Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2063

Geometrieadvektion zur Visualisierung von instationären Strömungen

Markus Üffinger

Studiengang:	Informatik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Thomas Klein
begonnen am:	02. Mai 2006
beendet am:	31. Oktober 2006
CR-Klassifikation:	I.3.3, I.3.7

Inhaltsverzeichnis

1 Einleitung

Das Untersuchen von Strömungen spielt schon seit mehreren Jahrzehnten eine wichtige Rolle bei der Konstruktion von Fahrzeugen. Im Windkanal werden maßstabsgetreue Modelle oder ganze Fahrzeuge von Luft umströmt, die vorher von Gebläsen beschleunigt wurde. Gemessen werden aerodynamische Eigenschaften der Modelle, wie zum Beispiel der Luftwiderstand oder die Auftriebskräfte, die auf die Tragflächen von Flugzeugen wirken. Unerwünschte, lokal auftretende Verwirbelungen sind nur schwer zu erkennen, da die Luftströmung an sich unsichtbar ist. Für Abhilfe sorgt Rauch, der durch Schlitze oder Sonden in den Versuchsaufbau injiziert wird. Der Rauch wird mit der Luft fortbewegt, und macht so die Strömung indirekt sichtbar. Die aus den Experimenten gewonnen Daten werden für weitere Optimierung bei der Entwicklung der Modelle verwendet. Auch in der Medizin, der Klimaforschung und in vielen anderen Bereichen haben Strömungen eine große Bedeutung. Es werden nicht nur Gase, sondern es wird auch das Strömungsverhalten von Flüssigkeiten erforscht. Mit der rasanten Entwicklung leistungsfähiger Computer verdrängen physikalische Simulationen zunehmend die klassischen Versuchsaufbauten. Simulationen bieten sich insbesondere dann an, wenn das Modell nur im Computer existiert (CAD), die Effekte sich nicht direkt beobachten lassen (z.B. bei Molekularbewegungen) oder reale Versuche zu teuer oder riskant sind (etwa bei Crashtests). Basierend auf den Navier-Stokes-Gleichungen berechnet die numerischen Strömungssimulation (CFD, Computational Fluid Dynamics) beliebige Strömungsprobleme unter den vorgegebenen Randbedingungen. Einzig die Rechendauer beschränkt die Komplexität des zu lösenden Systems. Da häufig eine längere zeitliche Entwicklung von räumlichen Systemen von Interesse ist, erhält man als Resultat riesige Datenmengen, die ohne weiter Verarbeitung durch einen Computer kaum interpretierbar sind. Die Computervisualisierung ist hier von entscheidender Bedeutung. Sie ermöglicht eine grafische Darstellung der Simulationsdaten. Es wurde eine Vielzahl von Methoden entwickelt, die lokale und globale Strömungseigenschaften grafisch abbilden. Einige haben reale Experimente als Vorbild, andere Darstellungen sind nur am Computer möglich. Häufig wird die physikalisch motivierte Bewegung von Teilchen oder anderen Objekten visualisiert, die im Feld freigelassen werden. In dieser Studienarbeit wird eine neuartige *Geometrieadvektions-Methode* vorgestellt, die auf dieser Grundidee basiert und aus Performancegründen zum großen Teil auf der GPU (Graphics Processing Unit) implementiert wurde. Die Teilchen werden repräsentiert durch deformierbare, geometrische Objekte wie Kugeln oder Schläuche, die bestrebt sind, ihre ursprüngliche Form beizubehalten. Sowohl durch die Objektbewegung als auch durch die Verformungen können Informationen über die Strömung vermittelt werden. Die dreidimensionale Darstellung erfolgt interaktiv, wobei der Benutzer mit einer großen Anzahl von Versuchsobjekten interagieren kann.

In **Kapitel 2: Strömungsvisualisierung** werden die Grundlagen, die Problemstellungen und die wichtigsten Methoden der Visualisierung von Strömungen behandelt. Dies führt am Ende des Kapitels zu einer Beschreibung der Geometrieadvektion.

Kapitel 3: Physik und Numerik behandelt die physikalischen Grundlagen, die der Bewegung von Objekten in Strömungen zugrunde liegen. Für eine effiziente Simulation wird ein einfaches Modell für deformierbare Objekte eingeführt und die numerischen Verfahren zur Lösung der Systemdynamik besprochen.

Kapitel 4: Numerische Simulation mit der GPU stellt die Architektur von GPUs und die Vorteile gegenüber handelsüblichen CPUs bei der numerischen Simulation vor. Die eigentliche Aufgabe von GPUs ist das Rendern von dreidimensionalen Modellen. Es wird ein Ansatz beschrieben, der es erlaubt,

ihre große Rechenleistung auch für nichtgrafische Anwendungen zu nutzen. Am Ende des Kapitels wird die Implementation des Simulationskerns behandelt.

Kapitel 5 - Implementation führt zuerst in die Benutzung des implementierten Visualisierungsprogramms ein. Außerdem geht es auf wichtige Implementationsdetails ein, wie die Interaktion mit den Versuchsobjekten, die Objektgenerierung oder das Rendern.

Kapitel 6 - Resultate stellt die Resultate, wie die Programmperformance und mehrere Beispielvisualisierungen vor. Ungelöste Probleme werden mitsamt Verbesserungsvorschlägen aufgeführt.

Kapitel 7 - Zusammenfassung schließt die Arbeit ab.

2 Strömungsvisualisierung

2.1 Einführung

Die numerische Strömungssimulation arbeitet auf einem diskreten Berechnungsraum. Oft wird ein kartesisches Gitter verwendet. Es gibt aber auch allgemeinere Fälle, bei denen gekrümmte oder sogar unstrukturierte Gitter von Vorteil sind. Je nach räumlicher Dimension wird die Strömung an jedem Gitterpunkt durch einen n-dimensionalen Vektor aus reelen Zahlen beschrieben. Dieser kann als Geschwindigkeitvektor des Strömungsfeldes interpretiert werden und legt somit die Richtung und die Stärke des Feldes fest. Dazu kommen weitere, zugeordnete physikalische Größen, wie das Druckfeld der Strömung, das bei der Simulation benötigt wird. Im Folgenden werden dreidimensionale Strömungen betrachtet, die auf gleichförmigen, kartesischen Gittern gegeben sind. Ist die zeitliche Entwicklung instationärer Strömungen von Interesse, dann muss das Feld zusätzlich in diskrete Zeitschritte zerlegt werden. Für alle gewünschten Zeitpunkte t wird der Zustand des Feldes $\boldsymbol{V}_t$ gespeichert,

$$\boldsymbol{V}^t_{i,j,k} \in \mathbb{R}^3 \qquad i = 1...X, \;\; j = 1...Y, \;\; k = 1...Z \tag{2.1}$$

$$t = t_0,\; t_0 + \Delta t,\; t_0 + 2 \cdot \Delta t,\; ... \tag{2.2}$$

Kleine und schnell veränderliche Effekte werden von der Simulation nur bei der Wahl einer entsprechend feinen Diskretisierung erfasst. Es können so sehr schnell riesige Datenmengen anfallen, die mit Hilfe einer geeigneten Visualisierung am Computer interpretiert werden müssen. Dabei interessieren sowohl globale Feldstrukturen als auch lokale Feldeigenschaften wie Divergenzen und Turbulenzen oder die Richtung und Stärke des Feldes. Divergenz und Rotation sind mathematische Funktionen, die auf ein kontinuierliches Vektorfeld angewandt werden können. Bei diskreten Feldern, die nur an Gitterpunkten vorliegen, approximiert man die dabei vorkommenden Ableitungen üblicherweise über Finite Differenzen.

Die Divergenz $\nabla \cdot \boldsymbol{V}(\boldsymbol{r})$ beschreibt die lokale Quelldichte eines Strömungsfeldes am Punkt $\boldsymbol{r}$,

$$\nabla \cdot \boldsymbol{V} = \frac{\partial}{\partial x}\boldsymbol{V}_x + \frac{\partial}{\partial y}\boldsymbol{V}_y + \frac{\partial}{\partial z}\boldsymbol{V}_z \tag{2.3}$$

Betrachtet man ein infinitesimales Volumen an der Stelle $\boldsymbol{x}$, dann kann die Divergenz als die Summe der Flüße verstanden werden, die zu einem festen Zeitpunkt durch dessen Oberfläche fließen. Wird von der Strömung Masse transportiert, dann weisst eine Divergenz kleiner Null darauf hin, dass mehr Masse in das Gebiet hineinfließt als herauskommt (Senke), ein Wert größer Null deutet auf eine Quelle hin, an der Masse entsteht.

Über die lokalen Verwirbelungen eines Vektorfeldes trifft die Rotation $\nabla \times \boldsymbol{V}(\boldsymbol{r})$ eine Aussage,

$$\nabla \times \boldsymbol{V} = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \times \begin{pmatrix} \boldsymbol{V}_x \\ \boldsymbol{V}_y \\ \boldsymbol{V}_z \end{pmatrix} \tag{2.4}$$

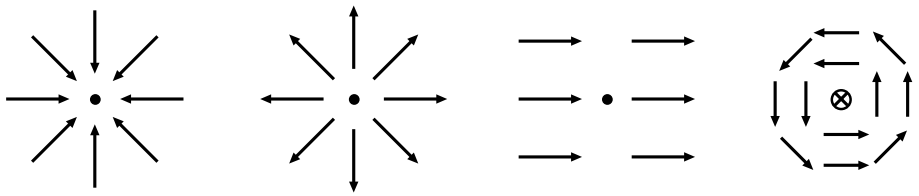


Abbildung 2.1: Rotation und Divergenz eines Vektorfelds

Bei einem Strömungsfeld gibt der Resultatvektor die Rotationsachse und Stärke an, mit der im Feld mitschwimmende Teilchen tendenziell um den Punkt $\boldsymbol{r}$ rotieren.

Globale Feldeigenschaften erfordern eine aufwändige Analyse des Gesamtfeldes und seiner Topologie. Über für das Feld charakteristische, *kritische Punkte*, die untereinander verbunden werden, kann das Feld partitioniert und seine globale Struktur erarbeitet werden. Kritische Punkte sind Stellen im Feld, an denen der Feldvektor verschwindet. Sie lassen sich über die Eigenwerte der Jakobimatrix klassifizieren [1]. Im Fall eines dreidimensionalen Feldes ist die Jakobimatrix ein Tensor, der komponentenweise die Änderung der Feldgeschwindigkeit an einem Raumpunkt $\boldsymbol{r}$ beschreibt.

$$\left[\frac{\partial \boldsymbol{V}}{\partial(x,y,z)}\right]_{\boldsymbol{r}} = \begin{bmatrix} \frac{\partial V_x}{\partial x} & \frac{\partial V_x}{\partial y} & \frac{\partial V_x}{\partial z} \\ \frac{\partial V_y}{\partial x} & \frac{\partial V_y}{\partial y} & \frac{\partial V_y}{\partial z} \\ \frac{\partial V_z}{\partial x} & \frac{\partial V_z}{\partial y} & \frac{\partial V_z}{\partial z} \end{bmatrix}_{\boldsymbol{r}} \tag{2.5}$$

Es gibt keine natürliche Repräsentation von Vektorfeldern, die vom menschlichen Wahrnehmungssystem intuitiv interpretiert werden kann und dabei gleichzeitig alle relevanten Eigenschaften des Feldes enthält. Das Problem dabei ist, eine einfache Abbildung der vektoriellen Daten auf Visualisierungsprimitive zu finden. Zusätzlich führt der dreidimensionale Datenraum schnell zu unübersichtlichen Bildern, da sich die Primitive gegenseitig verdecken und die genaue Position im Raum bei einem zweidimensionalen Bild nur schwer erkennbar ist. Die *visuelle Komplexität* ist generell ein großes Problem, das von den verschiedenen Visualisierungsmethoden unterschiedlich gut gelöst wird. Folgende Vorgehensweisen werden auch von grundsätzlich verschiedenen Verfahren aufgegriffen:

- Reduzierung der Strömungsdaten im Wertebereich.
- Visualisierung abgeleiteter oder zugeordneter skalarer Daten durch eine Abbildung auf Farben. In Frage kommen z.B. Druck, Temperatur, Feldstärke oder Divergenz.
- Beschränkung der Visualisierung auf Schnittflächen durch das Strömungsvolumen.
- 3D Visualisierung mit stereoskopen Verfahren.

