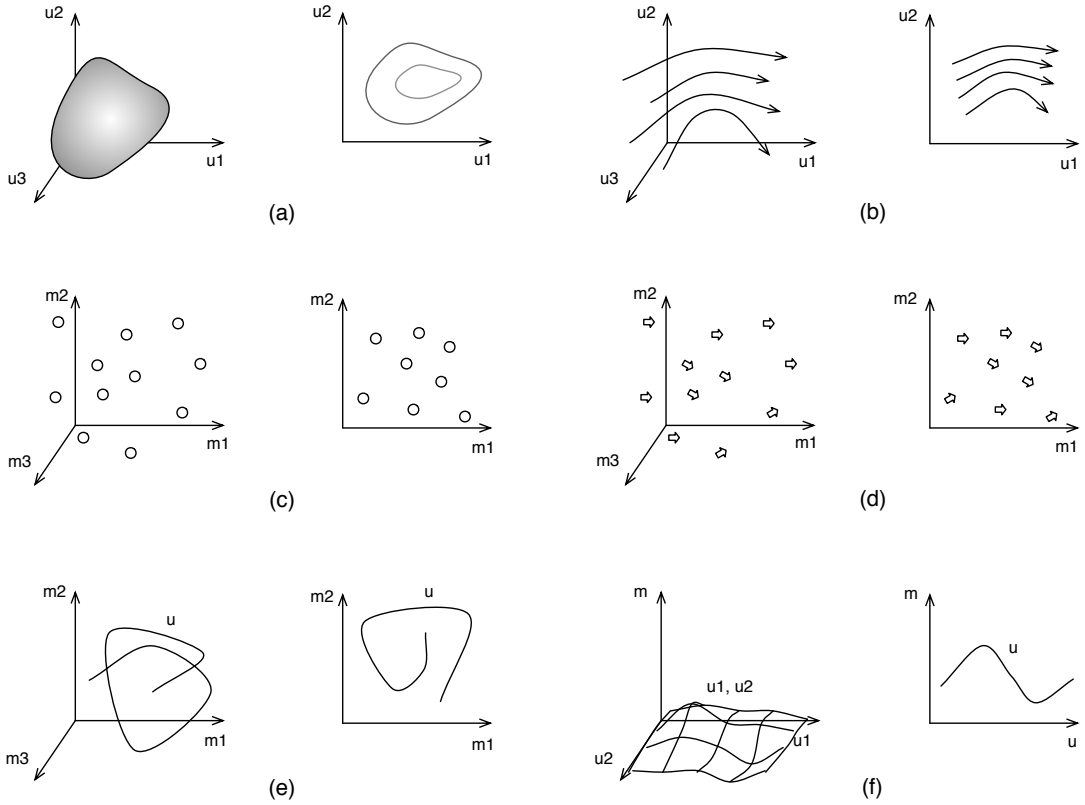

Abbildung 3.8: Übersicht über mögliche Visualisierungen: (a) Skalarfelder, (b) Vektorfelder, (c) Streudiagramme, (d) Streudiagramme mit gerichteten Glyphen, (e) Liniendiagramme: Trajektorien, (f) Liniendiagramme: Höhenfelder, Funktionsgraphen.

4 Implementierung

Dieses Kapitel behandelt, wie die flexible Konfiguration einer Visualisierung auf der Basis sowohl bestehender als auch in dieser Arbeit neu entwickelter VTK-Filter implementiert wurde.

Zunächst werden in Abschnitt 4.1 die Visualisierungen der Skalar- und Vektorfelder sowie Streu- und Liniendiagramme beschrieben. Hier wird auch erklärt, wie ihre Eingangsdatensätze aus den Simulationsdaten erzeugt werden. In Abschnitt 4.2 folgt eine Beschreibung der Konfigurationsklassenhierarchie und der Sprache Visualization Query Language (VQL). Abschnitt 4.3 beschreibt den Aufbau eines kompletten Visualsierungsnetzwerkes und wie dieses aus einer gegebenen Konfiguration erstellt wird.

4.1 Visualisierungen

Unterschiedliche Visualisierungstypen ergeben sich, je nachdem welche Feldtypen den Achsen und Attributen einer Visualisierung zugeordnet werden. Aus den Gitterdaten lassen sich Skalar- und Vektorfeldvisualisierungen erzeugen. Streudiagramme ergeben sich, wenn den Achsen Felder des Merkmalsraumes zugeordnet werden. Außerdem werden Liniendiagramme vorgestellt, die topologische Informationen nutzen um bestimmte Punkte eines Streudiagramms zu verbinden. Eine Normalisierung der Achsen ist erforderlich, wenn den Achsen Felder mit stark unterschiedlichen Wertebereichen zugeordnet werden.

4.1.1 Skalar- und Vektorfelder

Dieser Abschnitt behandelt, wie sich ein dreidimensionaler `vtkImageData` Datensatz aus dem Gitterdatensatz der Simulation erzeugen lässt. Dafür werden drei der vier *uniform*-Felder "`Position.X`", "`Position.Y`", "`Position.Z`"und "`Time`"den drei Achsen zugeordnet. Der Partikeldatensatz hat nur ein *uniform*-Feld – "`Time`"– und lässt sich deshalb nicht direkt zur Erzeugung von dreidimensionalen Skalar- und Vektorfeldern verwenden.

Da die Gitter einer Simulation unter Umständen eine sehr hohe Auflösung haben, insbesondere die zeitliche Dimension, ist es sinnvoll nur einen Teilbereich dieses Gitters für eine Visualisierung zu verwenden, um den benötigten Speicherplatz zu reduzieren. Die Klasse `vql::Range( int from, int to,`

`int step )` erzeugt Indizes zwischen einem Start- und Endwert mit einer gegebenen Schrittweite. Vier Indizes `t`, `x`, `y` und `z` werden benötigt um einen bestimmten Wert des Quelldatensatzes zu lesen und drei Indizes `xAxis`, `yAxis` und `zAxis` um diesen Wert an eine bestimmte Stelle des Zieldatensatzes zu schreiben.

Um die Zielindizes bei einer freien Achsenzuteilung aus den Quellindizes zu erzeugen, bietet `vql::Range` folgende Methoden:

- `currentVal()`: gibt den aktuellen Quellindex zurück.
- `size()`: ermittelt die Anzahl der Quellindizes.
- `i()`: gibt den internen Index i zurück, der von 0 bis $n-1$ läuft, wobei n die Anzahl der Quellindizes ist.

Zum Beispiel durchläuft `x` aus `vql::Range x( 4, 20, 2 )` mit `x.currentVal()` die Werte $4, 6, 8, ..., 20$ und `x.i()` die Werte $0, 1, 2, ..., 8$. Der Rückgabewert von `x.size()` berechnet sich aus `(to-from)/step + 1` und ist in diesem Fall der Wert 9.

Das folgende Listing zeigt, in verkürzter und vereinfachter Form, wie die Klasse `GridToImageDataLoader` aus dem Gitterdatensatz über vier Laufindizes entsprechende Feldwerte lädt und über die internen Indizes `i()` und der Methode `size()` den korrekten Zielindex für das VTK-Array berechnet. Dabei ist `vql::AbstractIDSet` die Basisklasse u.a. von `vql::Range`.

```
void GridToImageDataLoader::load(
        const std::string & name,
        vtkFieldData * fieldData,
        vql::AbstractIDSet * t,
        vql::AbstractIDSet * x,
        vql::AbstractIDSet * y,
        vql::AbstractIDSet * z,
        const vql::AbstractIDSet * xAxis,
        const vql::AbstractIDSet * yAxis,
        const vql::AbstractIDSet * zAxis
)
{
    vtkDoubleArray *array = vtkDoubleArray::New();
    array->SetNumberOfValues( xAxis->size() * yAxis->size() * zAxis->size() );
    array->SetName( name );

    for ( t->begin(); !t->end(); t->next() ) {

        GridData & data = getFile( t->currentVal() )

        for ( x->begin(); !x->end(); x->next() ) {
            for ( y->begin(); !y->end(); y->next() ) {
```

```
                for ( z->begin(); !z->end(); z->next() ) {
                    vtkIdType id = xAxis->i()
                                 + yAxis->i() * xAxis->size()
                                 + zAxis->i() * xAxis->size() * yAxis->size();
                    double val = data.get( x->currentVal(), y->currentVal(), z->currentVal() );
                    array->SetValue( id , val );
                }
            }
        }

    }

    fieldData->AddArray( array );

}
```

Wichtig ist, dass einer der vier Quellindizes nur einen einzigen Wert durchläuft, um aus dem 4D Quellgitter ein 3D Gitter zu erzeugen. Hierfür dient die Klasse `vql::At( int n )`, deren Verhalten equivalent zu `vql::Range( n, n, 1 )` ist. Um beispielsweise das Skalarfeld aus Abbildung 2.6 zu laden, wurde der Zeitschritt 100 gewählt und die x-, y-, und z-Indizes den x-, y- und z-Achsen zugeordnet, wie folgendes Listing zeigt:

```
GridToImageLoader *loader = new GridToImageLoader( "EX.dat" );

vql::At    *t = new vql::At( 100 );
vql::Range *x = new vql::Range( 0, 59, 1 );
vql::Range *y = new vql::Range( 0, 59, 1 );
vql::Range *z = new vql::Range( 0, 385, 1 );

loader->load( "E.X", fieldData, t, x, y, z, x, y, z );
```

Die `vql::At` und `vql::Range` Klassen sind Teil der Abfragesprache VQL, die in Abschnitt 4.2.2 vorgestellt wird. Die neu erstellten Skalar- und Vektorfelder können dann mit den Methoden aus den Abschnitten 2.4 und 2.5 visualisiert werden.

4.1.2 Streudiagramme

Die Partikelvisualisierung in Abschnitt 2.3 verwendete eine feste Zuweisung der Partikelpositionen auf die Punktdaten, der Geschwindigkeit auf das Vektorattribut und der Energie auf das Skalarattribut von `vtkPolyData`. Die resultierende Punktwolke kann als ein Spezialfall eines Streudiagramms (*scatterplot*) angesehen werden. Eine flexible Zuweisung der Datenfelder auf die Achsen dieses Diagrams ermöglicht weitere Darstellungen, z.B. der Teilchenenergie über der x- und z-Position, siehe Abbildung 4.1. Wenn jedes Tupel des Datensatzes mit n Feldern als ein Punkt im n-dimensionalen Raum betrachtet

wird, ist die Achsenzuordnung eine Projektion dieser Punkte in den dreidimensionalen Raum.

Damit die Daten flexibel zugeordnet werden können, müssen diese zunächst in allgemeinster Form vorliegen, als `vtkFieldData`. Der Quellfilter, der die Partikeldaten lädt, erbt nun nicht mehr von `vtkPolyDataAlgorithm`, sondern von `vtkDataObjectAlgorithm` und lädt jedes Feld des Eingangsdatensatzes in ein separates Array von `vtkFieldData`. Für die Zuweisung dieser Felder auf die Punktdaten und Attribute bietet VTK zwei spezielle Filter: `vtkDataObjectToDataSetFilter` und `vtkFieldDataToAttributeDataFilter`.

Mit `vtkDataObjectToDataSetFilter` lässt sich ein polygonaler Datensatz erzeugen, indem ein `vtkPoints`-Array aus drei Arrays eines Eingangsdatensatzes generiert wird. Beispielsweise wurde folgende Zuweisung verwendet, um Abbildung 4.1 zu erzeugen:

```
do2ds->SetPointComponent( 0, "Position.X", 0 );
do2ds->SetPointComponent( 1, "Energy", 0 );
do2ds->SetPointComponent( 2, "Position.Z", 0 );
```

wobei `do2ds` eine Instanz dieses Filters ist. Zusätzlich können weitere Merkmale eines Datensatzes zugewiesen werden, z.B. die Zell-Arrays, um Linien darzustellen. Hierauf geht Abschnitt 4.1.4 ein.