Direkte Methoden führen nur wenige Berechnungen auf den Strömungsdaten durch. Es werden nur die lokalen Eigenschaften zu einem festen Zeitpunkt und Ort betrachtet. Beispielsweise kann man die

Feldstärke und Richtung direkt visualisieren, indem man an den Gitterpunkten gerichtete Pfeile unterschiedlicher Länge zeichnet. Diese Darstellung kann aber sehr schnell unübersichtlich werden. Alternativ kann man nur gleichlange Pfeile zeichnen, und die Feldstärke farblich kodieren. Vor allem aber verbietet sich bei dreidimensionalen Darstellungen die *direkte Pfeilrepräsentation* aufgrund gegenseitiger Verdeckungen. Neben der Pfeildarstellung und der Kodierung von Merkmalen durch Farben können außerdem dem Feld zugeordnete skalare Daten durch Isoflächen oder Volumenrendering veranschaulicht werden.

Strömungsproben erfassen lokal eingeschränkte Bereiche des Feldes. Im Gegensatz zu Pfeilen kann die Probe neben der Geschwindigkeit auch die Änderung derselben anzeigen. Analog zu wirklichen Meßgeräten wird die Probe interaktiv im Feld platziert. Für die Darstellung stehen verschiedene Glyph-Objekte zur Verfügung [2].

Die *Indirekten Methoden* basieren auf komplizierteren Vorberechnungen und können neben lokalen auch weitreichende Feldeffekte veranschaulichen. Grob lassen sie sich in drei Klassen einteilen:

- Die *Visualisierung abgeleiteter Daten* extrahiert aus dem Felddatensatz charakteristische Merkmale, die dann grafisch dargestellt werden. Beispielsweise kann dem Betrachter durch eine Visualisierung der globalen Topologie oder von Feldklustern mit unterschiedlichen Eigenschaften Interpretationsarbeit abgenommen werden. Aufgrund des aufwändigen Vorverarbeitungschritts ist eine interaktive Berechnung und Visualisierung meistens aber nicht möglich [1, 3].

- *Dichte Visualisierungsverfahren* wie Texturadvektion über *Line Integral Convolution* erfassen das Feld vollständig. Hierzu werden an den Pixelpositonen von Texturen, die zu Beginn nur weißes Rauschen enthalten, kurze Strömungslinien berechnet. Der Mittelwert aus den Intensitäten dieser Linien bestimmt die Pixelwerte des Resultats. Bei einer animierten Darstellung fällt es dem Betrachter besonders leicht, die dominierenden Flussstrukturen zu erkennen. Allerdings beschränken Verdeckungsprobleme die Darstellung auf zweidimensionale Schnitte durch das Volumen [4].

- Der dritte Ansatz besteht darin, diskrete Objekte im Feld freizulassen und ihre Dynamik oder die durchlaufenen Trajektorien zu visualisieren. Die Objekte sind dabei direkt an das Strömungsfeld gekoppelt und ihre Positionen zu unterschiedlichen Zeitpunkten werden durch *numerische Integration* berechnet. Je mehr Objekte verwendet werden, umso stärker ähnelt das Resultat dem der dichten Methoden, wobei deren Verdeckungsprobleme dann auch verstärkt in den Vordergrund treten.

Nachfolgend werden verschiedene grundlegende Objekt-Integrationsmethoden und insbesondere die darauf basierende *Geometrieadvektion* vorgestellt.

2.2 Integrationsmethoden

Die analytische Berechnung der Teilchenwege ist im allgemeinen Fall viel zu aufwändig. In Abbildung 2.2 ist die Advektion eines masselosen Teilchens durch numerische Integration im Stömungsfeld dargestellt. Die neue Teilchenposition ist durch das Geschwindigkeitsfeld $\boldsymbol{V}$ an der Position des Teilchens und durch die Größe des Integrationsschritts Δt festgelegt. Da die Feldvektoren nur an diskreten Punkten vorliegen, müssen sie interpoliert werden.

Quelle	**Resultat**	**visualisierbare Felder und Eigenschaften**
Punkt (kont.)	Stromlinie, Streichlinie, Strompfad	laminare Strömungen, Teilchentrajektorie, lokale Feldrichtung
Punkt (diskret)	Stromteilchen	Feldrichtung und Stärke aus der Bewegung, auch geeignet für instationäre, turbulente Felder
Linie (kont.)	Strömungsflächen, Strömungsröhren	stationäre Felder, Feldstrukturen wie Divergenzen durch Verformungen
Objekte (diskret)	Stromobjekte	siehe Stromteilchen. Zusätzlich: Divergenzen, Expansion und Kompression durch Deformationen

Tabelle 2.1: Integrationsmethoden

An *Saatpunkten* (Partikelquellen) werden die Teilchen im Feld freigelassen. Abhängig von der Positionierung der Saatpunkte werden nur bestimmte Teilgebiete der Strömung und deren Eigenschaften erfasst. Im Gegensatz zu dichten Verfahren können deshalb wichtige Feldstrukturen übersehen werden, wenn sie nicht im visualisierten Bild vorhanden sind. Deshalb ist es sinnvoll, ein interaktives Erstellen und Verschieben der Saatquellen zu ermöglichen. Alternativ kann das Feld automatisch analysiert und geeignete Positionen berechnet werden.

Je nach Verteilung und Verbindung der Saatpunkte erhält man unterschiedliche Resultate (Tabelle 2.2). Visualisiert werden entweder die kontinuierlichen Wege, die von den Teilchen durchlaufen werden oder die Bewegung diskreter Teilchen, die mit Teilchen benachbarter Saatpunkte verbunden sein können. In der Praxis werden auch im kontinuierlichen Fall gepulst Teilchen vom Saatpunkt generiert und für die Darstellung miteinander verbunden. Instationäre Felder können nur über die flüchtige Animation der sich bewegenden Teilchen dargestellt werden.

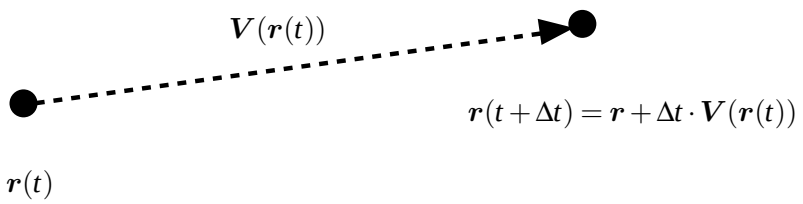


Abbildung 2.2: Integration eines Punktes entlang des Strömungsfeldes

2.2.1 Linien

Stromlinien sind Kurven im Raum, die an jedem Punkt tangential zum Vektorfeld verlaufen. Sie entsprechen der Trajektorie eines masselosen Teilchens in einem stationären Vektorfeld. In der Realität verändern sich Felder aber über die Zeit. *Pfadlinien* veranschaulichen die Teilchenwege in instationären Feldern. Beide Konzepte erfassen hauptsächlich die lokale Richtung des Feldes, wobei die Stärke des Feldes in der Farbe der Strömungslinien kodiert werden kann. Die Wahl des Saatpunktes ist für die Form der Linie von entscheidender Bedeutung. Mehrere Stromlinien, deren Saatpunkte an unterschiedlichen Positionen liegen, ermöglichen das Erfassen von allgemeineren Feldstrukturen. Dazu gehören

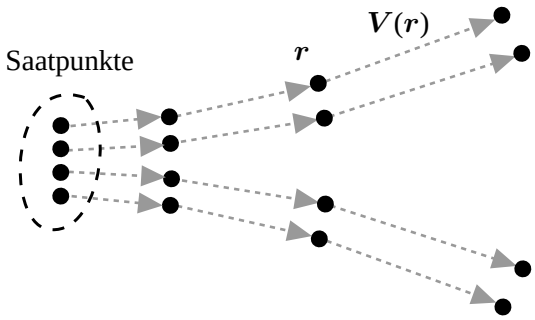


Abbildung 2.3: Stromlinien in der Nähe eines Sattelpunkts

Sattelpunkte, an denen die Linien auseinanderlaufen oder auch Attraktoren, an denen die Linien zusammenlaufen (Abb. 2.3). Der dreidimensionale Eindruck der Linien kann durch die Variation der Intensitäten, durch Stereodarstellungen oder Beleuchtungsberechnungen verbessert werden. Das Erzeugen gleichförmig verteilter Linien ist eine schwierige Aufgabe. Außerdem geht mit der Zahl der dargestellten Linien die Übersichtlichkeit verloren.

Es gibt noch weitere charakteristische Linien: *Streichlinien* werden bei instationären Strömungen verwendet. Sie verbinden mehrere Teilchen, die zeitlich nacheinander an einem festen Ort losgelassen werden. *Zeitlinien* veranschaulichen Divergenzen in der Strömung und sind mit Strömungsflächen verwandt. Sie bestehen aus einer Vielzahl von Teilchen, die sich an unterschiedlichen Orten befinden, und gleichzeitig im Feld freigelassen werden. In der Regel werden Zeitlinien beim Start orthogonal an der Strömung ausgerichtet.

2.2.2 Strömungsflächen

Strömungsflächen sind die natürliche Erweiterung von Stromlinien in die zweite Dimension. Statt eines punktförmigen Saatpunktes wird eine Saatlinie aus verbundenen Startpunkten gewählt. Aus dieser Linie heraus entsteht die Fläche, indem die Teilchenpositionen integriert werden. Ist die Linie geschlossen, dann erhält man *Strömungsröhren*. Gegenüber Stromlinien und Zeitlinien kann der 3D-Eindruck durch das Rendern einer beleuchteten und transparenten Fläche verbessert werden. Effekte wie Turbulenzen oder lokale Deformationen durch divergente Feldbereich können eine glatte Flächenstruktur schnell zerstören. Aufgrund der numerischen Berechnung der Teilchenpositionen, erhält man im Extremfall unbrauchbare Objekte, die bizarr verformt sind und sich sogar selbst durchdringen können. Deshalb ist sinnvoll, adaptiv zusätzliche Punkte in das Flächennetz einzufügen. Trotzdem sind Strömungsflächen hauptsächlich für stationäre und möglichst laminare Strömungen geeignet.

2.2.3 Stromteilchen und Stromobjekte

Die Dynamik einzelner, diskreter Teilchen und Objekte im Strömungsfeld wird animiert. Durch die Bewegung kann die Stärke und Richtung des Feldes auch bei turbulenten Feldern gut veranschaulicht werden.

Oberflächenpartikel kombinieren die Vorteile von Stromteilchen mit der Darstellung von Feldstrukturen

bei Stromflächen. Ein Normalenvektor wird als zusätzliches Attribut zu der Teilchenposition gespeichert und bei der Berechnung der Teilchenintensität während des Renderns verwendet. Der Vektor ist von den lokalen Feldvektoren abhängig und wird bei der Advektion aktualisiert. Wählt man die Startpositionen auf einer geschloßenen Kurve und läßt nacheinander eine große Zahl von Teilchen im Feld frei, dann können beleuchtete Stömungsflächen nachgebildet werden [5].

Streambubbles finden eine physikalische Entsprechung in Luftblasen, die in einer Flüssigkeitsströmung ausgesetzt sind. Modelliert werden die Streambubbles durch eine geschlossene Nurbsfläche, deren Kontrollpunkte sich frei im Feld bewegen. Durch die Bewegungen der Blasen und die Verformung ihrere Oberflächen, lassen sich verschiedene Feldeigenschaften visualisieren. Große Blasen veranschaulichen grob die Struktur des Gesamtfeldes, kleine Blasen hingegen zeigen lokale Eigenschaften an. Um die Feldeigenschaften auch in turbulenten Gebieten der Strömung gut erfassen zu können, werden die Blasen dort automatisch durch zwei kleinere ersetzt [6].

2.2.4 Geometrieadvektion

Die *Geometrieadvektion* ermöglicht die Visualisierung stationärer und instationärer Strömungen. Sie greift die Idee der Teilchenadvektion auf, wobei die punktförmigen Teilchen durch *geometrische Stromobjekte* ersetzt werden. Aufgebaut sind die Versuchsobjekte aus einzelnen Punktteilchen, die untereinander verbunden sind. Die Objekte bewegen sich im Geschwindigkeitsfeld der Strömung, wobei sie sich in divergenten Bereichen verformen. In homogenen Gebieten des Feldes, in denen die Strömung gleichförmig verläuft, bilden sich die Deformationen zurück. Diese Eigenschaft wird durch ein spezielles Körpermodell realisiert, das im nächsten Kapitel behandelt wird. Große, räumlich ausgedehnte Objekte können durch Dehnungen, Verkrümmungen oder Verdrillungen globale Feldstrukturen herausstellen. Lokale Eigenschaften des Feldes werden durch kleine Objekte oder durch lokal beschränkte Deformationen der Geometrie erfasst. Gleichzeitig ist es möglich, die Feldstärke und die Richtung, wie bei den Stromteilchen, durch die Animation der Objektbewegungen zu veranschaulichen. Die Dynamik der Objekte wird mit Hilfe einer physikalisch motivierten Simulation numerisch berechnet. Über das zugrunde liegende physikalische Modell ist den Objekten eine Masse zugeordnet. So können Effekte wie Trägheit und Reibung realisiert werden, die bei realen Objekten in Strömungen auftreten. Das Resultat der Visualisierung hängt von der Form, der Anzahl und den initialen Positionen der Versuchsobjekte ab. Die Objekte können interaktiv von Hand platziert werden oder sie werden an Saatpunkten generiert. Stromlinien, Strompfade und Stromflächen lassen sich durch eine große Menge von Objekten imitieren, die an festen Saatpunkten in kurzen Zeitabständen freigelassen werden. Auch Strömungsproben können realisiert werden, indem den Objekten zusätzliche Zwangsbedingungen aufgeprägt werden. Die Positionen eines oder mehrer Objektpunkte werden hierbei im Raum fixiert. Das Objekt als Ganzes kann sich deshalb nicht mehr fortbewegen, die freien Punkte allerdings bewegen sich weiterhin in der Strömung und zeigen so die lokalen Feldeigenschaften an.

2.3 Zusammenfassung

Das Kapitel hat die grundlegenden Verfahren der Strömungsvisualisierung vorgestellt. Einen umfassenden Überblick über dieses Teilgebiet der wissenschaftlichen Visualisierung findet sich zum Beispiel in [7]. Abschließend bleibt zu bemerken, dass es kein Verfahren gibt, das in allen Situationen optimale

Ergebnisse liefert. Vielmehr erlaubt es die Kombination aus einer Vielzahl von Untersuchungsmethoden, die Eigenschaften des Feldes besser zu verstehen. Eine einfache, intuitive und möglichst interaktive Darstellung kann dabei sehr hilfreich sein.

3 Physik und Numerik

3.1 Einleitung

Die physikalisch motivierte Visualisierung von Objektbewegungen in Strömungsfeldern erfordert die Berechnung von Positionen und Geschwindigkeiten der betrachteten Versuchsobjekte. Zuerst werden am Beispiel des Punktmassenmodells die physikalischen Grundlagen besprochen. Die Physik deformierbarer Körper ist ausgesprochen komplex. Eine Simulation der Dynamik in Echtzeit ist deshalb nur mit vereinfachten Modellen realisierbar. Aufbauend auf einem Modell für starre Körper wird das hier verwendete Feder-Masse-Modell erklärt. Der letzte Abschnitt behandelt numerische Lösungsverfahren.

3.2 Physik

3.2.1 Grundlagen der klassischen Mechanik

In der klassischen Mechanik werden Teilchen häufig als Punktmassen ohne räumliche Ausdehnung approximiert. In einem kartesischen Koordinatensystem ist die zeitabhängige Position eines Teilchens durch einen Vektor $\boldsymbol{r}(t)$ gegeben,

$$\boldsymbol{r}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \tag{3.1}$$

Die Geschwindigkeit $\boldsymbol{v}(t)$,

$$\boldsymbol{v}(t) = \dot{\boldsymbol{r}}(t) = \begin{pmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{z}(t) \end{pmatrix} \tag{3.2}$$

und die Beschleunigung $\boldsymbol{a}(t)$,

$$\boldsymbol{a}(t) = \dot{\boldsymbol{v}}(t) = \ddot{\boldsymbol{r}}(t) \tag{3.3}$$

erhält man als erste beziehungsweise zweite Zeitableitung von $\boldsymbol{r}(t)$. Ist die Konfiguation $\boldsymbol{r}(t), \boldsymbol{v}(t)$ zu einem gegebenen Zeitpunkt bekannt, dann bestimmt sie den Zustand des Systems vollständig und zukünftige Positionen und Geschwindigkeiten können vorhergesagt werden. Gleichungen, die die Dynamik eines Systems bestimmen, werden Bewegungsgleichungen genannt. Es handelt sich um Differentialgleichungen zweiter Ordnung in $\boldsymbol{r}(t)$. Sie verknüpfen die Koordinaten, die Geschwindigkeiten und Beschleunigungen sowie die äußeren Kräfte $\boldsymbol{F}(\boldsymbol{r}$,t), die auf die Teilchen wirken.

$$\frac{d^2\boldsymbol{r}}{dt^2}(t) = \boldsymbol{a}(t) = \frac{\boldsymbol{F}(\boldsymbol{r},t)}{m} \tag{3.4}$$

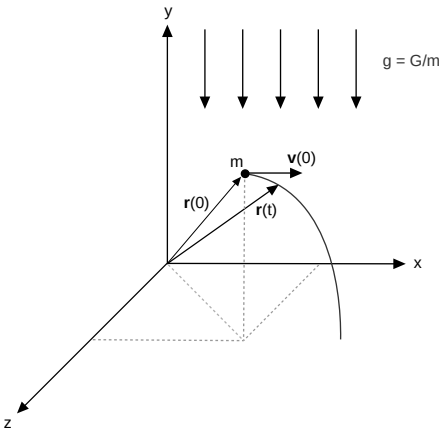


Abbildung 3.1: Teilchentrajektorie unter dem Einfluss der Gravitation

Im allgemeinen Fall eines freien Teilchens ohne äußere Krafteinwirkung, d.h. $\boldsymbol{F}(\boldsymbol{r},t) = 0$, erhält man die Lösung durch zweimaliges Integrieren und der Angabe der Startbedingungen $\boldsymbol{r}_0 = \boldsymbol{r}(t_0)$. In Abbildung 3.1 ist der freie Wurf eines Teilchens in Richtung der x-Achse skizziert. Unter dem Einfluss der zeitunabhängigen Gewichtskraft G durchläuft das Teilchen eine parabelförmige Trajektorie.

Analytisches Lösen der Bewegungsgleichung ist nur in den einfachsten Fällen möglich. Schon die Lösung eines einfachen N-Teilchen Systems kann in der Regel nur mit numerischen Methoden erreicht werden, da sich durch die Wechselwirkungen jedes Teilchens mit den übrigen N-1 Teilchen sehr komplexe Beziehungen ergeben.

Die Korrektheit des Punktmassenmodells hängt von der zu lösenden Aufgabenstellung ab. Beispielsweise könnte man einen Fußball als Punktmasse betrachten. Für die Dynamik des Balls relevante Effekte wie Verformungen oder Rotationen werden dabei allerdings vernachlässigt.

3.2.2 Starre Körper

Sollen räumlich ausgedehnte Körper und deren Rotation um die eigene Achse oder ihren Schwerpunkt betrachtet werden, dann bietet sich das Modell des starren Körpers an [8]. Dieser läßt sich als eine Menge von Punktmassen definieren, deren Abstand untereinander konstant ist. Realistischer ist allerdings eine Beschreibung durch eine kontinuierliche Masseverteilungen. Da diese über die Massendichte $\rho(\boldsymbol{r})$ festliegt und sich nicht ändern darf, werden plastische Deformationen von diesem physikalischen Modell nicht erfasst.

Dem starren Körper wird ein eigenes, lokales Koordinatensystem zugeordnet, dessen Ursprung sich in seinem Schwerpunkt befindet. Die Gesamtmasse M und die Position $\boldsymbol{s}$ des Schwerpunktes kann durch eine Integration über das Volumen des Körpers bestimmt werden.

$$M = \int \rho(\boldsymbol{r}) dV \tag{3.5}$$

$$\boldsymbol{s} = \int \boldsymbol{r} \cdot \rho(\overline{\boldsymbol{r}}) dV / M \tag{3.6}$$

Das System eines einzelnen starren Körpers besitzt im räumlichen Fall sechs Freiheitsgrade. Die Translationsbewegung legt die Dynamik des Schwerpunkts fest, der als Punktteilchen der Masse M betrachtet wird. Zusätzlich kann der Körper eine Rotationsbewegung um den Schwerpunkt ausführen. Der Körper in Abbildung 3.2 bewegt sich aufgrund der drei wirkenden Kräfte in Richtung der x-Achse und dreht sich dabei gleichzeitig um $\boldsymbol{S}$.

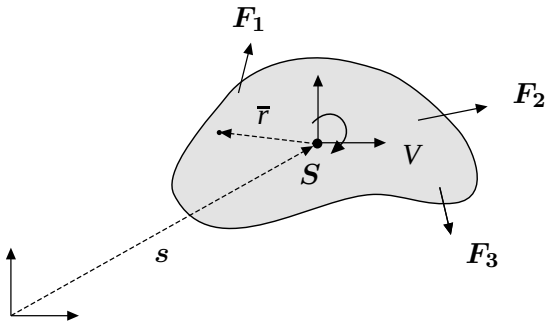


Abbildung 3.2: Ein starrer Körper mit einer kontinuierlichen Masseverteilung. Die drei angreifenden Kräft bewirken eine Translation des Schwerpunkts und eine Rotation des Körpers um den Schwerpunkt.

3.2.3 Feder-Masse-Modell

Eine Vereinfachung von realen Objekten findet beim Feder-Masse-Modell Anwendung. Auf atomarer Ebene bestehen Festkörper aus Atomen und Molekülen, die auf ihren Gitterplätzen beschränkt schwingen können. Die verschiedenartigen Bindungen der Atome untereinander und die dabei auftretenden Kräfte werden durch Federn modelliert. In der Realität wechselwirken alle atomaren Teilchen miteinander, wobei die Stärke mit zunehmender Entfernung rasch abfällt. Im Feder-Masse-Modell dagegen beschränken sich die Wechselwirkungen auf direkt benachbarte Massepunkte. Unterschiedliche Materialeigenschaften können durch eine geeignete Wahl der Massen und Federeigenschaften realisiert werden. Inhomogene Körper können so vergleichsweise leicht realisiert werden. Eine geschickte Wahl der Massepunkte und der Federn ermöglicht dabei deformierbare Körper, die ihre charakteristische Grundform im kräftefreien Fall zurückerlangen.