Der `vtkFieldDataToAttributeDataFilter` erzeugt aus den Felddaten entsprechende Skalar- und Vektorattribute. Für die Partikelvisualisierung wurden z.B. folgende Einstellungen verwendet:

```
fd2ad->SetScalarComponent( 0, "Energy", 0 );
fd2ad->SetVectorComponent( 0, "Velocity.X", 0 );
fd2ad->SetVectorComponent( 1, "Velocity.Y", 0 );
fd2ad->SetVectorComponent( 2, "Velocity.Z", 0 );
```

wobei `fd2ad` eine Instanz dieses Filters ist. Die Darstellung des resultierenden Datensatzes erfolgt mit Glyphen, wie sie Abschnitt 2.3 bereits vorgestellt hat.

Zu beachten ist, dass ein Streudiagramm auch aus dem Gitterdatensatz erzeugt werden kann, indem z.B. die Attributfelder “`E.X`”, “`E.Y`”und “`E.Z`”den Achsen zugeordnet werden. Die entsprechende Ladeklasse muss also auch hier allgemeine Felddaten bereitstellen, aus denen die obigen Filter einen polygonalen Datensatz erstellen können.

4.1.3 Normalisierung der Achsen

Durch die freie Zuordnung von Feldern, deren Wertebereiche zum Zeitpunkt des Programmentwurfs nicht bekannt sind, zu den Achsen ergeben sich unter Umständen zwei neue Probleme. Einerseits muss die Größe der Glyphen an den dargestellten Wertebereich angepasst werden, damit diese groß genug sind um sichtbar zu sein und klein genug um sich möglichst wenig gegenseitig zu verdecken und zu

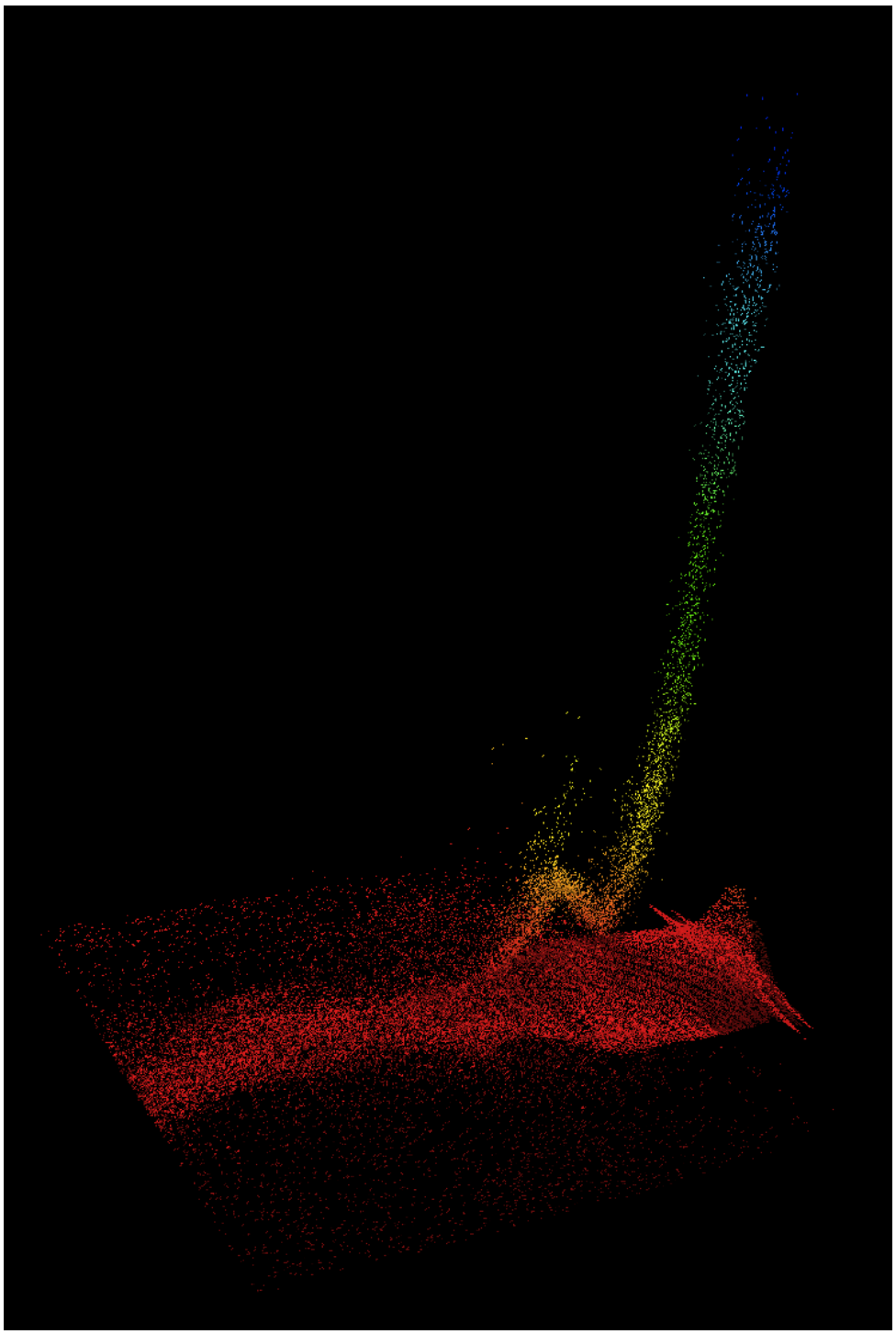

Abbildung 4.1: Darstellung der Teilchenenergie über der x- und z-Position in einem 3D-Streudiagramm

überschneiden. Ein weiteres Problem entsteht wenn den unterschiedlichen Achsen zwei oder mehr Felder zugeordnet werden deren Wertebereiche sich um Größenordnungen unterscheiden. Wenn z.B. der x-Achse das Feld "`Position.Z`" mit dem Wertebereich $[0{,}0; 8{,}0]$ und der y-Achse das Feld "`Energy`" mit dem Wertebereich $[0{,}0; 1e6]$ zugeordnet werden[1], wäre die y-Achse um ca. den Faktor 10^5 länger als die x-Achse.

Eine Normalisierung der zugeordneten Feldwerte ermöglicht eine gleichmäßigere Verteilung der Punktdaten und der Achsenlängen. Der Filter `vtkDataObjectToDataSetFilter` bietet über das Flag `DefaultNormalize` die Möglichkeit Datensätze zu normalisieren. Die Punktdaten aller Achsen werden dadurch auf den Wertebereich $[0{,}0; 1{,}0]$ transformiert. Dadurch, dass nun der Wertebeich bekannt ist, kann den Glyphen eine konstante Größe zugeordnet werden und die Achsen haben auch für unterschiedliche Felder die gleiche Länge.

Allerdings ist diese Normalisierung nicht gut geeignet, wenn den Achsen verwandte Felder zugeordnet werden, z.B. "`Position.X`", "`Position.Y`" und "`Position.Z`" der Teilchen. In diesem Fall führt eine Normalisierung dazu, dass die resultierende Punktwolke bei unterschiedlichen Wertebereichen gestaucht wird. Bei dieser Zuordnung soll das Seitenverhältnis der Achsen erhalten bleiben. Aus diesem Grund wurde ein neuer Filter, `NormalizePoints`, entwickelt der eine Normalisierung der Felder vornimmt, aber die Seitenverhältnisse verwandter Felder beibehält.

Dazu werden die Felder zunächst bestimmten Äquivalenzklassen zugeordnet. Für den Teilchendatensatz aus 2.1 wären dies z.B.:

$$\{\text{“}\texttt{Position.X}\text{”}, \text{“}\texttt{Position.Y}\text{”}, \text{“}\texttt{Position.Z}\text{”}\} \rightarrow 1$$
$$\{\text{“}\texttt{Velocity.X}\text{”}, \text{“}\texttt{Velocity.Y}\text{”}, \text{“}\texttt{Velocity.Z}\text{”}\} \rightarrow 2$$
$$\{\text{“}\texttt{Energy}\text{”}\} \rightarrow 3$$
$$\{\text{“}\texttt{ID}\text{”}\} \rightarrow 4$$

Felder innerhalb einer Klasse werden gleichmäßig skaliert und Felder aus unterschiedlichen Klassen unabhängig voneinander. `NormalizePoints` verwendet hierfür den `vtkTransformFilter`, der Punktdaten beliebig transformieren kann. Für die Normalisierung wird eine `vtkTransform`-Instanz übergeben, die eine Skalierung S, mit

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[1] Die Angaben dienen der Veranschaulichung und entsprechen nicht den Wertebereichen des Datensatzes aus 2.1.

ausführt. Die Skalierungsfaktoren s_x, s_y und s_z werden in `NormalizePoints` aus den Wertebereichen der zugewiesenen Felder und ihrer Klassen berechnet. Zunächst wird für jede Achse mit dem Wertebereiche $[a; b]$ das Maximum von $|a|$ und $|b|$ bestimmt. Dann wird über eine `std::map` jeweils der maximale Betrag der Werte dieser Klassen bestimmt und die entsprechenden Skalierungsfaktoren gesetzt, wie folgendes Listing zeigt.

```
void NormalizePoints::normalize( const double * xRange,
                                 const double * yRange,
                                 const double * zRange,
                                 int xGroup,
                                 int yGroup,
                                 int zGroup )
{
    std::map< int, double > groupMax;

    groupMax[xGroup] = 0.0;
    groupMax[yGroup] = 0.0;
    groupMax[zGroup] = 0.0;

    // get maximum absolute value of each axis
    double xMax = std::max( std::abs(xRange[0]), std::abs(xRange[1]) );
    double yMax = std::max( std::abs(yRange[0]), std::abs(yRange[1]) );
    double zMax = std::max( std::abs(zRange[0]), std::abs(zRange[1]) );

    // determine group maximum
    groupMax[xGroup] = std::max( groupMax[xGroup], xMax );
    groupMax[yGroup] = std::max( groupMax[yGroup], yMax );
    groupMax[zGroup] = std::max( groupMax[zGroup], zMax );

    // scale each axis by group maximum
    double sx = 1.0/groupMax[xGroup];
    double sy = 1.0/groupMax[yGroup];
    double sz = 1.0/groupMax[zGroup];

    this->setScale( sx, sy, sz );
}
```

Tabelle 4.1 stellt die resultierenden Wertebereiche der Normalisierung von `vtkDataObjectToDataSetFilter` (Normalisiert 1) und von `NormalizePoints` (Normalisiert 2) gegenüber. Es werden beispielhaft drei Fälle betrachtet, in denen den Achsen Felder aus ein, zwei und drei unterschiedlichen Klassen zugeordnet sind. Es ist zu erkennen, dass sich die resultierenden Wertebereiche von `NormalizePoints` zwischen $[-1{,}0; 1{,}0]$ befinden und das Seitenverhältnis verwandter Felder beibehalten wird.