Federgleichung

Im Ruhezustand wirken keine Kräfte und die Feder hat eine vorgegebene Länge. Diese wird im Folgenden als Ruhelänge bezeichnet. Wird die Feder unter Arbeitsaufwand komprimiert oder gestreckt, so resultiert an beiden Enden eine Rückstellkraft. Die *Hookesche-Federgleichung* (3.7) besagt, dass bei einer idealen Feder die Rückstellkraft proportional zur Auslenkung ist.

$$F_{rueck} = D * \frac{\text{Länge} - \text{Ruhelänge}}{\text{Ruhelänge}} \tag{3.7}$$

Die Federhärte oder auch Federsteifigkeit D geht als zusätzlicher Faktor in die Gleichung ein und bestimmt dadurch das Verhalten der Feder. Federn mit einer kleinen Federhärte lassen sich leichter kom-

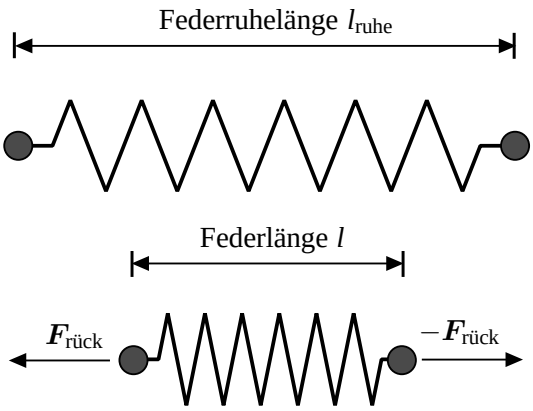


Abbildung 3.3: Feder im Ruhezustand (oben), komprimierte Feder mit auftretenden Rückstellkräften (unten)

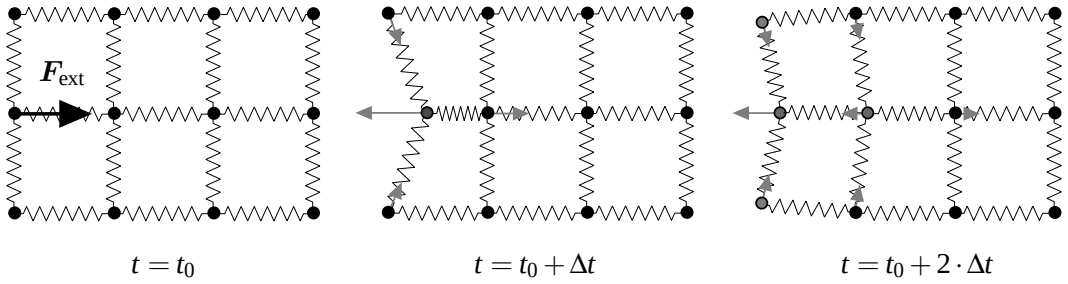


Abbildung 3.4: Verhalten eines Feder-Masse-Systems bei äußerer Krafteinwirkung

primieren. Hohe Federhärten erfordern für die gleiche Auslenkung eine entsprechend größere Arbeit. Die dabei aufgewendete Energie steckt als potentielle Energie in der Feder. Beim Zurückformen in den Ursprungszustand wandelt sie sich in kinetische Energie der angekoppelten Teilchen um.

Systemdynamik

Physikalische Systeme sind immer bestrebt einen Zustand möglichst niedriger potentieller Energie zu erreichen. Im Fall des Feder-Masse-Systems setzt sich die Gesamtenergie des Systems aus der kinetischen Energie der Teilchen sowie der gespeicherten potentiellen Federenergie zusammen. Äußere Kräfte bringen das System aus dem Gleichgewicht. Die dabei auftretenden Federkräfte wirken den externen Kräften entgegen und versuchen diese zu kompensieren. Das Erreichen eines Gleichgewichtzustandes erfolgt dabei immer dynamisch.

Eine äußere Kraft oder ein Impuls werden zunächst nur auf die Federn am Rand des Objektes wirken. Der Randbereich propagiert dann die in den Federn gespeicherte Energie ins Innere (Abb. 3.4). Wäre der Körper in Abbildung 3.2 durch Punktteilchen modelliert, die über Federn verbunden sind, dann würde er sich zusätzlich ausdehnen.

Bewegungsgleichungen

Im Allgemeinen besteht ein Feder-Masse-System aus N Masseteilchen, welchen eine Position $\boldsymbol{r}_i$ und eine Masse m_i zugeordnet ist ($i \in 1..N$). Das i-te Teilchen ist über M_i Federn der Länge $\mid \boldsymbol{l}_{ij} \mid = \mid \boldsymbol{r}_j - \boldsymbol{r}_i \mid$ mit seinen nächsten Nachbarn verbunden ($j \in 1..M_i$), wobei die Anzahl der Nachbarn variieren kann. Im Ruhezustand entspricht die Länge der Ruhelänge $\boldsymbol{L}_{ij}$. Die gesamte auf ein Teilchen wirkende Federkraft $\boldsymbol{F}_i$ setzt sich aus der Summe der Federkräfte $\boldsymbol{F}_{ij}$ zusammen.

$$\boldsymbol{F}_{ij} = D_{ij} \frac{\mid \boldsymbol{l}_{ij} \mid - \mid \boldsymbol{L}_{ij} \mid}{\mid \boldsymbol{L}_{ij} \mid} \cdot \boldsymbol{L}_{ij} \tag{3.8}$$

Bei der folgenden Formulierung der Bewegungsgleichungen des Gesamtsystems wird die Zeitabhängigkeit nicht explizit angegeben. In das System geht ausser den externen Kräften $\boldsymbol{F}_{\text{ext}}$ und den entgegenwirkenden Federkräften ein zusätzlicher dämpfender Term ein. Der Faktor c bestimmt dabei die Stärke der Dämpfung und unterbindet ein unkontrolliertes Schwingen der Federn. Für die einzelnen Teilchen gilt:

$$m_i \cdot \ddot{\boldsymbol{r}}_i + \underbrace{c \cdot \dot{\boldsymbol{r}}_i}_{\text{Dämpfung}} = \boldsymbol{F}_i^{ext} - \sum_{j=1}^{M_i} \boldsymbol{F}_{ij} \qquad i \in 1..N \tag{3.9}$$

Insgesamt erhält man ein gekoppeltes, nichtlineares Gleichungssystem für die Teilchenpositionen $\boldsymbol{r}_i$. Das System besteht aus N gewöhnlichen Differentialgleichungen zweiter Ordnung. Da die einzelnen Gleichungen über die Federkräft $\boldsymbol{F}_{ij}$ gekoppelt sind, ist das System schon bei kleinen N sehr komplex.

3.2.4 Kopplung an das Strömungsfeld

Bei der Geometrieadvektion sollen sich die Objekte, die durch das Feder-Masse-Modell beschrieben werden, in der Strömung fortbewegen. Deshalb ist eine Kopplung an die Geschwindigkeitsvektoren des Strömungsfeldes $\boldsymbol{V}$ nötig. Das *Gesetz von Stokes* legt die Stärke der Reibungskraft fest, die kugelförmige Körper mit der Geschwindigkeit v und dem Radius r in einem ruhenden Fluid (Flüssigkeiten oder Gase) erfahren. Die Zähflüssigkeit des Fluids wird dabei durch die Viskosität η beschrieben.

$$F = 6\pi\eta r\boldsymbol{v} \tag{3.10}$$

In unserem Fall befindet sich das Medium nicht in Ruhe, sondern seine Geschwindigkeit ist an jedem Punkt $\boldsymbol{r}$ und zu jedem Zeitpunkt t durch das Strömungsfeld $\boldsymbol{V}(\boldsymbol{r},\text{t})$ gegeben. Wird die relative Geschwindigkeit der Masseteilchen zur Strömung verwendet, dann stellt sich der gewünschte Effekt ein. Ist das Teilchen schneller, als das Feld, dann wird es abgebremst. Ist es langsamer, dann wird es beschleunigt. Die Kopplungsgleichung lautet somit:

$$\boldsymbol{F}(\boldsymbol{r},t) = 6\pi\eta r \cdot [\boldsymbol{V}(\boldsymbol{r},t) - \boldsymbol{v}(t)] \tag{3.11}$$

3.3 Numerische Verfahren für gewöhnliche Differentialgleichungen

Besonders die zeitlichen Anforderungen an Echtzeitsimulationen schliessen analytisches Lösen der Differentialgleichung von vornherein aus. Erst numerische Verfahren ermöglichen es, komplexe Feder-Masse-Syteme aus mehreren hunderttausend Federn schnell genug zu lösen [9].

3.3.1 Gewöhnliche Differentialgleichungen

Gewöhnliche Differentialgleichungen n-ter Ordnung lassen sich durch Variablensubstitution immer in ein gekoppeltes System von Differentialgleichungen erster Ordnung umformen. Als Resultat erhält man n Gleichungen:

$$\frac{dy_1}{dt} = y_2 \quad \ldots \quad \frac{dy_{n-1}}{dt} = y_n \quad , \quad \frac{dy_n}{dt} = f(t, y_1, \ldots, y_n) \tag{3.12}$$

Beispielsweise führt man für die N Bewegungsgleichungen des Feder-Masse-Systems eine neue Variable $\boldsymbol{v}$ ein und erhält somit 2N neue Gleichungen:

$$\dot{\boldsymbol{r}}_i = \boldsymbol{v}_i \tag{3.13}$$

$$\dot{\boldsymbol{v}}_i = (\boldsymbol{F}_i^{ext} - \sum_{j=1}^{M_i} \boldsymbol{F}_{ij} - c\boldsymbol{v}_i)/m_i \tag{3.14}$$

Allein durch die Differentialgleichung ist das Problem nicht vollständig spezifiziert. Bei Anfangswertproblemen werden zu einem festgelegten Startzeitpunkt t_0 zusätzlich Startwerte für alle $\boldsymbol{r}_i$ angegeben. Randwertprobleme hingegen verteilen die n vorgegebenen Werte auf mehrere Zeitpunkte. Statt die Werte explizit anzugeben, kann das System auch durch zusätzliche algebraische Gleichungen vervollständigt werden. Bei unserem Feder-Masse-Systems handelt es sich um ein Anfangswertpoblem:

$$\boldsymbol{r}_i(t_0) = \boldsymbol{r}_i^0 \tag{3.15}$$

$$\boldsymbol{v}_i(t_0) = \boldsymbol{v}_i^0 \qquad i = 1..N \tag{3.16}$$

3.3.2 Numerische Lösung

Für die Simulation physikalischer Systeme müssen die Daten in eine geeignete diskrete Form gebracht werden. In unserem Fall ist dies in der räumlichen Dimension nicht nötig, da das zugrundeliegende Objektmodell schon diskret vorliegen. Neben der Raumdimension muss auch die zeitliche Dimension diskretisiert werden. Im einfachsten Fall wird eine konstante Schrittweite Δt vorgegeben. Raffiniertere Verfahren passen ihre *Schrittweite* dem aktuellen Systemzustand an und können so numerische Probleme vermeiden oder an besonders einfachen Stellen durch große Schritte Zeit einsparen. Die adaptive Wahl ist vor allem bei hochdimensionalen System eher problematisch. Auf solche Optimierungen wird deshalb nicht weiter eingegangen.

Numerische Verfahren werden nach ihrer *Genauigkeit* im Vergleich zur korrekten Lösung beurteilt. Ausserdem verdient ihre *numerische Stabilität* besondere Aufmerksamkeit. Numerisch instabile Algorithmen können dazu führen, dass das untersuchte System eine Reihe von extremen, nichtvorhersagbaren und chaotischen Zustandsänderungen durchläuft. Die Ergebnisse einer solchen Berechnung sind in der Regel unbrauchbar.

3.3.2.1 Explizite Verfahren

Explizite Verfahren sind nicht unter allen Konfigurationen stabil. Der Benutzer muss deshalb bei der Simulation aufpassen und dafür sorgen, dass die *Courant-Bedingung* für die maximal mögliche Schrittweite eingehalten wird, welche von der Diskretisierung des Raumes abhängt. Explizit angeben läßt sich eine solche Bedingung allerdings nur bei einfachen Modellbeispielen.

Die einfachsten expliziten Verfahren basieren auf der Taylorentwicklung um den Funktionswert $\boldsymbol{r}$ zur Zeit t:

$$\boldsymbol{r}(t \pm \delta t) = \boldsymbol{r}(t) \pm \dot{\boldsymbol{r}}(t) \cdot \delta t + (\ddot{\boldsymbol{r}}(t) \cdot \delta t^2)/2 + \sum_{k=3}^{\infty} (\boldsymbol{r}^{(k)} \cdot \delta t^k \cdot (\pm 1)^k)/k! \tag{3.17}$$

Das *Explizite Euler-Verfahren* approximiert den Zustand des neuen Zeitpunkts, indem es eine diskrete Schrittweite Δt einführt und Terme der Entwicklung ab einer Ordnung von zwei ignoriert.

$$\boldsymbol{r}_{t \pm \Delta t} = \boldsymbol{r}_t \pm \Delta t \cdot \dot{\boldsymbol{r}}_t \tag{3.18}$$

Gelöst wird das Anfangswertproblem, indem die Werte des aktuellen Zeitschritts t zum neuen Zeitpunkt $t + \Delta t$ 'integriert' werden. Um ein größeres Zeitintervall zu durchschreiten, wird die Gleichung iterativ auf den jeweils aktuellen Zustand $\boldsymbol{r}_t$ angewandt. Soll ein gekoppeltes Gleichungssystem aus mehrern Differntialgleichungen erster Ordnung gelöst werden, dann ist es nötig, jede einzelne dieser Gleichungen zu integrieren. Beispielsweise sind es beim Feder-Masse-System pro Schritt und Teilchen zwei Gleichungen (siehe 3.13 und 3.14). Addiert man die zwei Näherungen zweiter Ordnung, so erhält man das etwas bessere *Verlet-Verfahren*. Der Vorteil dieser Methode ist, dass die Geschwindigkeiten nicht explizit integriert werden müssen. Allerdings werden bei der Integration die Positionen der beiden letzten Zeitschritte benötigt:

$$\boldsymbol{r}_{t+\Delta t} = 2 \cdot \boldsymbol{r}_t - \boldsymbol{r}_{t-\Delta t} + \ddot{\boldsymbol{r}}_t \cdot \Delta t^2 \tag{3.19}$$

Methoden höherer Ordnung, wie das *Runge-Kutta-Verfahren* vierter Ordnung (RK4) bieten eine größere Genauigkeit. Pro Zeitschritt werden die rechten Seiten des Systems an mehreren Stellen mit Hilfe von Eulerschritten ausgewertet. Der neue Systemzustand ergibt sich aus einem gewichteten Mittel der Zwischenergebnisse. Bei RK4 gehen in die Integration zusätzlich zwei Auswertungen zur Zeit $t + \Delta t/2$ sowie eine Auswertung zur Zeit $t + \Delta t$ ein (Abb 3.5). In der Praxis gilt RK4 als geeigneter Kompromiss aus Aufwand und Genauigkeit und wird deshalb oft im Zusammenspiel mit einer adaptiven Schrittweitenanpassung eingesetzt.

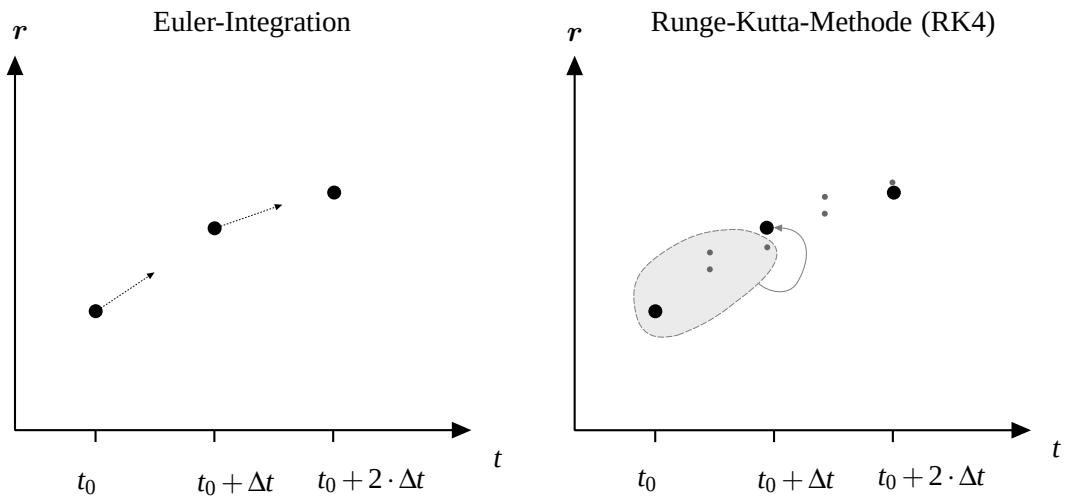


Abbildung 3.5: Die kleinen Punkte verdeutlichen die zusätzlichen Stellen, an denen die rechten Seiten der Entwicklungsgleichungen ausgewertet werden. Die großen Punkte entsprechen den integrierten Werten des jeweiligen Zeitschritts.

3.3.2.2 Implizite Verfahren

In der Gleichung des *Impliziten-Euler-Verfahrens* erscheinen sowohl auf der linken, als auch auf der rechten Seite die gesuchten Werte des neuen Zeitpunkts.

$$\boldsymbol{r}_{t+\Delta t} = \boldsymbol{r}_t + \Delta t \cdot f(\boldsymbol{r}_{t+\Delta t}, r) \tag{3.20}$$

Da die Werte $\boldsymbol{r}_{t+\Delta t}$ noch nicht bekannt sind, kann die Gleichnung nicht einfach integriert werden. Ist eine größere Menge solcher Differentialgleichungen gekoppelt, dann muss pro Zeitschritt ein kompliziertes, nichtlineares Gleichungssystem gelöst werden. Dieser zusätzlich Aufwand kann sich aber lohnen, da die impliziten Methoden numerisch stabil sind und deshalb eine gößere Schrittweite Δt als bei den expliziten Methoden gewählt werden kann. Aufgrund der Komplexität dieser Lösungsverfahren werden hier keine weiteren Details angegeben.

3.4 Zusammenfassung

Es wurden die physikalischen Grundlagen und das numerische Rüstzeug für die Simulation eines Feder-Masse-Systems vorgestellt. Die Entscheidung das einfache *Verlet-Verfahren* zu implementieren wird im nächsten Kapitel über *GPGPU*-Anwendungen näher begründet.

4 Numerische Simulation mit der GPU

4.1 Einführung

In den letzten Jahren hat sich eine neue Art von Hochleistungsprozessoren für Grafikanwendungen etabliert. Angetrieben durch den mittlerweile milliardenschweren Markt für Computerspiele wurde und wird die Technik der *GPUs* (Graphics Processing Units) in relativ kurzen Produktzyklen immer weiter verbessert. Aus einfachen 2D-Grafikkarten haben sich nach und nach Grafikchips entwickelt, die komplexe 3D-Szenen flüssig darstellen können. Neben der Spieleentwicklung profitiert insbesondere auch die Visualisierung von wissenschaftlichen Daten und die technische Konstruktion (CAD) von den neuen Möglichkeiten. Aufgrund der großen Nachfrage und ihrer weiten Verbreitung werden die 3D-Grafikkarten zu erschwinglichen Preisen angeboten. Ihre Architektur wurde speziell auf die Anforderungen der *Grafikpipeline* [10] zugeschnitten, wobei die Leistungsfähigkeit auf ihrem Spezialgebiet die üblicher CPUs um ein Vielfaches übersteigt. Mittlerweile sind die GPUs teilweise programmierbar. Dies ermöglicht den Programmierern, immer realistischere Materialien nachzubilden und raffiniertere Effekte in ihre neuesten Spiele zu integrieren. Durch die hinzugekommene Flexibiliät und Aufgrund des guten Preis-Leistungsverhältnisses sind 3D-Grafikkarten inzwischen auch für Nichtgrafikanwendungen von Interesse (*General Purpose Computing, GPGPU*).

Zunächst wird im Abschnitt *'Stream Computing'* das der GPU zugrundeliegende Rechenparadigma vorgestellt. In dem folgenden Unterkapitel wird dann die Grafikpipline und ihre Implementation in der Hardware besprochen. Auch einige interessante GPU-Features werden erläutert. Das Kapitel *'GPU Programmierung'* beschäftigt sich mit den zur Verfügung stehenden Programmiersprachen. *'General Purpose Computing mit der GPU'* spezifiziert eine Klasse von Algorithmen, die sich für eine Implementation auf der GPU besonders eignen und erörtert die dabei auftretenden Problemstellungen. Zuletzt wird insbesondere die numerische Simulation auf der GPU am Beispiel des Feder-Masse-Modells ausführlich beschrieben.

4.2 Stream Computing

Heutige Prozessoren, die in üblichen Arbeitsplatzrechnern eingesetzt werden, basieren auf der seriellen *Von-Neumann-Architektur*. Es handelt sich meist um SISD-CPUs (Single Instruction Single Data), die keine starke Parallelverarbeitung von Daten vorsieht. Der Prozessorkern besteht aus Steuereinheiten und ALUs, die über einen Speicherbus mit Daten und Anweisungen versorgt werden. Der Transport der Daten aus dem Hauptspeicher zur CPU ist relativ langsam. Deshalb werden Daten in einer mehrstufigen Cachearchitektur direkt im CPU-Kern verwaltet. Dies kann die Zugriffszeit bei mehrmaliger Benutzung ein und desselben Datenelements drastisch reduzieren. Ihr allgemeiner Aufbau macht die CPUs sehr flexibel und für eine Vielzahl von Problemen einsetzbar. Gleichzeitig bedeutet es aber auch, dass ein großer Teil der Transistoren für Caches und Steuerschaltungen statt für Recheneinheiten verwendet wird.

Berechnungen, bei denen auf den gleichartigen Datenelemente fortlaufender Datenströme (*Streams*) immer wieder die selben Operationen angewandt werden, können von den großen Caches nicht profitieren.

Hier kommt es vielmehr auf einen möglichst hohen Durchsatz an. Sind die Berechnungen auf den einzelnen Stromelementen zusätzlich voneinander unabhängig, so könnten sie theoretisch parallel ausgeführt werden. *Stream Computing* ist ein Rechenkonzept, welches speziell für solche Problemstellungen optimiert ist. *Rechenkernel* (Abb. 4.1) führen vordefinierte Operationen auf den Eingabeströmen durch und produzieren neue Ausgabeströme. Durch das Replizieren eines Kernels steigt der Datendurchsatz beinahe linear mit der Anzahl der Kopien an (Parallelität auf Datenebene). Komplexere Aufgaben lassen sich durch das Hintereinanderschalten mehrerer Kernels realisieren (Fliessbandprinzip). *Pipelining* erlaubt dabei das gleichzeitige Verarbeiten aufeinanderfolgender Stromelemente in den hintereinandergeschalteten Kernels (Parallelität auf Task-Ebene). Um Wartezeiten (Pipeline Stalls) zwischen den Stufen zu vermeiden, sollten diese besonders gut aufeinander abgestimmt werden. Zusätzlich können elementare Operationen, wie zum Beispiel Vektoraddition auf niedriger Ebene parallelisiert werden (Parallelität auf Anweisungsebene).

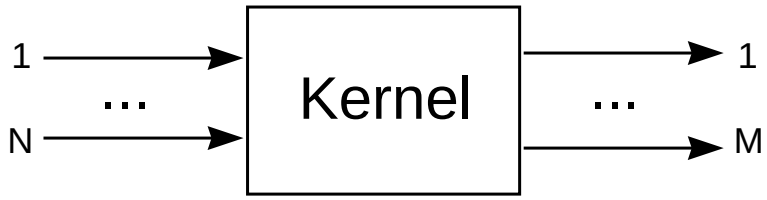


Abbildung 4.1: Stream Computing Kernel

Aufgrund der festgelegten Aufgaben der einzelnen Rechenkernel kann die Steuerhardware relativ einfach ausfallen. Statt großer Caches werden nur kleine Bufferspeicher zwischen den verschiedenen Stufen der Berechnung benötigt. *Stream Computing* erlaubt deshalb eine hochgradig effiziente und parallele Verarbeitung von Daten. Der Performancegewinn gegenüber allgemeinen Prozessoren ist dementsprechend hoch.

4.3 GPUs

Die Architektur des GPU-Kerns ist am Stream-Computing-Modell orientiert und implementiert die Grafikpipeline für dreidimensionale Szenen in Hardware. Die Grafikpipeline beschreibt den Weg einer Szene vom virtuellen 3D-Modell bis zur zweidimensionalen Ausgabe auf dem Bildschirm bzw. im Framebuffer. Die verschiedenen Kernels und Streams sind in Abbildung 4.2 skizziert. Zunächst werden die einzelnen Objektvertices (*Vertexstrom*), die in Modellkoordinaten vorliegen, transformiert und in den Bildraum projiziert. Bei programmierbaren GPUs wird diese Funktionalität, neben anderen, in den Vertexeinheiten realisiert. Aus dem Strom der transformierten Vertices und deren Nachbarschaftsbeziehungen wird nun eine Folge von Dreiecksprimitiven erstellt (*Dreiecksstrom*). Die Dreiecke durchlaufen daraufhin eine Reihe von Tests, wobei sie verworfen werden, wenn sie komplett außerhalb des Darstellungsvolumens liegen (*Culling*), oder wenn sie von anderen Primitiven verdeckt werden (*Early Z-Test*). Primitve, die den Test überstanden haben werden anschließend rasterisiert. Die programmierbaren Fragmenteinheiten führen auf den Elementen des resultierenden Fragmentstroms eine Reihe von Operationen aus. Als Ergebnis der Berechnung erhält man die Farbe des Fragments. Diese hängt in der Regel von den Materialeigenschaften der zugrundeliegenden Objektoberfläche und den Lichtern der Szene ab. Die

bearbeiteten Fragmente werden daraufhin in das Renderziel geschrieben. Dies kann der Framebuffer, aber auch eine Textur sein. Die *Blending-Hardware* erlaubt es dabei, die vorherigen Pixelinhalte mit den neuen Fragmentfarben über verschiedene Operationen zu verknüpfen. In der Regel ist die Zahl der Fragmente nach dem Rasterisieren sehr viel größer als die Zahl der Eingabevertices. Deshalb sind normalerweise mehr Fragmenteinheiten als Rastereinheiten vorhanden. Die einzelnen Einheiten arbeiten parallel und verfügen über eigene Register und kleine Caches.