Tabelle 4.1: Normalisierung

Achsenzuweisung	Wertebereich	Normalisiert 1	Normalisiert 2
“Position.X”	[-2,0; 2,0]	[0,0; 1,0]	[-0,25; 0,25]
“Position.Y”	[-2,0; 2,0]	[0,0; 1,0]	[-0,25; 0,25]
“Position.Z”	[0,0; 8,0]	[0,0; 1,0]	[0,0; 1,0]
“Position.X”	[-2,0; 2,0]	[0,0; 1,0]	[-0,25; 0,25]
“Energy”	[0,0; 1e6]	[0,0; 1,0]	[0,0; 1,0]
“Position.Z”	[0,0; 8,0]	[0,0; 1,0]	[0,0; 1,0]
“Position.X”	[-2,0; 2,0]	[0,0; 1,0]	[-1,0; 1,0]
“Energy”	[0,0; 1e6]	[0,0; 1,0]	[0,0; 1,0]
“Velocity.Z”	[-1e2; 1e4]	[0,0; 1,0]	[-1e-2; 1,0]

4.1.4 Liniendiagramme

Oft enthalten die Primärschlüssel eines Datensatzes Felder des Typs *uniform*. Wenn alle anderen Werte des Primärschlüssels konstant sind, beschreibt dieses Feld ein eindimensionales Gitter. Die Zellen dieses Gitters können genutzt werden, um Liniendiagramme zu erstellen. Wie bei den Streudiagrammen aus Abschnitt 4.1.2 werden den drei Achsen der Visualisierung mindestens ein Feld des Merkmalsraumes zugeordnet um eine Punktwolke zu erzeugen. Diese Punkte lassen sich nun durch Linien verbinden, indem ein oder mehrere *uniform*-Felder dem “LineParam”-Attribut der Visualisierung zugeordnet werden. Es folgen drei Beispiele, die dieses Prinzip veranschaulichen.

Bereits erwähnt wurden die Bahnlinien der Partikel. Sie lassen sich erzeugen indem der Nutzer den Achsen die “Position.X”, “Position.Y” und “Position.Z” Felder zuweist und dem “LineParam”-Attribut das *uniform*-Feld “Time”. Anstatt nur einen bestimmten Zeitschritt darzustellen, wie dies in Abschnitt 2.3 der Fall war, wird nun ein Zeitbereich ausgewählt, z.B. `vql::Range t(1,100,1)` für die ersten 100 Zeitschritte. Weitere Filter, die im späteren Verlauf dieser Arbeit vorgestellt werden, ermöglichen die Auswahl bestimmter Teilchen IDs, für die die Bahnlinien erstellt werden sollen. Abbildung 4.2 zeigt die Bahnlinien aller Teilchen, die sich zum Zeitpunkt 0 in einer bestimmten xy-Ebene befunden haben.

Auch aus dem Gitterdatensatz lassen sich Liniendiagramme erzeugen, z.B. indem die Attributfelder “E.X”, “E.Y”und “E.Z”den Achsen zugeordnet werden und bestimmte Punkte des xyz-Raumes über der Zeit beobachtet werden. Dadurch lässt sich der Verlauf der **E**-Felder an bestimmten Punkten als

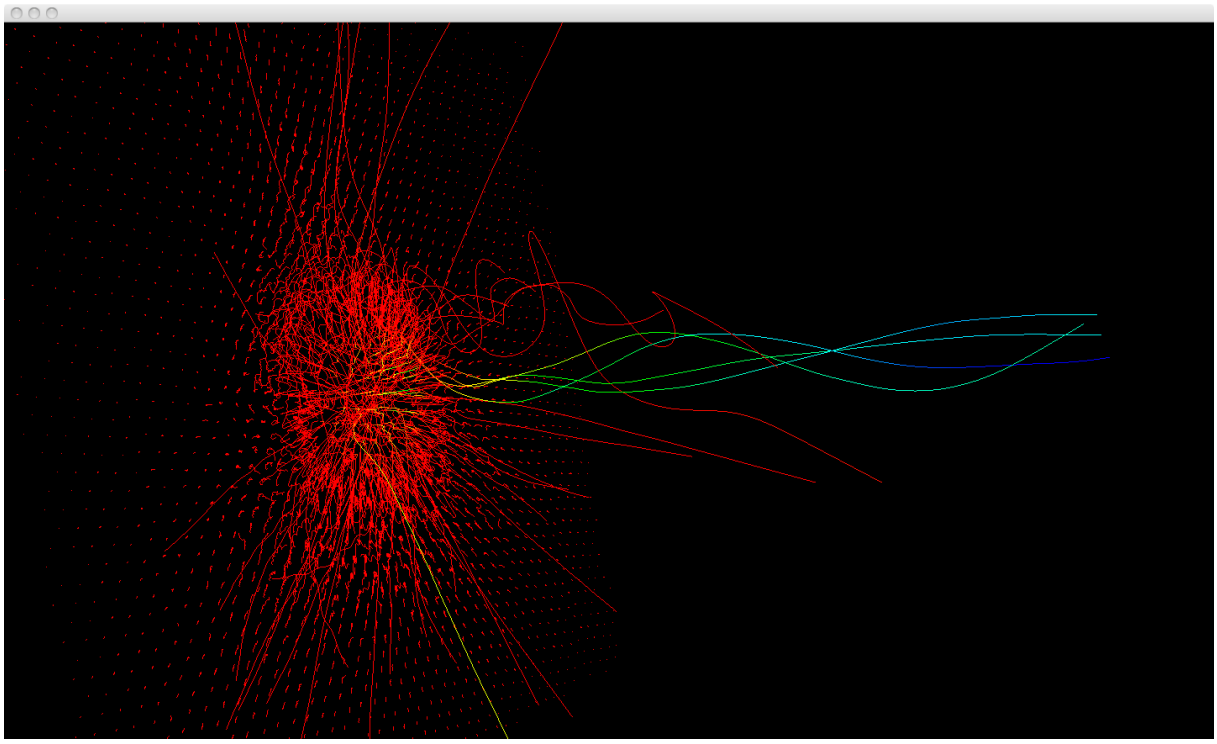

Abbildung 4.2: Bahnlinien der Teilchen einer Ebene.

Trajektorien im **E**-Raum darstellen. Abbildung 4.3 zeigt, dass die **E**-Feldwerte in dem gegebenen Beispieldatensatz nur kurz hohe Beträge annehmen, da die zeitliche Auflösung der Simulation sehr gering ist und sich der Laserimpuls bereits nach einem Zeitschritt wesentlich entfernt hat.

Ein weiteres Beispiel ist die Darstellung des Energieverlaufs eines oder mehrerer Teilchen über der Zeit. Hierzu wählt der Nutzer bestimmte Partikel IDs aus und ordnet sowohl der x-Achse als auch dem "`LineParam`"-Attribut das Feld "`Time`" zu und der y-Achse das Feld "`Energy`".

Folgender Pseudo-Code veranschaulicht, wie die entsprechenden Zelldaten für die Bahnlinien der Partikel erstellt werden. Für alle gewünschten Zeitschritte und alle gewünschten Partikel, erstellt und speichert diese Methode eine neue `vtkIdList` für jede neue Partikel `id`. Dieser Liste wird der Index `i` des aktuellen Datentupels hinzugefügt.

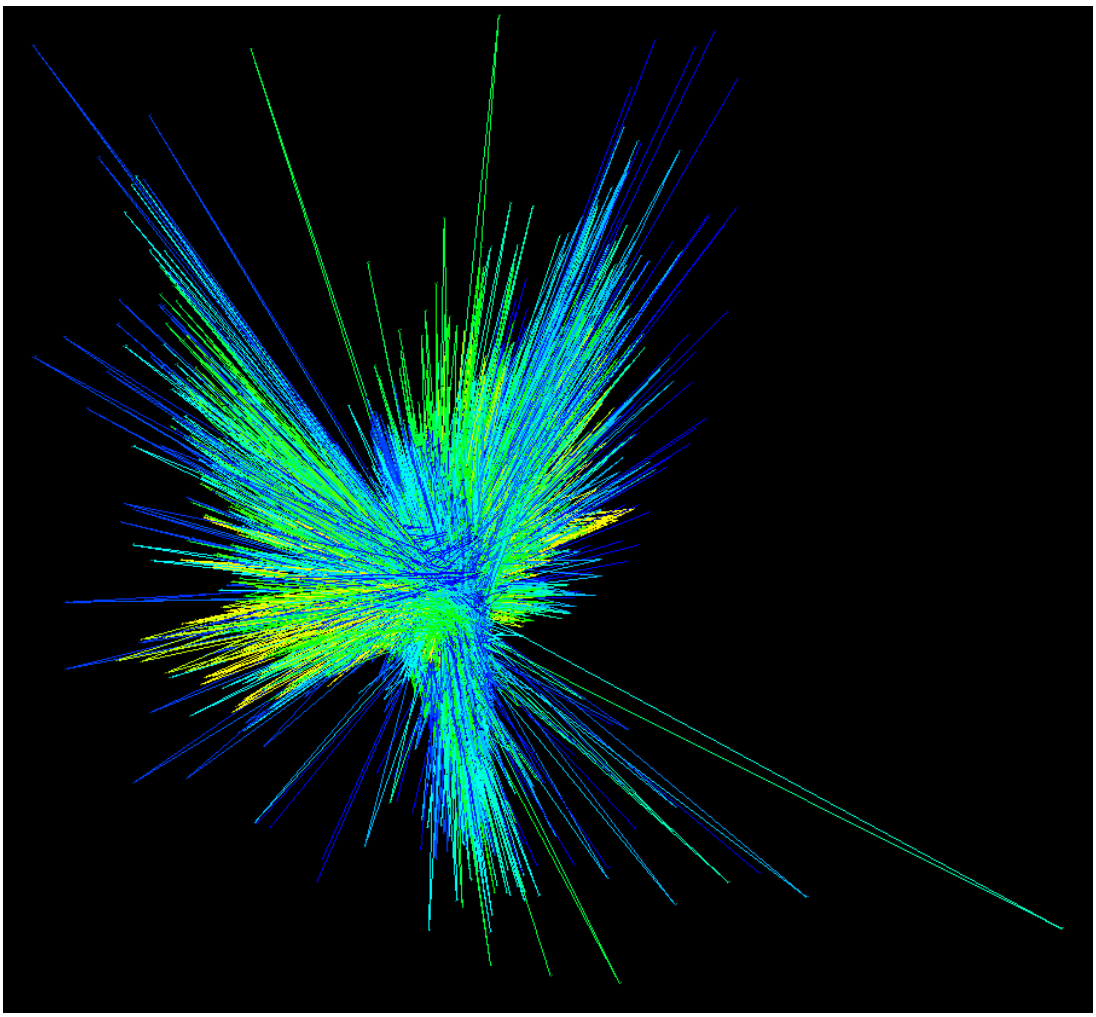

Abbildung 4.3: Trajektorien im “E.X”, “E.Y”und “E.Z”Raum

```
typedef std::map< unsigned int, vtkSmartPointer<vtkIdList> > IDMapType;
IDMapType idLists;

for each time t {
    for each particle p {
        add particle data to field data
        if ( timeLinesOn ) {
            IDMapType::iterator it = idLists.find( p.id );
            if ( it == idLists.end() ) {
                it = idLists.insert( std::make_pair( p.id, vtkSmartPointer<vtkIdList>::New() ) ).first;
            }
            it->second->InsertNextId( i );
        }
        ++i;
    }
}
```

Nachdem alle Partikeldaten geladen sind, müssen die einzelnen Linien, die je in einer Instanz von `vtkIdList` gespeichert sind, in einem Zell-Array zusammengefasst werden. Das folgende Listing zeigt, wie dies implementiert wird. Das resultierende Array wird dann unter dem Name "`PolyLines`" zu `vtkFieldData` hinzugefügt.

```
vtkCellArray *cells = vtkCellArray::New();

for ( IDMapType::const_iterator it = idLists.begin(); it != idLists.end(); ++it ) {
    cells->InsertNextCell( it->second );
}

vtkIdTypeArray *array = vtkIdTypeArray::New();
array = cells->GetData();
array->SetName( "PolyLines" );

fieldData->AddArray(array);
```

Zu beachten ist, dass der obige Code in der Ladeklasse für Partikeldaten eingebettet ist. Diese erbt, wie in Abschnitt 4.1.2 erläutert wurde, von `vtkDataObjectAlgorithm` und erzeugt Datenfelder ohne geometrische Interpretation. Der Filter `vtkDataObjectToDataSetFilter` wird genutzt um hieraus einen `vtkPolyData` Datensatz zu erzeugen. Um aus dem "`PolyLines`" Array *polyline*-Zellen zu erzeugen, muss zusätzlich

```
do2ds->SetLinesComponent( "PolyLines", 0 );
```

aufgerufen werden, wobei `do2ds` eine Instanz dieses Filters ist.