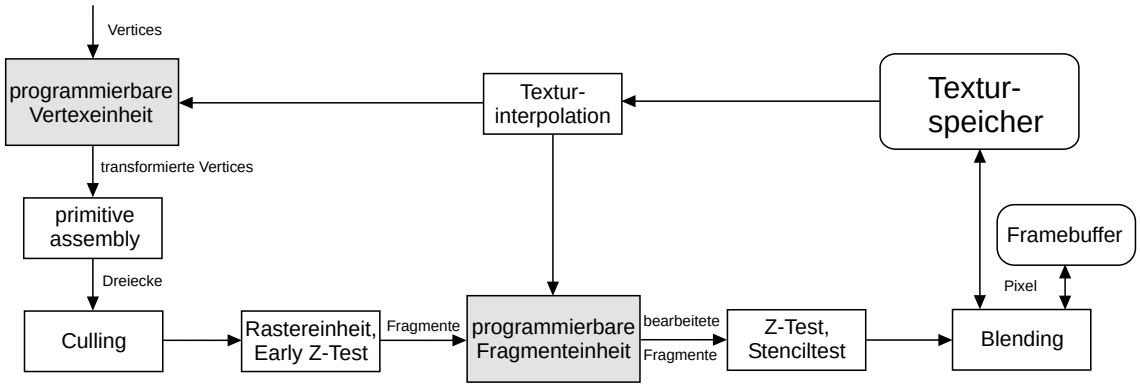


Abbildung 4.2: Die 3D Grafikpipeline in der GPU Implementation

4.3.1 GPU Hardware

GPUs enthalten zwei unterschiedliche Typen von programmierbaren Einheiten und eine Reihe von festeingebauten Funktionen. Dieser Abschnitt geht auf die für die GPGPU-Programmierung relevanten Merkmale und Restriktionen der Grafikhardware ein. Die folgenden Angaben beziehen sich auf die NVidia Geforce 6800.

Texturierung

Texturmapping war einer der ersten Aufgaben, die Grafikkarten übernommen haben. Mittlerweile werden neben 2D-Texturen und Cube-Maps auch 3D-Texturen unterstützt. Pro Texel stehen bis zu vier Komponenten zur Verfügung, die in einem Floating-Point-Format gespeichert werden können. Im fp16-Format ist es möglich, verschiedene Texturfiltermethoden in Hardware auszuführen. Schnelle Filterung von fp32-Texturen beherrscht die Hardware allerdings noch nicht.

(early) Z-Test

Beim Rastern größerer Mengen von Primitiven kommt es häufig vor, dass sich Fragmente gegenseitig verdecken, da sie in der Bildausgabe auf das gleiche Pixel fallen. Deshalb werden im Tiefenpuffer die relativen Tiefenwerte der gerenderten Pixel gespeichert. Basierend auf dem Tiefenpuffereintrag kann der *Z-Test* entscheiden, ob das aktuelle Fragment überhaupt sichtbar ist, und ob es verworfen werden soll. Jedoch wird die Entscheidung erst nach der Ausführung des Fragmentprogramms getroffen. Um

unnötige Aufrufe des Programms zu vermeiden, versucht der *Early Z-Test* schon vorher die Fragmente zu verwerfen. Allerdings ist das Testen jedes einzelnen Fragments in dieser frühen Stufe der Pipeline zu zeitaufwendig, deshalb werden größere, zusammenhängende Fragmentblöcke auf einmal getestet.

Occlusion Query

Über *Occlusion Querys* kann die Anzahl der Fragmente abgefragt werden, die den Z-Test und den *Stenciltest* bestanden haben. Der Stenciltest wird in dieser Arbeit nicht verwendet.

Blending

Am Ende der Rendering-Pipeline werden die Fragmente mit dem schon vorhandenen Framebufferinhalt über verschiedene Operationen verknüpft. Fragmente im fp32-Format werden von der Blending-Hardware nicht unterstüzt.

Programmierbare Einheiten

Aktuelle Hardware erlaubt den Zugriff auf Texturdaten sowohl im Vertex- als auch im Fragmentprogramm. Jedoch ist eine Texturfilterung bei Vertexprogrammen nicht in Hardware möglich. Generell sind *Texturlookups* verhältnismäßig zeitaufwendige Operationen und sollten möglichst durch gleichzeitig laufende Berechnungen kaschiert werden. Es gilt mehrstufig abhängige Texturabfragen sowohl im Vertex- als auch im Fragmentprogramm zu vermeiden. Auch Dynamische Flusskontrolle, wie Verzweigungen und Schleifen, werden von der Hardware unterstützt. Dabei wird der Programmzweig jedes Vertex unabhängig von den anderen Vertices ausgeführt. Im Fragmentprogramm hingegen müssen Operationen immer auf einer größeren Menge von Fragmenten ausgeführt werden. Deshalb kann es passieren, dass beide Zweige einer if-Anweisung ausgewertet werden und danach entschieden wird, welches Ergebnis verwendet wird. Vor allem Schleifen, deren Anzahl von Durchläufen von Fragment zu Fragment stark variiert, sollten deshalb vermieden werden. Die Resultate der Berechnungen des Fragmentprogramms können in bis zu vier Rendertargets geschrieben werden.

4.3.2 GPU Programmierung

Anfangs konnten die Fragment- und Vertexeinheiten der GPU nur mit einer maschinennahen Assemblersprache programmiert werden. Mittlerweile gibt es eine Reihe von höheren Programmiersprachen wie Cg, HLSL und GLSL. Die C-nahe Shadersprache GLSL (OpenGL Shading Language) ist seit Version 2.0 in OpenGL integriert und wurde für die Imlementation des Visualiserungstools verwendet [11, 12]. Bei OpenGL werden die Shaderquelltexte zur Laufzeit vom Treiber kompiliert und auf die GPU geladen. Vor dem eigentlichen Aufruf der GLSL-Programme müssen die Renderziele der Berechnung und die Programmparameter festgelegt werden. Soll in Texturen gerendert werden, dann geschieht dies mit Hilfe eines *OpenGL Framebuffer Objekts* (FBO), an das die entsprechenden Ziel-Texturen gebunden sind. Über *Uniforme Parameter*, die den Vertex-und Fragmentprogrammen übergeben werden, kann den Programmen zum Beispiel mitgeteilt werden, welche Textureinheiten sie bei Lookups verwenden sollen. Zusätzlich ist es möglich, zu jedem Eingabevertex mehrere *Vertexattribute* anzugeben, die im Vertexprogramm abgefragt werden können.

4.4 General Purpose Computing mit der GPU

Die hohe Leistungsfähigheit der GPUs bietet sich dank ihrer Programmierbarkeit auch für grafikfremde Probleme an [13, 14, 15]. Viele wissenschaftliche oder geometrische Algorithmen lassen sich so umformulieren, dass sie auf dem Grafikprozessor ausgeführt werden können. In den letzten Jahren wurden verschiedene klassische Algorithmen auf der GPU implementiert. Zu ihnen gehören zum Beispiel die FFT, Matrix Multiplikation oder auch Strömungssimulationen [16, 17]. Besonders geeignet sind Probleme mit einer hohen *arithmetischen Intensität*, bei denen der Quotient aus der Anzahl der Rechenoperationen und der dabei übertragenen Datenmengen groß ist. Deshalb sollten die Berechnungen auf den einzelnen Stromdatenelementen möglichst unabhängig voneinander sein. Je höher der Grad der Abhängigkeit ist, umso mehr Daten müssen übertragen werden, was letztendlich eine schlechtere Performance zur Folge hat. Die Kommunikation zwischen den Stromelementen untereinander kann nicht immer umgangen werden, wobei es zwei Möglichkeit gibt, Daten auszutauschen. *Gather-Operationen* rufen die Daten anderer Stromelemente ab. *Scatter-Operationen* dagegen teilen das Ergebnis einer Kernelberechnung anderen Stromelementen mit. Normalerweise sind *Gather-Operationen* vorzuziehen, da sie nur lesende RAM-Speicherzugriffe erfordern. Diese sind auf der GPU durch Texturlookups realisierbar. *Scatter-Operationen* hingegen können nur über umständliche Tricks auf der GPU durchgeführt werden. Die Eingabedaten eines CPU-Algorithmus sind häufig im Hauptspeicher in einer Array- oder Zeigerdatenstruktur gespeichert. Um bei einer GPU-Berechnung als Eingabe dienen zu können, müssen die Daten vorher in Texturform gespeichert werden, weshalb dynamische Zeigerstrukturen nur schwer realisierbar sind. Üblicherweise wird auch das Resultat der Berechnung in eine Textur geschrieben. Da die programmierbaren Einheiten intern mit floating-point Zahlen rechnen, ist es sinnvoll, die Texturdaten in diesem Format bereitzustellen. Im Folgenden wird eine grundlegende Vorgehensweise bei der Implementation von GPGPU-Algorithmen am Beispiel von OpenGL erklärt. Für die eigentlichen Berechnungen sind die Fragmenteinheiten den Vertexeinheiten aufgrund ihrer größeren Gesamtrechenleistung vorzuziehen.

Die Daten auf denen der Algorithmus ausgeführt werden soll, sind in Texturen gespeichert, wobei in jedem Texel ein Datenelement mit maximal vier Werten enthalten ist. Werden mehr als vier Werte pro Element benötigt, dann müssen mehrere Eingabetexturen erstellt werden. Das Fragmentprogramm soll auf jedem Datenelement der Eingabe einmal ausgeführt werden. Zunächst wird die Größe des gerenderten Bildes (*Viewport*) auf die Texturauflösung eingestellt (Listing 4.1, Zeile 1). Die Projektions und Modelview-Matrix werden so angepasst (Zeilen 2-6), dass ein in den Bildraum projiziertes Rechteck exakt den Viewport ausfüllt (Abb. 4.3, Vertex Transformation). Zeile 7 stellt sicher, dass die inneren Bereiche des Rechtecks rasterisiert werden. In den folgenden Zeilen werden die Eingabe- und Ausgabetexturen festgelegt (Zeilen 9-14) und das gewünschte Fragmentprogramm initialisiert (Zeile 17, 18). Die eigentliche Berechnung stößt man durch das Zeichnen des Rechtecks an. Die Texturkoordinaten, die dabei angegeben werden, legen den Berechnungsraum fest. Das Rechteck wird genau in $Texres_X \cdot Texres_Y$ Fragmente rasterisiert. Somit wird das Fragmentprogramm für jedes Viewportpixel exakt einmal ausgeführt. Über die bilinear interpolierten Texturkoordinaten des Rechtecks können die zugehörigen Eingabedaten aus den Datentexturen gelesen werden (Abb. 4.3, Texturlookups). Die Texturfilterung aller Datentexturen muss auf *Nearest Neighbour* eingestellt sein, sonst kann es passieren, dass interpolierte Datenwerte zurückgeben werden. Das Fragmentprogramm kann mehrere Resultate pro Fragment in die vom FBO spezifizierten Texturen schreiben. Ein freier Schreibzugriff auf die Ausgabetexturen ist allerdings nicht möglich, da die Position des verarbeiteten Fragments der Position des Texels in der Ergebnistextur entspricht. Iterative Algorithmen verwenden die Ergebnistexturen in weiteren Render-

durchläufen als neue Eingabe, ohne dass die Daten vorher in den Hauptspeicher zurückgelesen werden müssen.

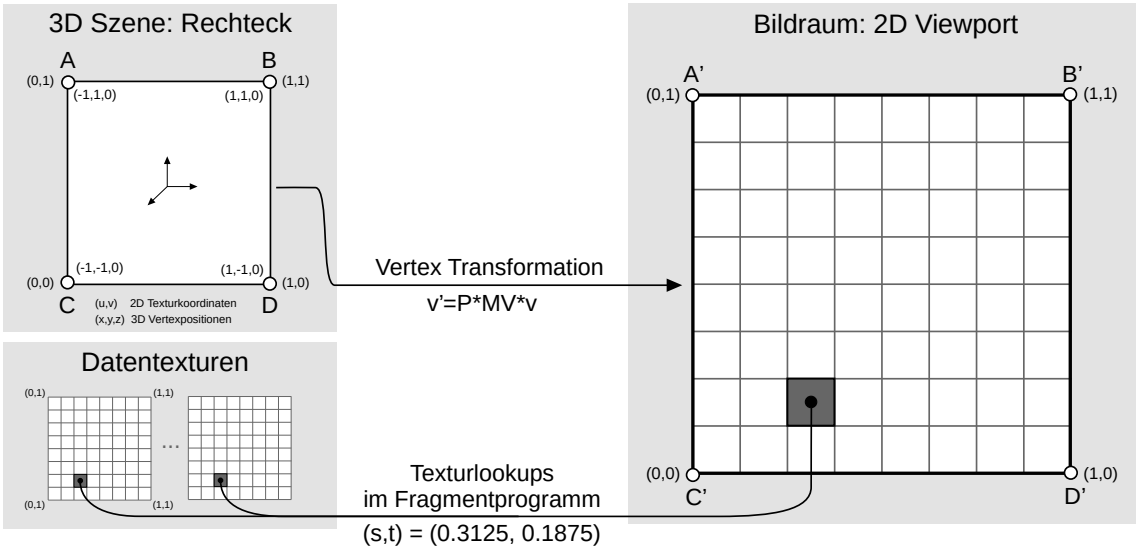


Abbildung 4.3: Durch das Zeichnen eines speziellen Rechtecks, kann im Fragmentprogramm auf die einzelnen Texel der Datentexturen zugegriffen werden.

Listing 4.1: Initialisierung und Starten einer GPGPU-Berechnung mit OpenGL

```
1  glViewport(0, 0, TEXRES_X, TEXRES_Y);
2  glMatrixMode(GL_PROJECTION);        // Projektions-Matrix setzen
3  glLoadIdentity();
4  gluOrtho2D(-1.0,1.0,-1.0,1.0);
5  glMatrixMode(GL_MODELVIEW);         // Modelview-Matrix setzen
6  glLoadIdentity();
7  glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
8
9  // Eingabe: N Datentexturen
10 glActiveTexture(GL_TEXTURE(i-1));glBindTexture(GL_TEXTURE_2D, DATATEX_i);
11
12 // Ausgabe: zb. in (mehrere) Texturen über FBOs
13 glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, FBO);
14 glDrawBuffer(DRAWBUFFERS);
15
16 // Kernel: Fragmentshader-Programm initialisieren
17 glUseProgram(FRAGMENTPROGRAMM);
18 glUniform(variable_id, ...Werte...);
19
20 // Berechnung starten
21 glBegin(GL_QUADS);
22   glTexCoord2f(0.0, 0.0);      glVertex2f(-1.0,-1.0);
23   glTexCoord2f(1.0, 0.0);      glVertex2f( 1.0,-1.0);
24   glTexCoord2f(1.0, 1.0);      glVertex2f( 1.0, 1.0);
```

```
25      glTexCoord2f(0.0, 1.0);      glVertex2f(-1.0, 1.0);
26    glEnd();
```

4.5 Feder-Masse-Modell auf der GPU

Die Grundlagen des Feder-Masse-Modells wurden in Kapitel 3 ausführlich besprochen. Aufgrund ihrer Einfachheit wurde die explizite Verlet-Integration als numerisches Integrationssverfahren implementiert [18, 19]. Methoden höherer Ordnung, wie RK4, bieten zwar eine bessere Genauigkeit, allerdings erfordert die größere Zahl von Funktionsauswertungen zusätzliche Texturlookups, welche Stalls in der GPU-Pipeline verursachen können und so die Berechnungen verlangsamen. Implizite Verfahren erfordern das Lösen komplizierter Gleichungssysteme [20, 21].

Die explizite Verlet-Gleichung für die Teilchenintegration lautet:

$$\boldsymbol{r}(t+\Delta t) = 2 \cdot \boldsymbol{r}(t) - \boldsymbol{r}(t-\Delta t) + \frac{\boldsymbol{F}_{total} \cdot \Delta t^2}{m} \tag{4.1}$$

In die Gleichung geht die Gesamtkraft $\boldsymbol{F}_{total}$ ein, die auf ein Teilchen wirkt. Sie setzt sich aus der Federkraft und der Stokes'schen Reibungskraft zusammen (Glg. 3.11), wobei der Vorfaktor für die Reibung, in den die Viskosität und der Teilchenradius eingehen, auf eins gesetzt wird. Zusätzlich kann das dynamische Verhalten der Geometrie über zwei skalare Faktoren für die Stärke des Feldes f_{fac} und die Dämpfung d_{fac} eingestellt werden. Die Gesamtkraft ergibt sich zu

$$\boldsymbol{F}_{total} = \frac{f_{\text{fac}} \cdot \boldsymbol{V}(\boldsymbol{r}(t), t) - \boldsymbol{v}(t)}{\Delta t} - \boldsymbol{F}_{feder} - d_{\text{fac}} \cdot \boldsymbol{v}(t) \tag{4.2}$$

mit

$$\boldsymbol{v}(t) = \frac{\boldsymbol{r}(t) - \boldsymbol{r}(t-\Delta t)}{\Delta t} \tag{4.3}$$

Es gibt zwei grundsätzlich verschiedene Vorgehensweisen um die gesamte Federkraft $\boldsymbol{F}_{feder}$ zu bestimmen. Der vertexorientierte Gather-Ansatz berechnet die Abstände eines Vertex zu jedem einzelnen seiner Nachbarn und akkumuliert die resultierenden Teilkräfte:

```
for (alle Teilchen i)
  for (alle Nachbarn j von i)
    Federkraft_Gesamt_i += Teilkraft_ij
```

Die Federkraft wird hierbei pro Feder zweimal berechnet. Einmal für das linke Teilchen und einmal für das rechte. Ein kantenorientierter Scatter-Ansatz würde diese unnötigen Berechnungen vermeiden. Die Kräfte, die von einer Feder ausgehen, werden berechnet und auf die zwei verbundenen Vertices verteilt.

```
for (alle Federn i)
{
  Federkraft_linker_Nachbar  += Teilkraft_i
  Federkraft_rechter_Nachbar += Teilkraft_i
}
```

Wie wir gesehen haben, werden *Scatter-Operationen* von der GPU nur unzureichend unterstützt. Deshalb wird trotz des höheren Aufwands die erste Methode verwendet.

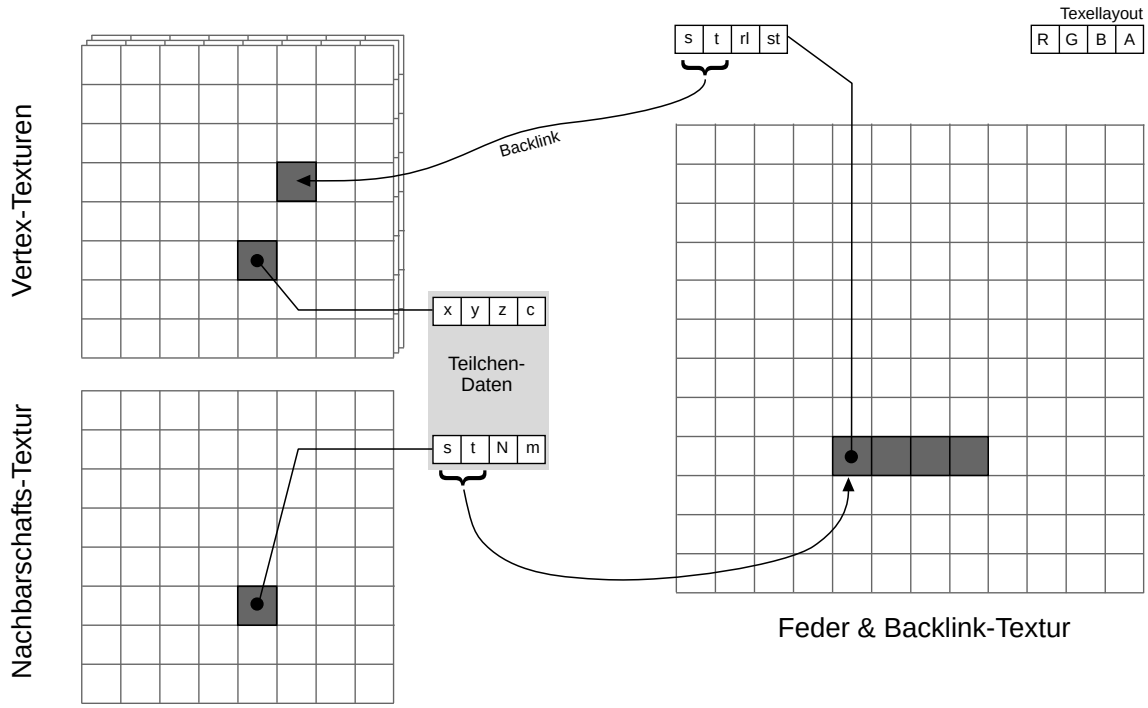


Abbildung 4.4: Die Vertex-Texturen speichern die Teilchenpositionen. Die Nachbarschafts- und die FB-Textur enthalten die Informationen über die Federverbindungen zwischen den Teilchen.

Für die Berechnung eines neuen Zeitschritts sind jeweils die Positionen des aktuellen und des letzten Zeitschritts nötig. Die Positionsinformationen werden hierzu in den RGB-Komponenten dreier *Vertex-Texturen* gleicher Auflösung gespeichert (Abb. 4.4). Eine explizite Berechnung und Speicherung der Geschwindigkeiten ist nicht nötig. In einer weiteren Textur der selben Auflösung (*Nachbarschafts-Textur*) sind jeweils die zugehörigen Massen m der Teilchen, die Anzahl N der über Federn verbundenen Nachbarn und eine zweidimensionale Texturkoordinate (s,t) enthalten. Diese Texturkoordinate ermöglicht den Zugriff auf die Nachbarschaftsdaten eines Teilchens (Federdaten und Backlinks), die in einem N-Texel breitem Nachbarblock in der FB-Textur gespeichert sind. Jedes einzelne Texel dieses Blocks enthält einen Backlink (s,t) zu den Positionsdaten eines Nachbarteilchens, sowie die zugehörige Federruhelänge (rl) und Federsteifigkeit (st) der Feder, über die die Teilchen verbunden sind. Ein Texturzeilenumbruch innerhalb eines solchen Blocks ist nicht erlaubt, da sonst eine aufwendige Berechnung der Texturkoordinaten vor dem Zugriff auf die Nachbardaten nötig ist. Die Nachbarblöcke verschiedener Vertices werden aufeinanderfolgend in den Zeilen der FB-Textur gespeichert, wobei die maximal erlaubte Tex-

turbreite von 4096 Texeln ausgenutzt wird. Alle vier Texturen speichern pro Texel vier Komponenten im fp32-Format.

Das Strömungsfeld ist als Vektorfeld gegeben und wird in einer 3D-Textur gespeichert. Da die aktuelle Generation von GPUs keine hardwareseitige Unterstützung für Interpolation in fp32 3D-Texturen anbietet, müssen die Komponenten des Feldes vor dem Textur-Download auf die GPU in das 16bit *Half-Float-Format* der GPU konvertiert werden.