Die Ladeklasse für den Gitterdatensatz kann Linien entlang vier unabhängiger Gitterdimensionen erzeugen. Sie ist analog zu der Ladeklasse der Partikel aufgebaut, wobei anstatt der id, jeweils die Tupel

(x, y, z), (x, y, t), (x, z, t) und (y, z, t) als Schlüssel für eine bestimmte t, z, y und x Linie stehen.

4.2 Konfiguration

Ziel dieses Abschnitts ist eine einheitliche Beschreibung der Visualisierungen zu entwickeln, aus der ein VTK-Visualisierungsnetzwerk automatisch erstellt und konfiguriert werden kann. Die Konfiguration einer Visualisierung umfasst folgende Schritte:

- Auswahl einer oder mehrerer geeigneter Visualisierungsnetzwerke.
- Auswahl eines Datensatzes.
- Angabe der Tupel, die geladen werden sollen.
- Zuweisung bestimmter Felder zu den Achsen und Attributen der Visualisierung.
- (optional) Filterung der Tupel.
- (optional) Transformation von Feldwerten.

Die folgenden Abschnitte beschreiben eine Klassenhierarchie und eine deklarative Sprache, mit der eine solche Konfiguration spezifiziert werden kann.

4.2.1 Konfigurationsklassen

Die Konfigurationsobjekte sind hierarchisch strukturiert. Abbildung 4.4 zeigt die beteiligten Klassen.
Eine Instanz der Klasse `Program` bildet das Wurzelobjekt und enthält die Beschreibung einer oder mehrerer Visualisierungen. Eine Visualisierung wird durch die Klasse `Plot` beschrieben. Sie enthält eine oder mehrere Instanzen der Klasse `VisPipe`, die ein bestimmtes Visualisierungsnetzwerk repräsentiert, und die Beschreibung einer Datenquelle `Source`. Diese Beschreibung umfasst alle nötigen Informationen zum Laden, Filtern und Transformieren von Daten, deren Zuweisung zu den Achsen und Attributen einer Visualisierung, sowie der Option entlang bestimmter Dimensionen Linien darzustellen.

Eine Instanz der Klasse `DataSet` enthält den Namen des Datensatzes der visualisiert werden soll, z.B. "`Particles`" oder "`Fields`". Die Klasse `Query` erzeugt die Primärschlüssel der Tupel, die geladen werden sollen, indem für jedes Indexfeld eine Indexmenge angegeben wird. Dies kann ein einzelner Index sein, `At`, eine Folge von Indizes, `Range`, oder das Ergebnis eine Unterabfrage, `SubQuery`. Ein Beispiel ist die Unterabfrage der IDs der Partikel, die eine bestimmte Bedingung zu einem gegebenen Zeitpunkt erfüllen. Ist für ein bestimmtes Indexfeld keine Angabe vorhanden, erzeugt das Programm implizit sämtliche gültigen Indizes dieses Feldes. Um alle Partikel des Zeitschritts 100 zu laden, muss

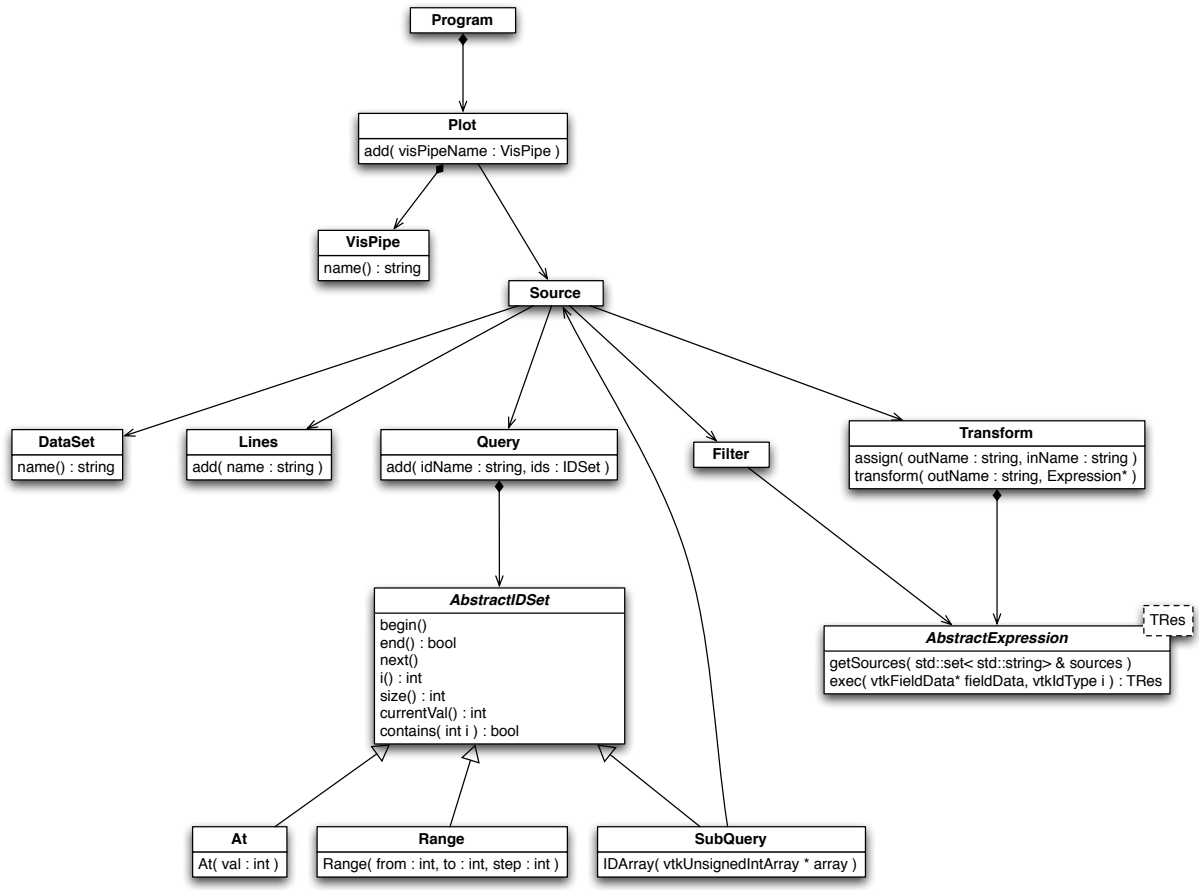

Abbildung 4.4: Klassendiagramm der Konfigurationsklassen einer Visualisierung

also nur `add( "Time", At(100) )` angegeben werden.

Ein Objekt der Klasse `Lines` ist optional und speichert die Namen der Indexfelder, aus denen Linien generiert werden sollen, z.B. "`Time`" , um Partikel über der Zeit zu verfolgen und ihre Bahnlinien zu generieren.

Ein Objekt der Klasse `Filter` enthält ein Prädikat, das von `AbstractExpression<TRes>` abgeleitet ist, wobei hier `TRes` der Typ `bool` ist. Es ist optional und ermöglicht die Überprüfung, ob das jeweilige Tupel eine bestimmte Bedingung erfüllt. Zum Beispiel kann so überprüft werden, ob ein Partikel eine bestimmte Mindestenergie hat. Tupel, die eine Bedingung nicht erfüllen, werden verworfen oder maskiert, je nachdem ob ein Punkt- oder Gitterdatensatz erzeugt wird.

Instanzen der Klasse `Transform` enthalten Informationen über die Zuweisung von Datenfeldern zu den Achsen und Attributen einer Visualisierung. Einfache Zuweisungen, z.B. "`Position.X`" zu "`Axis.X`", können über die Methode `assign()` gesetzt werden. Desweiteren besteht die Möglichkeit, den Attributen Felder zuzuweisen, die sich aus transformierten Quelldatenfeldern ergeben. Dies geschieht über die Methode `transform()`, der der Name des Zielfeldes und ein Ausdruck übergeben werden. Dieser Ausdruck erbt von `AbstractExpression<TRes>`, wobei `TRes` meist vom Typ `double` ist. Beispielsweise kann durch folgende Angabe dem Skalarattribut einer Visualisierung der Logarithmus der Partikelenergie zugewiesen werden:

```
transform( "Scalar", new Log10<double>( new FieldVariable<double>("Energy") ) );
```

Auf die Implementierung der Ausdrücke geht Abschnitt 4.2.3 näher ein. Zunächst wird jedoch eine deklarative Sprache, VQL , vorgestellt, die eine kompakte Beschreibung einer Konfiguration in textueller Form ermöglicht.

4.2.2 VQL

Die Konfigurationsobjekte des vorigen Abschnitts können durch ein interaktives Programm direkt erstellt und manipuliert werden oder mit einer deklarativen Sprache beschrieben und durch einen *Parser* erzeugt werden. Insbesondere für Client-Server-Anwendungen soll es außerdem möglich sein, die Konfiguration auf dem Client zu serialisieren, den resultierenden Programmtext über das Netzwerk zu übertragen, und auf dem Server wieder in entsprechende Konfigurationsobjekte zu übersetzen. Aus diesem Grund ist für alle Klassen der *streaming*-Operator « oder eine `print()` Methode definiert, die die entsprechenden Informationen in einen `std::ostream` schreiben.

Bei VQL handelt es sich um eine deklarative Sprache, deren Syntax an der Syntax der Structured Query Language (SQL) [SQL] angelehnt ist. Unterschiede ergeben sich insbesondere in der Benennung von

Schlüsselwörtern, um der Domäne der Visualisierung Rechnung zu tragen, und in der Möglichkeit topologische Beziehungen zwischen den Tupeln eines Datensatzes (Gitter, Linien, usw.) zu verwenden oder zu erzeugen.

Um einen ersten Überblick über VQL zu bekommen zeigt das folgende Listing, wie die Partikelvisualisierung aus Abschnitt 2.3 mit VQL beschrieben werden kann.

```
VISUALIZE
    Scatterplot
MAP
    Position.X TO Axis.X,
    Position.Y TO Axis.Y,
    Position.Z TO Axis.Z,
    Velocity.X TO Vector.X,
    Velocity.Y TO Vector.Y,
    Velocity.Z TO Vector.Z,
    log10(Energy) TO Scalar
FROM
    Particles
WITH
    Time = 100
WHERE
    Energy > 0.01;
```

Das verwendete Visualisierungsnetzwerk hat den Namen "`Scatterplot`" und erzeugt ein Streudiagramm. Den Achsen werden die x-, y- und z-Positionen des Partikeldatensatzes zugewiesen, dem Vektorattribut die einzelnen Geschwindigkeitskomponenten und dem Skalarattribut das Feld "`Energy`", dessen Werte jedoch zuvor mit dem Logarithmus transformiert werden. Die Angabe `WITH Time = 100` spezifiziert, dass alle Tupel mit dem Zeitindex 100 geladen werden sollen. Diese Tupel werden gefiltert, damit nur Partikel, deren Energiewert größer als 0,01 ist, dargestellt werden.

Das folgende Listing wird genutzt um die VQL Sprachkonstrukte und deren Zuordnung zu den Konfigurationsklassen zu erklären. Eine genaue Spezifikation der Syntax in der Extended Backus-Naur Form (EBNF) [EBN] findet sich in Anhang A.1.

```
VISUALIZATION
    visName1,
    visName2
MAP
    field TO visAttr,
    expression TO visAttr
LINK
    uniformField
FROM
    dataSet
WITH
```

```
    idField = value,
    idField IN RANGE from to step,
    idField IN ( subquery )
WHERE
    predicate
;
```

Das obige Listing beschreibt die Konfiguration einer Visualisierung und entspricht einer Instanz der `Plot` Klasse in der Klassenhierarchie aus Abbildung 4.4. Ein VQL-Programm kann aus einem oder mehrerer dieser Beschreibungen bestehen. Um beispielsweise eine Partikelwolke und ein Skalarfeld gleichzeitig zu visualisieren, kann ein VQL Programm folgende Befehle enthalten.

```
VISUALIZE
    Scatterplot
MAP
...
FROM
   Particles
...;

VISUALIZE
    ScalarField3D
MAP
...
FROM
    Fields
...;
```

Nach dem `VISUALIZE` Befehl müssen die Namen eines oder mehrerer Visualisierungsnetzwerke stehen, die Instanzen der `VisPipe` Klasse entsprechen. Es besteht auch die Möglichkeit mehrere Visualisierungsnetzwerke zu erstellen die gemeinsam einen Datensatz darstellen. Die folgende Auswahl ermöglicht eine Visualisierung der Partikel mit Glyphen und Bahnlinien.

```
VISUALIZE
    Scatterplot,
    Lineplot
MAP
...
LINK
    Time
FROM
   Particles
...;
```

Alle Angaben zwischen `MAP` und `;` beschreiben den Aufbau eines `Source` Objekts. Nach dem Schlüsselwort `FROM` wird der Name des Quelldatensatzes, `DataSet`, angegeben. Nach `LINK` folgt eine durch

Kommas getrennte Liste mit den Namen von *uniform*-Feldern, die spezifizieren, welche Punkte durch Linien verbunden werden sollen. Diese entsprechen den Einträgen in dem `Lines` Objekt.

Das Schlüsselwort `WITH` leitet die Definition eines `Query`s ein. Ihm folgt eine Liste von einem oder mehreren Feldern, die den Primärschlüssel des Datensatzes bilden, und eine Angabe welche Werte für das jeweilige Feld generiert werden sollen. Dies können `field = value` Ausdrücke sein, die `add( field, At(value) )` entsprechen, `field IN RANGE from to step` Ausdrücke, die `add( field, Range( from, to, step ) )` entsprechen, wobei `step` optional ist, oder `field IN ( subquery )` Ausdrücke, die Indizes aus einer Unterabfrage beziehen.

Der `MAP` Befehl beschreibt ein `Transform` Objekt. Ihm folgt eine Liste von einfachen Zuweisungen, z.B. `Position.X TO Axis.X` oder Transformationen, z.B. `log10(Energy) TO Scalar`. Der nächste Abschnitt behandelt die möglichen Ausdrücke. Nach einem `WHERE` Befehlt folgt genau ein Prädikat, das einer `AbstractExpression<bool>` des `Filter` Objekts entspricht.

4.2.3 Ausdrücke

Filter und Transformationen bauen auf `AbstractExpression<TRes>` auf, wobei `Tres` der Rückgabetyp der Methode `exec()` ist und Filter einen boolschen Rückgabewert `bool` benötigen. Dies ist sowohl Relationen, wie =, <, > als auch boolsche Ausdrücke, `and`, `or`, `not` der Fall.

Ausdrücke bilden eine Baumstruktur. Beispielsweise hat der VQL Ausdruck

```
WHERE ID mod 10 = 0 and Energy > 0.01
```

folgende Entsprechung in der Objekthierarchie einer Visualisierungskonfiguration:

```
vql::Filter * filter = new vql::Filter();
 filter->setExpression(
     new vql::And (
         new vql::Equals<int>(
             new vql::Mod<int>(
                 new vql::FieldVariable<int>("ID"),
                 new vql::Const<int>( 10 )
             ),
             new vql::Const<int>(0)
         ),
         new vql::Greater<double>(
             new vql::FieldVariable<double>("Energy"),
             new vql::Const<double>( 0.01 )
         )
     )
 );
```

Ein Ausdruck kann über die Methode `exec()`, die von jeder Spezialisierung von `AbstractExpression<TRes>` definiert werden muss, ausgeführt werden. Sie hat folgende Signatur:

```
virtual TRes exec( const vtkFieldData * fieldData, vtkIdType i ) const = 0;
```

Die Klassen `Const<T>` und `FieldVariable<T>` bilden die Blätter des Baumes und geben Konstante Werte oder Feldwerte zurück. `FieldVariable` ist auch der Grund, warum der `exec()` Methode ein Zeiger zu einem VTK-Felddatensatz und ein Index `i` übergeben werden. Über den Feldnamen, mit dem die `FieldVariable` initialisiert wird, kann die `exec()` Methode das entsprechende Array von `vtkFieldData` abfragen und dessen `i`-ten Wert zurückgeben.

```
T exec( const vtkFieldData * fieldData, vtkIdType i ) const {
    return fieldData->GetArray( name )->GetValue( i );
}
```

Diese Werte werden dann von weiteren Ausdrücken verarbeitet, bis die Wurzel des Baumes erreicht ist und ein Endergebnis zurückgibt. Abbildung 4.5 stellt das Klassendiagramm aller implementierten Ausdrücke dar. Zwei Spezialisierungen, `UnaryExpression<T,TRes>` und `BinaryExpression<T1,T2,TRes>`, bilden die Basisklassen für die unären und binären Relationen und Funktionen. Der Rückgabewert eines unären Ausdrucks ergibt sich aus der Transformation eines Eingangswertes `e` und der Rückgabewert eines binären Ausdrucks aus einer Funktion zweier Eingangswerte `e1` und `e2`. Beispielsweise erbt `vql::Log10<T>` von `vql::UnaryExpression<T,T>` und seine `exec()` Methode ist wie folgt definiert:

```
T exec( const vtkFieldData * fieldData, vtkIdType i ) const {
    return std::log10( e().exec( fieldData, i ) );
}
```

Die Relation `vql::Greater<T>` erbt von `vql::BinaryExpression<T,T,bool>` und gibt `true` zurück, wenn der Wert des Ausdrucks `e1` größer ist als der Wert des Ausdrucks `e2`, sonst `false`.

```
bool exec( const vtkFieldData * fieldData, vtkIdType i ) const {
    return e1().exec( fieldData, i ) > e2().exec( fieldData, i );
}
```

Die Ausdrücke finden in der *Filter*- und *Transform*-Stufe eines Visualisierungsnetzwerks Verwendung. Der folgende Abschnitt behandelt den Aufbau und die Konfiguration eines solchen Netzwerkes.

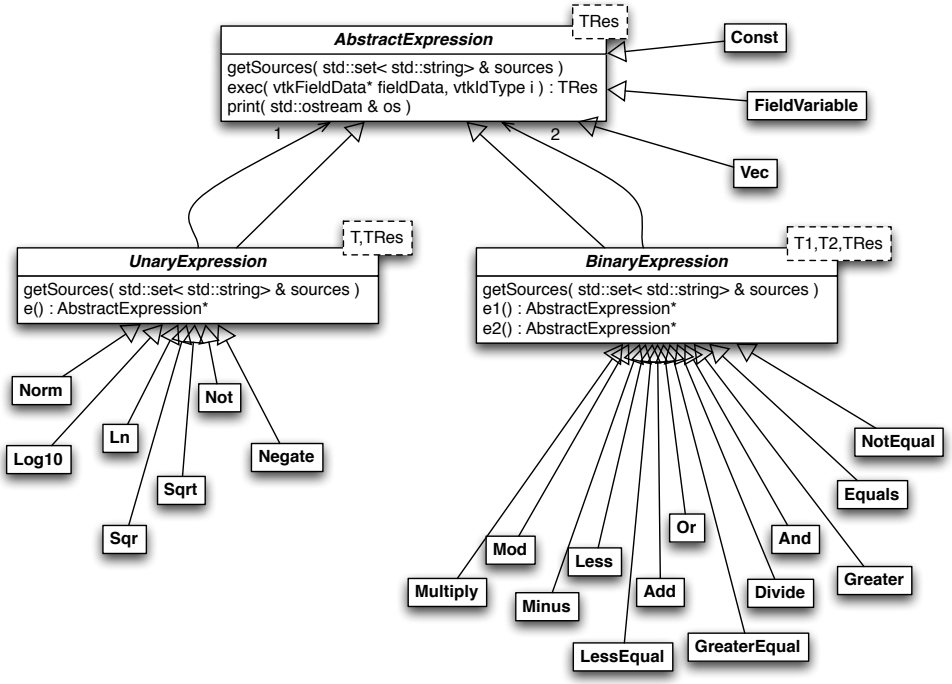

Abbildung 4.5: Klassendiagramm der VQL Ausdrücke

4.3 Visualisierungsnetzwerk

In diesem Abschnitt wird der Aufbau und die Konfiguration eines Visualisierungsnetzwerkes beschrieben. Ein solches Netzwerk soll automatisch aus den Konfigurationsobjekten, die der vorige Abschnitt vorgestellt hat, erstellt werden. Um aus einem Datensatz eine Visualisierung zu erzeugen, muss ein Visualisierungsnetzwerk folgende Schritte ausführen.

1. Die benötigten Feldwerte bestimmter Tupel aus einem Datensatz laden.
2. Die Tupel filtern. Erfüllt ein Tupel die gegebene Bedingung nicht, wird es verworfen oder maskiert.
3. Feldwerte transformieren und neue Werte erzeugen.
4. Die Ergebnisse den Achsen oder Attributen einer Visualisierung zuordnen.
5. Die Achsen normalisieren.
6. Die Daten visualisieren.

Abbildung 4.6 stellt diesen Ablauf schematisch dar. Die Funktionen der einzelnen Schritte müssen nun, um ein ausführbares Visualisierungsnetzwerk zu bilden, auf VTK-Klassen abgebildet werden.

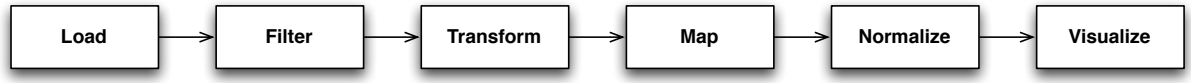

Abbildung 4.6: Datenfluss.

Die Quelle eines Visualisierungsnetzwerkes stellt den nachfolgenden VTK-Filtern eine Auswahl des Datensatzes bereit. Sie wird anhand eines gegebenen `Source`-Konfigurationsobjekts erstellt und konfiguriert. Das Klassendiagramm in Abbildung 4.7 zeigt drei verschiedene `DataSource`-Klassen als Spezialisierungen von `vtkAlgorithm`. Die Auswahl erfolgt anhand der Informationen der `DataSet` und `Transform` Konfigurationsobjekte. Die Klasse `VtkParticleToPolyDataSource` stellt Daten des Partikeldatensatzes für Streu- und Liniendiagramme bereit. Die Klasse `VtkGridToImage-DataSource` stellt einen Gitterdatensatz für die Skalar- oder Vektorfeldvisualisierungen bereit und wird verwendet, wenn allen Achsen ein Feld von Typ *uniform* zugeordnet wurde. Diese Informationen lassen sich aus dem `Transform`-Konfigurationsobjekt ermitteln. Wird mindestens einer Achse ein Feld des Merkmalsraumes zugeordnet, wird die Klasse `VtkGridToPolyDataSource` verwendet, um entsprechende Punkt- und Liniendaten für ein Streu- oder Liniendiagramm bereitzustellen. Linien, bzw. *polyline*-Zellen, werden für jeden Eintrag des `Lines` Objekts erzeugt.

Die `VtkDataSource` Klassen delegieren in dem `RequestData()`-Schritt das Erstellen der Datenfelder an eine oder mehrere Datenquellen, die die Daten generieren oder aus Dateien laden. Über die Methode `load()` erhalten diese Objekte entsprechende Anfragen aus dem `Query`-Konfigurationsobjekt, erzeugen neue Arrays mit den gewünschten Daten und fügen diese Arrays der Instanz von `vtkFieldData` hinzu, die sie von dem `DataSource` erhalten.

Der zweite Verarbeitungsschritt, Filter, erfolgt direkt in den `DataSource`-Klassen, um so früh wie möglich die Menge der gespeicherten Daten zu reduzieren. Sie nutzen direkt den Ausdruck aus dem `Filter`-Konfigurationsobjekt, um diesen auf jedes geladene Tupel anzuwenden. Wenn dieser Filter als separater Algorithmus des Visualisierungsnetzwerkes implementiert würde, müssten zunächst die Werte aller Tupel und Felder in einen Datensatz geladen werden, aus dem der Filter dann ein neuen Datensatz erstellt, der nur noch die gefilterten Tupel enthält. Auch die Felder, die nur zur Auswertung der Bedingung benötigt werden und im späteren Verlauf der Verarbeitung keine Verwendung finden, können so frühzeitig verworfen werden.

Der dritte Verarbeitungsschritt, Transform, geschieht hingegen in einem separaten Filter des VTK-Visualisierungsnetzwerkes. Dazu wurde eine neue Klasse, `VtkDataTransform`, implementiert, die die Angaben des `Transform`-Konfigurationsobjekts nutzt, um einen neuen Datensatz aus dem Eingangs-

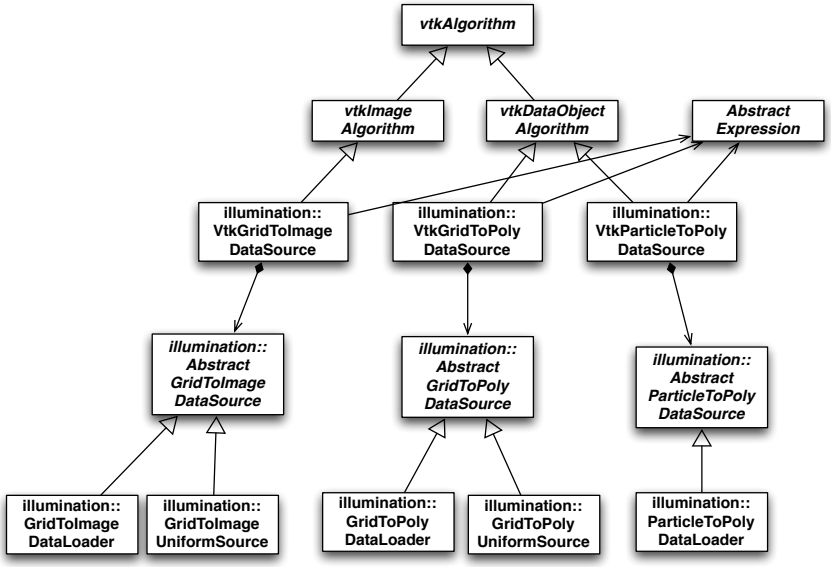

Abbildung 4.7: Klassendiagramm der Datenquellen.

datensatz zu erzeugen. Einfache Zuweisungen, z.B. “`Position.X`” zu “`Axis.X`”, bedeuten, dass das Array mit dem Namen “`Position.X`” in den Ausgangsdatensatz kopiert und nach “`Axis.X`” umbenannt wird. An dieser Stelle muss beachtet werden, dass für gleichmäßige Gitter die “`Axis.X`”, “`Axis.Y`” und “`Axis.Z`” Felder implizit über *origin* und *spacing* Werte vorliegen und nicht als Arrays in `vtkFieldData` erzeugt werden.

Das folgende Listing zeigt wie neue Feldwerte aus den Tupeln des Eingangsdatensatzes `inFieldData` berechnet werden. Der Iterator `it` durchläuft alle Eintrage des `Transforms`-Konfigurationsobjekts, die ursprünglich mit der Methode `transform( outArrayName, expression )` gesetzt wurden, siehe Abschnitt 4.2.2. Dabei ist `it->first` der Name des Zielfeldes, z.B. “`Scalar`”, und `it->second` der Zeiger auf eine Instanz von `AbstractExpression<double>`. Diese wird durch die `exec()`-Methode auf das `i`-te Tupel des Eingangsdatensatzes angewandt, um einen neuen Wert `val` zu berechnen.

```
vtkIdType count = inFieldData->GetNumberOfTuples();

for ( Transforms::const_iterator it = transforms.begin(); it != transforms.end(); ++it ) {

    vtkDoubleArray * array = vtkDoubleArray::New();
    array->SetNumberOfValues( count );
    array->SetName( it->first.c_str() );

    for ( vtkIdType i = 0; i < count ; ++i ) {

        double val = it->second->exec( inFieldData, i );
```

```
        array->SetValue( i, val );

        if ( i % ( count / 100 ) == 0 ) {
            this->UpdateProgress( double(i)/double(count) );
        }
    }

    outFieldData->AddArray( array );
    array->Delete();
}
```

Die `UpdateProgress()`-Methode ist Teil der Schnittstelle von `vtkAlgorithm` und löst Ereignisse aus, über die der Fortschritt des Algorithmus beobachtet werden kann.

Der vierte Schritt, Map, wird durch die beiden VTK-Filter `vtkDataObjectToDataSetFilter` und `vtkFieldDataToAttributeDataFilter` implementiert. Diese wurden bereits in Abschnitt 4.1.2 behandelt, nutzen nun jedoch eine feste Zuweisung der Array-Namen "`Axis.X`", "`Axis.Y`" und "`Axis.Z`" zu den Achsen, "`PolyLines`" zu den Linien, "`Scalar`" zu dem Skalarattribut und "`Vector.X`", "`Vector.Y`", "`Vector.Z`" zu den Vektorkomponenten. Es wird vorausgesetzt, dass die Namen der Arrays durch den Transform-Schritt gesetzt wurden.

Der letzte Schritt vor der eigentlichen Visualisierung ist die Normalisierung der Achsen durch den Filter `NormalizePoints`, der in Abschnitt 4.1.3 entwickelt wurde. Der Visualisierungsschritt danach kann aus einer oder mehreren Visualisierungen bestehen, die durch die `VisPipe`-Konfigurationsobjekte angegeben werden. Zum Beispiel können sowohl ein Streudiagramm als auch ein Liniendiagramm erzeugt und beide mit dem `NormalizePoints`-Filter verbunden werden. Mögliche Visualisierungen für polygonale Daten sind das Streudiagramm und Liniendiagramm, für skalare Felddaten die Isoflächen, und für Vektorfelder die Hedgehogs und die Stromlinien. Diese wurden in den vorangegangenen Abschnitten behandelt.

5 Ausblick

Während der Entwicklung des Systems und der Konzepte für eine flexible Visualisierung von Simulationsdaten ergaben sich diverse offene Fragen und Ansatzpunkte für weitere Entwicklungen. Diese betreffen die Datenhaltung, das Visualisierungsmodell und die Interaktion mit einer Visualisierung.

5.1 Datenhaltung

Das aktuelle Programm bezieht die Daten aus den original Simulationsdateien. Insbesondere wenn zeitliche Verläufe bestimmter Merkmale visualisiert werden sollen, z.B. die Pfadlinien der Partikel, müssen Daten aus sehr vielen Dateien gelesen und gefiltert werden, was sehr zeitaufwändig sein kann. Hier ist eine optimierte Datenhaltung in einer Datenbank wünschenswert. Allerdings sind relationale Datenbanken für den operativen Betrieb von Unternehmen und auf viele kleine, voneinander unabhängige Anfragen und Datenänderungen spezialisiert, z.B. Buchungen, Einkäufe usw. [Gei09]. Bei der Visualisierung der Simulationsdaten arbeiten jedoch nur ein oder sehr wenige Nutzer auf einem Datensatz. Außerdem wird nur lesender Zugriff gefordert, da die Simulationsdaten selbst unverändert bleiben sollen, dies aber auf sehr große Mengen von Daten.

Das *Data Warehouse* ist eine relativ neue Entwicklung im Gebiet der Datenbanken, das sich mit der Archivierung und Analyse von großen Mengen zeitveränderlicher Daten beschäftigt [Gei09]. Da auch diese überwiegend im Unternehmensbereich verwendet werden, um beispielsweise Geschäftszahlen über dem Jahresverlauf zu analysieren, liegen die Daten als Punktdaten, bzw. voneinander unabhängige Zeilen in Tabellen vor. Topologische Informationen, die den Aufbau von n-dimensionalen strukturierten und unstrukturierten Gittern und Zellen beschreiben, wie sie im Bereich der numerischen Simulation physikalischer Phänomene vorliegen, werden durch diese Datenbanken nicht direkt unterstützt.

5.2 Datenreduktion

Um den benötigten Speicherplatz und die Rechenzeit einer Visualisierung zu reduzieren, kann in einer Instanz der Klasse `vql::Range` eine Schrittweite angegeben werden, damit nur jeder n-te Wert geladen

wird. Allerdings kann es dabei zu *aliasing*-Effekten kommen. Abbildung 5.1 betrachtet diesen Effekt beispielhaft anhand eines eindimensionalen Signals im Frequenzbereich. Eine detaillierte Behandlung dieses Themas findest sich u.a. in [Hof98]. Laut dem Abtasttheorem muss ein kontinuierliches Zeitsignal der Grenzfrequenz f_g mit einer Abtastfrequenz

$$f_a > 2 \cdot f_g$$

abgetastet werden, um das ursprüngliche Signal wiederherstellen zu können. Die n Werte, die eine Simulation erzeugt, können als eine Periode eines solchen diskretisierten Zeitsignals aufgefasst werden. In Abbildung 5.1 zeigt (a) das Spektrum eines solchen Signals mit der Grenzfrequenz f_g. Diese Spektren haben jedoch die Eigenschaft, dass sie periodisch sind mit der Periode f_a, siehe (b). Wird nun in dem diskreten Zeitsignal nur jeder zweite Wert gelesen, wie dies mit `step = 2` der Fall ist, bedeutet dies, dass die Abtastfrequenz halbiert wird, d.h. $f_{a2} = 0{,}5 \cdot f_a$. Wie in (c) zu sehen ist, überlagern und addieren sich nun die Spektren, da die neue Abtastfrequenz f_{a2} kleiner ist als $2 \cdot f_g$, und verfälschen somit das Signal. Dieser Effekt wird *aliasing* genannt.

Um diesen Effekt zu vermeiden, muss auch die Grenzfrequenz des ursprünglichen Signals halbiert werden, d.h. $f_{g2} = 0{,}5 \cdot f_g$, indem mit einem Tiefpass alle Frequenzen größer f_{g2} herausgefiltert werden. Hierfür wird das ursprüngliche Spektrum mit einer Rechteckfunktion

$$\text{rect}_{x_0}(x) = \begin{cases} 1 & \text{für} \quad |x| < x_0 \\ 0{,}5 & \text{für} \quad |x| = x_0 \\ 0 & \text{sonst} \end{cases}$$

multipliziert, mit $x_0 = f_{g2}$. Das resultierende Spektrum kann dann in einen Datensatz mit halb so vielen Werten wie ursprünglich zurücktransformiert werden. Es kommt nicht mehr zu Überlagerungen im Frequenzbereich, siehe (d), - *aliasing* wird vermieden.

So wie *mipmaps* für 2D-Texturen in OpenGL erzeugt werden [OSW+08], können für die n-dimensionalen Gitterdaten einer Simulation reduzierte Versionen des Datensatzes berechnet und gespeichert werden. Die folgende Gleichung zeigt, um welchen Faktor sich der benötigte Speicherplatz für ein n-dimensionales Gitter nach m Schritten erhöht, wenn in jedem Schritt i alle die Anzahl der Abtastwerte aller n Dimensionen halbiert wird.

$$\sum_{i=0}^{m} \left(\frac{1}{2^n}\right)^i = \frac{1 - \left(\frac{1}{2^n}\right)^{m \cdot i + 1}}{1 - \frac{1}{2^n}}$$

Eine Abschätzung des benötigten Speicherplatz läßt sich treffen, wenn die Anzahl der Unterteilungsschritte m gegen Unendlich geht:

$$\lim_{m \to \infty} \frac{1 - \left(\frac{1}{2^n}\right)^{m \cdot i + 1}}{1 - \frac{1}{2^n}} = \frac{1}{1 - \frac{1}{2^n}} = \frac{2^n}{2^n - 1} = 1 + \frac{1}{2^n - 1}.$$

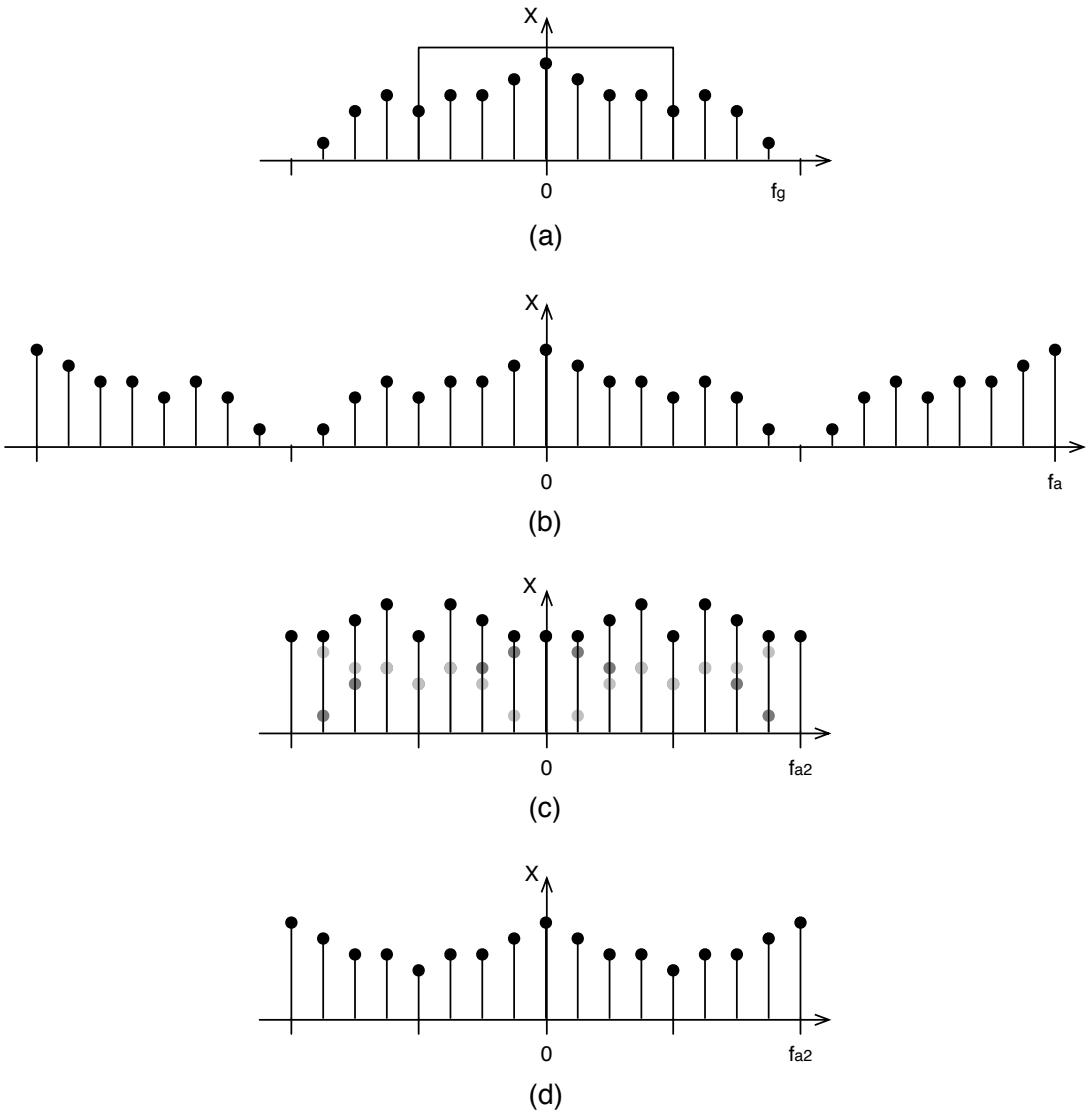

Abbildung 5.1: Bei der Reduktion der Datensätze müssen *aliasing*-Probleme beactet werden.

Bei zwei Dimensionen wird dementsprechend ca. $33{,}3\%$ zusätzlicher Speicherplatz benötigt und bei drei Dimensionen ca. $14{,}3\%$. Der in Abschnitt 2.1 gegebene Gitterdatensatz ist vierdimensional und würde lediglich ca. $6{,}7\%$ zusätzlichen Speicherplatz benötigen.

5.3 Interaktion

Für die Kamerasteuerung ist die Klasse `vtkRenderWindowInteractor` und ihre Spezialisierungen zuständig. Für die Interaktion mit Objekten der Szene sind sog. *widgets* verfügbar, Objekte zur Darstellung und Manipulation von Punkten, Ebenen, Text usw. Die VTK-Widgets erster Generation vereinen den gesamten Kontroll- und Darstellungscode eines Widgets in einer Klasse. Die Widgets zwei-

ter Generation trennen die Ereignisverarbeitung von der Repräsentation der Widgets in zwei Klassen `vtkAbstractWidget` und `vtkWidgetRepresentation`. Das Ziel war, dass Widgets für unterschiedliche Maus- und Tastaturereignisse konfiguriert werden können und die Repräsentation austauschbar ist. Außerdem werden dadurch Client-Server-Anwendungen besser unterstützt [VTK11c].

Die Kamerasteuerung und Widgets in VTK verarbeiten momentan ausschließlich Tastatur- und Maus-Ereignisse. Eine Nutzung mit 3D-Eingabegeräten ist somit nicht direkt möglich, da die Widgets keine 3D-Weltkoordinaten verarbeiten können und ausschließlich mit 2D-Bildschirmkoordinaten arbeiten, wie sie durch eine Maus erzeugt werden. Um weitere Eingabegeräte zu unterstützen ist eine Weiterentwicklung des VTK-Widget-Frameworks nötig. Dieses könnte beispielsweise auf dem Model-View-Controller (MVC)-Entwurfsmuster [ES10] basieren, damit unterschiedliche Steuerungsmöglichkeiten und Darstellungen für ein gemeinsames Modell implementiert werden können. Einige Fragen die sich dabei stellen sind:

- Wie können die Widgets und ihre Funktion modelliert werden, unabhängig von einer konkreten Steuerung und Repräsentation?
- Wie können Steuerereignisse unterschiedlicher Art unterstützt werden? Dies kann Ereignisse aus Skripten, Tastatureingaben, 2D-Bildschirmkoordinaten, 3D-Weltkoordinaten usw. umfassen.
- Wie beeinflussen das Modell und die Steuerung eines Widgets seine Repräsentation? Beispielsweise werden für die Manipulation eines Widgets mit der Maus bestimmte *handles* benötigt, die bei einer 3D-Steuerung entfallen können.
- Wie können gleichzeitig mehrere Steuergeräte unterstützt werden? Zum Beispiel ein 3D-Tracker für die Kamerasteuerung und zwei Datenhandschuhe für die Interaktion mit den Widgets.

Auch Client-Server-Anwendungen müssen hier berücksichtigt werden, da hier die Visualisierung und Modelle auf einem Server gehalten werden, während die Darstellung und Interaktion über einen Client erfolgt.

6 Zusammenfassung

Diese Arbeit hat Konzepte und eine Implementierung für die Visualisierung von Simulationsdaten aus dem Bereich der Laser-Plasma-Physik vorgestellt. Es wurde ein System entwickelt, das aus einer flexiblen Zuordnung von Datenfeldern zu Visualisierungsattributen ein Visualisierungsnetzwerk erstellt und konfiguriert. Das Netzwerk besteht sowohl aus vorhandenen Filtern des Visualization Toolkit (VTK), als auch aus neuen Filtern, die im Rahmen dieser Arbeit entwickelt wurden.

Es wurde ein Datenmodell entwickelt, das Anleihe an dem Datenmodell relationaler Datenbanken nimmt und um topologische Beziehungen erweitert. Im Gegensatz zu dem VTK-Datenmodell ist dieses nicht auf dreidimensionale Visualisierungensstrukuren beschränkt, sondern beschreibt Daten, Zellen und Gitter beliebiger Dimension. Unterschiedliche Darstellungen können durch eine flexible Zuordnung der Felder eines Datensatzes zu den Achsen und Attributen einer Visualisierung erzeugt werden. Auch eine Dimensionsreduktion ist möglich, da die Simulationsdaten in der Regel in mehr als drei Dimensionen vorliegen. Diese einfachen und flexiblen Konfigurationsmöglichkeiten ermöglichen eine explorative Datenanalyse, die mit gängigen Programmen der wissenschaftlichen Visualisierung nicht oder nur mit großem Aufwand möglich ist.

Die Achsenzuordnung und weitere Einstellungen werden in einer Hierarchie von Konfigurationsobjekten gespeichert. Desweiteren wurde eine SQL-ähnliche Sprache namens VQL entwickelt, die eine solche Konfiguration in textueller Form beschreibt. Aus einer Konfiguration wird automatisch ein Visualisierungsnetzwerk erzeugt, das die Schritte: Laden, Filtern, Transformieren, Zuordnen, Normalisieren und Visualisieren, durch VTK-Filter implementiert. Während die Visualisierung überwiegend vorhandene VTK-Filter nutzt, wurden für die Lade-, Filter-, Transformations- und Normalisierungsschritte neue Filter implementiert.

Je nach Achsenzuordnung können verschiedene Visualisierungsnetzwerke verwendet werden. Implementiert wurden Streudiagramme, die durch eine freie Achsenzuteilung Punktwolken eines beliebigen dreidimensionalen Merkmalsraumes darstellen, sowie Liniendiagramme mit denen beispielsweise die Bahnlinien von Teilchen, Trajektorien beliebiger Art und Funktionsgraphen visualisiert werden können. Gitterdaten können visualisiert werden indem allen drei Achsen Felder des Beobachtungsraumes zugeordnet werden. Diese Strukturen bilden die Grundlage für Skalarfeldvisualisierungen mit Isoflächen und

Vektorfeldvisualisierungen mit Hedgehogs oder Stromlinien.

Desweiteren wurden mögliche Verbesserungen und Erweiterungen dieser Arbeit diskutiert. Diese umfassen die Datenhaltung in Data Warehouses, die Datenreduktion unter Vermeidung von Aliasing-Effekten, und Erweiterungen des VTK-Widget-Systems, da dieses eine Interaktion über 3D-Eingabegeräte nicht unterstützt.

Literaturverzeichnis

[ALS+00] AHRENS, James ; LAW, Charles ; SCHROEDER, Will ; MARTIN, Ken ; INC, Kitware ; PAPKA, Michael: A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets. 2000. – Forschungsbericht

[AMHH08] AKENINE-MÖLLER, Tomas ; HAINES, Eric ; HOFFMAN, Naty: *Real-Time Rendering*. 3. Auflage. A K Peters, Ltd, 2008

[BGM+07] BIDDISCOMBE, John ; GEVECI, Berk ; MARTIN, Ken ; MORELAND, Kenneth ; THOMPSON, David: Time Dependent Processing in a Parallel Pipeline Architecture. In: *IEEE Transactions on Visualization and Computer Graphics* (2007)

[BWH+10] BURAU, Heiko ; WIDERA, Renée ; HÖNIG, Wolfgang ; JUCKELAND, Guido ; DEBUS, Alexander ; KLUGE, Thomas ; SCHRAMM, Ulrich ; COWAN, Tomas E. ; SAUERBREY, Roland ; BUSSMANN, Michael: PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. In: *IEEE Transactions on Plasma Science* 38 (2010), October, Nr. 10

[EBN] *EBNF Standard*. ISO/IEC 14977: 1996(E)

[EMP09] EILEMANN, Stefan ; MAKHINYA, Maxim ; PAJAROLA, Renato: Equalizer: A Scalable Parallel Rendering Framework. In: *IEEE Transactions on Visualization and Computer Graphics* 15 (2009), Nr. 3, S. 436–452

[ES10] EILEBRECHT, Karl ; STARKE, Gernot: *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 3. Auflage. Spektrum Akademischer Verlag, 2010

[Gei09] GEISLER, Frank: *Datenbanken. Grundlagen und Design*. 3. Auflage. mitp-verlag, 2009

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. Addison-Wesley Professional, 1994

[GSM06] GEISSLER, Michael ; SCHREIBER, Jörg ; MEYER-TER-VEHN, Jürgen: Bubble acceleration of electrons with few-cycle laser pulses. In: *New Journal of Physics* 8 (2006), September, Nr. 186

[HM04] HOWE, Bill ; MAIER, David: Algebraic Manipulation of Scientific Datasets. In: *Procee-*

dings of the 30th VLDB Conference, 2004

[Hof98] HOFFMANN, Rüdiger: *Signalanalyse und -erkennung: Eine Einführung für Informationstechniker*. Springer-Verlag Berlin Heidelberg, 1998

[LC87] LORENSEN, William E. ; CLINE, Harvey E.: Marching Cubes: A high resolution 3D surface construction algorithm. In: *Computer Graphics* Bd. 21, 1987

[Mat11] *MATLAB - The Language Of Technical Computing*. http://www.mathworks.com/products/matlab/. 2011

[Ope11] *OpenGL Homepage*. http://www.opengl.org/. 2011

[OSW+08] OPENGL ARCHITECTURE REVIEW BOARD ; SHREINER, D. ; WOO, M. ; NEIDER, J. ; DAVIS, T.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1*. 6. Auflage. Addison-Wesley Professional, 2008

[Par11] *ParaView*. http://www.paraview.org/. 2011

[SH09] STARKE, Gernot ; HRUSCHKA, Peter: *Software-Architektur kompakt*. Spektrum Akademischer Verlag, 2009

[SML96] SCHROEDER, William J. ; MARTIN, Kenneth M. ; LORENSEN, William E.: The Design and Implementation Of An Object-Oriented Toolkit For 3D Graphics And Visualization. In: *Proceedings of the 7th conference on Visualization*, 1996

[SML06] SCHROEDER, Will ; MARTIN, Ken ; LORENSEN, Bill: *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. 4. Auflage. Kitware, Inc., 2006

[SQL] *SQL Standard*. ISO/IEC 9075:2008

[Stö05] STÖCKER, Horst: *Taschenbuch der Physik*. 5. Auflage. Wissenschaftlicher Verlag Harri Deutsch, 2004, 2005

[Taj08] TAJIMA, Toshiki: Laser wakefields: Bringing accelerators down to size. In: *Nature Photonics* 2 (2008), September, S. 526 – 527

[VTK04] *The VTK User's Guide: Install, Use and Extend The Visualization Toolkit*. Kitware, Inc., 2004

[VTK11a] *VTK Executives*. http://www.vtk.org/Wiki/VTK/Executives. 2011

[VTK11b] *VTK Homepage*. http://www.vtk.org/. 2011

[VTK11c] *VTK Widgets*. http://www.paraview.org/Wiki/VTKWidgets. 2011

[Wil05] WILKINSON, Leland: *The Grammar of Graphics*. 2. Auflage. Springer, 2005

Abkürzungsverzeichnis

ASCII American Standard Code for Information Interchange

EBNF Extended Backus-Naur Form

GPU Graphical Processing Unit

HZDR Helmholtz-Zentrum Dresden-Rossendorf

LWFA Laser Wakefield Acceleration

MVC Model-View-Controller

PIC Particle-in-Cell

SQL Structured Query Language

VIPER Visual Interactive Plasma computer Experiments for Researchers

VQL Visualization Query Language

VTK Visualization Toolkit

ZIH Zentrum für Informationsdienste und Hochleistungsrechnen

A VQL

A.1 Grammatik

Die VQL Grammatik in EBNF Notation [EBN].

```
start = {program};

program = plot, source, ';';


plot = "VISUALIZE", vispipelist;

vispipelist = name, { ',', name };


source = map, [link], from, [with], [where];


map = "MAP", maplist;

maplist = mapping, { ',', mapping };

mapping = assign | transform;

assign = name, "TO", name;

transform = expr, "TO", name;


link = "LINK", linelist;

linelist = name, { ',', name };


from = "FROM", name;


with = "WITH", idsetlist;

idsetlist = idset, { ',', idset };
```

```
idset = at | range | subquery;

at = name, '=', integer;

range = name, "in", "range", integer, integer | name, "in", "range", integer, integer, integer;

subquery = name, "in", '(', source, ')';


where = "WHERE", predicate;


predicate = "true"
          | "false"
          | "not", predicate
          | predicate, "and", predicate
          | predicate, "or", predicate
          | expr, '=', expr
          | expr, "!=", expr
          | expr, '>', expr
          | expr, '<', expr
          | expr, ">=", expr
          | expr, "<=", expr;

expr = integer
     | real
     | fieldName
     | '-', expr
     | expr, '+', expr
     | expr, '-', expr
     | expr, '*', expr
     | expr, '/', expr
     | expr, "mod", expr
     | "log10", '(', expr, ')'
     | "ln", '(', expr, ')'
     | "sqrt", '(', expr, ')'
     | "norm", '(', expr, ')'
     | "vec", '(', expr, { ',', expr }, ')';
```

VQL Literals as reguläre Ausdrücke.

```
name    = [a-zA-Z][a-zA-Z0-9\.]+
integer = [\-+]?[0-9]+
real    = [\-+]?[0-9]+"."[0-9]+([eE][\-+]?[0-9]+)?
        | [\-+]?[0-9]+[eE][\-+]?[0-9]+
```

A.2 Beispiele

A.2.1 Streudiagramm: Partikel

Beschreibung der Partikelvisualisierung aus Abbildung 2.3.

```
VISUALIZE
    Scatterplot
MAP
    Position.X TO Axis.X,
    Position.Y TO Axis.Y,
    Position.Z TO Axis.Z,
    Velocity.X TO Vector.X,
    Velocity.Y TO Vector.Y,
    Velocity.Z TO Vector.Z,
    log10(Energy) TO Scalar
FROM
    Particles
WITH
    Time = 100
WHERE
    Energy > 0.01;
```

A.2.2 Skalarfeld

Beschreibung der Skalarfeldvisualisierung aus Abbildung 2.6.

```
VISUALIZE
    ScalarField3D
MAP
    Position.X TO Axis.X,
    Position.Y TO Axis.Y,
    Position.Z TO Axis.Z,
    E.X TO Scalar
FROM
    Fields
WITH
    Time = 100;
```

A.2.3 Vektorfeld

Beschreibung der Vektorfeldvisualisierung aus Abbildung 2.11.

```
VISUALIZE
    Streamlines3D
MAP
    Position.X TO Axis.X,
    Position.Y TO Axis.Y,
    Position.Z TO Axis.Z,
    E.X TO Vector.X,
    E.Y TO Vector.Y,
    E.Z TO Vector.Z,
    norm(vec( E.X, E.Y, E.Z )) TO Scalar
FROM
    Fields
WITH
    Time = 100;
```

A.2.4 Streudiagramm: Teilchenenergie

Beschreibung des Streudiagramms aus Abbildung 4.1.

```
VISUALIZE
    Scatterplot
MAP
    Position.X TO Axis.X,
    Energy TO Axis.Y,
    Position.Z TO Axis.Z,
    Velocity.X TO Vector.X,
    Velocity.Y TO Vector.Y,
    Velocity.Z TO Vector.Z,
    Energy TO Scalar
FROM
    Particles
WITH
    Time = 100
WHERE
    Energy > 0.01;
```

A.2.5 Pfadlinien

Beschreibung der Pfadlinienvisualisierung aus Abbildung 4.2.

```
VISUALIZE
    ParametricLines
MAP
    Position.X TO Axis.X,
    Position.Y TO axis.Y,
    Position.Z TO axis.Z,
    log10(Energy) TO Scalar
LINK
    Time
FROM
    Particles
WITH
    Time in range 1 100
    ID in (
        QUERY
            Particles
        WITH
            Time = 0
        WHERE
            Position.Z > 35.0 and Position.Z < 35.1
    );
```

A.2.6 Trajektorien

Beschreibung der **E**-Trajektorien aus Abbildung 4.3.

```
VISUALIZE
    ParametricLines
MAP
    E.X TO Axis.X,
    E.Y TO Axis.Y,
    E.Z TO Axis.Z
LINK
    Time
FROM
    Fields
WITH
    Position.X in range 0 59 5,
    Position.Y in range 0 59 5,
    Position.Z = 50
    Time in range 0 99;
```

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während meiner Diplomarbeit unterstützt haben. Das sind in erster Linie mein betreuender Hochschullehrer an der Technischen Universität Dresden, Prof. Dr. Stefan Gumhold, sowie mein Betreuer am Helmholtz-Zentrum Dresden-Rossendorf (HZDR), Dr. Ulrich Schramm. Besonderen Dank möchte ich außerdem Dr. Michael Bussmann vom HZDR für seine vielfältige Unterstützung aussprechen. Das Visual Interactive Plasma computer Experiments for Researchers (VIPER) Projekt, das er initiiert hat, bildet den Rahmen u.a. für diese Diplomarbeit.

Weiterhin möchte ich Thomas Kluge und Alexander Debus vom HZDR danken, die die Simulationsdatensätze erzeugt und mir sehr geholfen haben, diese Daten und ihre Struktur zu verstehen. Ebenso danke ich Nils Schmeisser und Mirko Gruhl für ihre Unterstützung. Auch das Team vom Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH) an der TU Dresden darf an dieser Stelle nicht unerwähnt bleiben. René Widera und Felix Schmitt waren mir wichtige Ansprechpartner zu Fragen über die PIConGPU Simulation, und über Guido Juckeland bin ich überhaupt erst an das HZDR vermittelt worden. Ohne ihn wäre ich wohl nicht auf dieses interessante Gebiet der wissenschaftlichen Visualisierung aufmerksam geworden.

Natürlich danke ich auch meinen Eltern, die mir dieses Studium ermöglicht und mich sehr unterstützt haben, und meinen Freunden, die mir stets zur Seite standen.