Der numerische Algorithmus ist als Fragmentprogramm implementiert (Listing 4.2). Zunächst werden die Positionen der letzten beiden Zeitschritte in den Vertex-Texturen nachgeschlagen (Zeile 3 und 4). Darauf folgen ein oder zwei weitere, von den ermittelten Positionen abhängige, Lookups in den Vektorfeldtexturen. Bei instationären Feldern resultiert der Feldvektor zur Simulationszeit t aus einer linearen Interpolation der zwei angrenzenden Feldsamples (Zeilen 11-13). Zeile 19 holt die nötigen Teilchenattribute und die Texturkoordinaten für den Zugriff auf die Nachbarschaftsinformation. Die darauf folgende Schleife ermittelt die gesamte Federkraft, die auf das aktuelle Teilchen einwirkt (Zeilen 25-34). Am Ende werden die Feder-, Reibungs- und Dämpfungskräfte addiert (Zeilen 37-39) und anschließend wird die Integrationsgleichungen angewandt. Das Resultat schreibt das Programm als Fragmentfarbe in die Ausgabetextur.

Listing 4.2: Auszüge aus dem GLSL Quelltext des Verlet-Integration-Shaders

```
1
2  // aktuelle und letzte Vertexposition holen
3  vec4 pos_t     = texture2D(vTextur_t    , gl_TexCoord[0].st);
4  vec4 pos_t_alt = texture2D(vTextur_t_alt, gl_TexCoord[0].st);
5
6  // Vektorfeldsample
7  # im Fall von stationären Feldern
8  vec4 V = texture3D(vectorField0, pos_t.xyz/vfAuflösung);
9
10 # im Fall von instationären Feldern
11 vec4 fv0 = texture3D(vectorField0, pos_t.xyz/vfAuflösung);
12 vec4 fv1 = texture3D(vectorField1, pos_t.xyz/vfAuflösung);
13 vec4 V = t_relativ*fv0 + (1.0-t_relativ)*fv1;
14
15 // Nachbarschafts-und Teilchenattribute
16 //   np.st = Texturkoordinate für NB-Textur
17 //   np.z  = Nachbaranzahl
18 //   np.w  = Teilchenmasse
19 vec4 np = texture2D(npTex, gl_TexCoord[0].st);
20
21 // intialisiere Federkraft
22 vec3 F_feder = {0.0 ,0.0, 0.0};
23
24 // einzelne Federkräfte aufsummieren
25 for (float i=0.0; i<np.z; i++)
26 {
27   vec4 ns    = texture2D(nsTex, np.st+i*vec2(texel_ds,0.0));
28   vec4 neigh = texture2D(vTex, ns.st);
29
30   vec3 spring = pos_t.xyz - neigh.xyz;
31   float l = length(spring);
```

```
32      float factor = fac_steifigkeit*ns.w*((l - ns.z)/l);
33      F_feder += factor * spring;
34    }
35
36    // Gesamtkraft
37    vec3 F_total = (f_fac*V - (pos_t.xyz-pos_t_alt.xyz)/dt)/dt
38                   - F_feder
39                   - d_fac*(pos_t.xyz-pos_t_alt.xyz)/dt;
40
41    // Verlet-Integration
42    gl_FragColor.xyz = 2*pos_t.xyz - pos_t_alt.xyz
43                       + F_total*dt*dt/np.w;
```

5 Implementation

5.1 Einleitung

Dieses Kapitel geht näher auf das implementierte Visualisierungsprogramm ein. Zuerst wird ein Überblick über das Systemdesign gegeben. Danach werden die aufgetretenen Teilprobleme einschließlich Lösungsvorschlägen in separaten Abschnitten behandelt. Auf Implementationsdetails wird an interessanten Stellen eingegangen. Ferner wird die grafische Benutzeroberfläche (GUI) und die Bedienung des Programms erläutert.

Systemüberlick

Das komplette Visualisierungstool wurde unter Windows XP mit Visual Studio .Net 2003 in C++ und OpenGL programmiert. Es setzt auf ein schon vorhandes Basis-Framework auf und verwendet wie dieses GLUT [20]. In Abbildung 5.1 ist eine vereinfachte Übersicht über die Programmklassen und deren Zusammenhänge dargestellt. Die Hauptfunkionalität steckt in der *Geometrieadvektions-Klasse*. Sie übernimmt alle Simulationsaufgaben, die Interaktion mit den Objekten, das automatische Platzieren der Versuchsobjekte, die Geometrierandbehandlung sowie das Rendern. Die *Umgebungs-Klasse* dient dagegen hauptsächlich der Initialisierung und Verwaltung des Systems. Sie enthält die *Hauptschleife*, die für jedes gerenderte Bild einmal aufgerufen wird und die Vektorfeld-Streaming-Komponente. Außerdem implementiert sie die Schnittstelle zum Benutzer. Die restlichen Klassen realisieren Teilaufgaben, die für die Geometrieadvektion relevant sind. Die geometrischen Versuchsobjekte, deren Bewegungen im Strömungsfeld visualisiert werden, werden vom *GPU-Daten-Generator* vervielfältigt und in Texturen gespeichert. Außerdem werden für die Simulation geeignete Parameter benötigt. Diese berechnet der *Feldanalysator* in Abhängigkeit verschiedener Eigenschaften des zugrundeliegenden Strömungsfeldes. Der *GLSL-Lader* verwaltet die Vertex- und Fragmentprogramme, die das System verwendet.

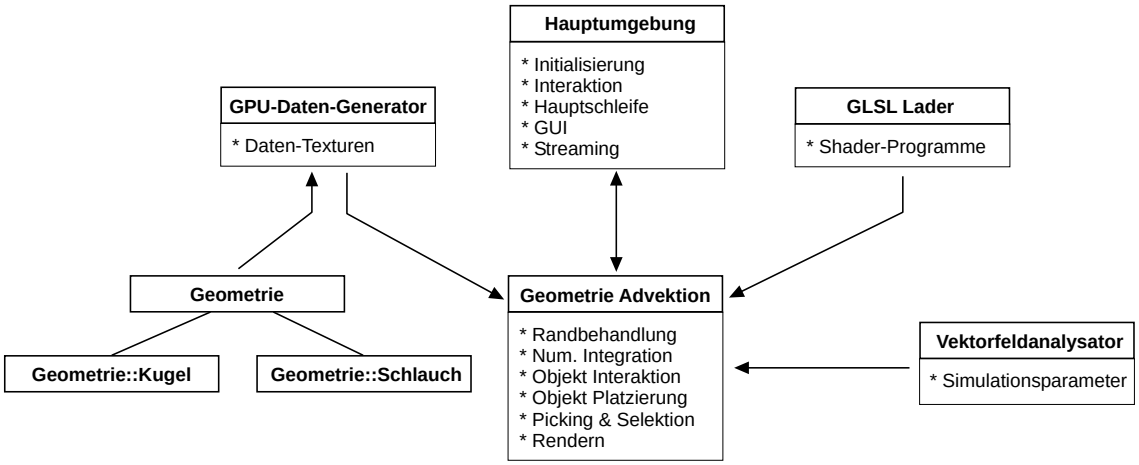


Abbildung 5.1: Systemaufbau des Visualisierungtools

Im Unterkapitel *Bedienung und Interaktion* ist eine kurze Einführung in die Benutzung des Programms gegeben. In den darauf folgenden Abschnitten werden die Hauptfunktionen in der Reihenfolge behandelt, in der sie von der Hauptschleife ausgeführt werden (Abb. 5.2):

Geometrieproben behandelt die Generierung und die Interaktion mit den Proben sowie die Problematik einer geeigneten Platzierung im Strömungsfeld. *Simulation* geht nochmals auf die Integration der Geometrie und auf die Wahl von Simulationsparametern ein. *Randbehandlung* zeigt die besondere Behandlung von Geometrieproben auf, die am Rand des Simulationsgebietes angekommen sind. Das *Rendern* der Geometrie und das *Streaming der Vektorfelddaten* wird in den letzten beiden Unterkapiteln behandelt.

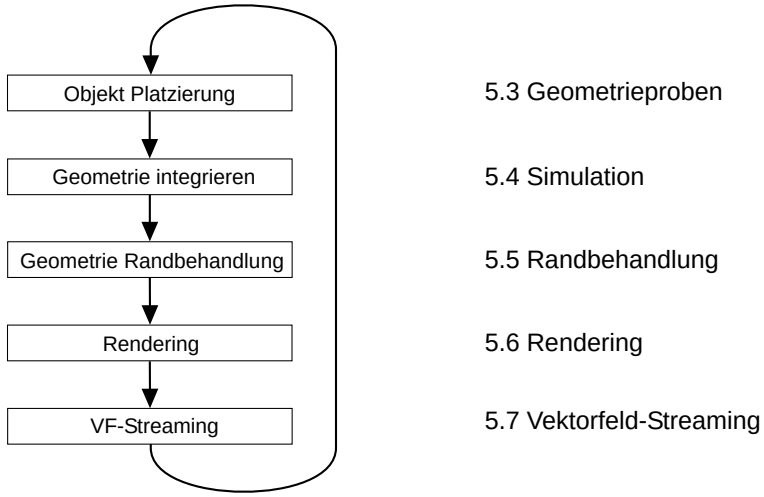


Abbildung 5.2: Die Hauptschleife wird für jedes gerenderte Bild einmal ausgeführt.

5.2 Bedienung und Interaktion

Das Programm muss mit dem Dateinamen einer Konfigurationsdatei (.par) als einzigem Parameter gestartet werden. In dieser Textdatei werden das Strömungsfeld und die Probenobjekte spezifiziert. Außerdem können die meisten Simulations- und Visualisierungsparameter festgelegt werden (Tabelle 5.1). Jeder Eintrag muss in einer eigenen Zeile stehen. Kommentarzeilen, die mit dem Raute Zeichen beginnen, werden ignoriert. Fehlende Parameter werden vom Programm durch Standardwerte ersetzt. Wählt man den Wert für die Simulationsschrittweite kleiner Null, dann werden die Schrittweite und der Faktor für die Federsteifigkeit basierend auf einer automatischen Analyse des Strömungsfeldes bestimmt. Als Versuchsobjekte stehen Kugeln und Schläuche zur Verfügung, die aber nicht gleichzeitig verwendet werden können. Mehrere Parameter für die Geometriegenerierung legen das Aussehen und die Eigenschaften des jeweiligen Typs fest. Wobei der Objektversatz zusammen mit der Startposition die eigentlichen Positionen der n Objekte bestimmt ($pos_n = obj\_pos + n * obj\_offset$). Soll das Feld mit Hilfe von Saatobjekten untersucht weden, muss der Objektversatz auf den Standardwert eingestellt sein.

Nach dem Start des Programms ist der Begrenzungsrahmen des Vektorfeldes zu sehen. Der Ursprung $\mathbf{0} = (0.0, 0.0, 0.0)$ des Koordinatensystems ist durch ein farbiges Dreibein markiert und die Versuchs-

Beschreibung	**Schlüsselwort**	**erlaubte Werte**	**Standardwert**
Vektorfelddatensatz	vfield	Dateiname	default.dat
Feldstreaming (an/aus)	stream_on	0, 1	0 (aus)
Feldstreaming Pufferanzahl	stream_buf	> 0	5
Feldstreaming Puffer vorladen	stream_pre	(1..stream_buf)	4
Feldstreaming Startzeitschritt	stream_offset	(0..#Zeitschritte-1)	0
Objekttyp	obj_type	0=Schlauch, 1=Kugel	1
Objekt Stützvertices (an/aus)	obj_v_in	0,1	0 (aus)
Kugel Subdivision-Schritte	obj_sphere_sub	1..5	1
Schlauch Segmentanzahl	obj_tube_seg	$>= 1$	10
Schlauch Querschnittvertices	obj_tube_v	$>= 3$	3
Objektlänge	obj_length	> 0.0	30.0
Objektradius	obj_radius	> 0.0	2.0
Objektanzahl	obj_num	0..4096	1024
Objektposition	obj_pos	(x,y,z) $\in \mathbb{R}^3$	(0.0, 0.0, 0.0)
Objektversatz	obj_offset	$(\Delta x, \Delta y, \Delta z) \in \mathbb{R}^3$	(0.0, 0.0, 0.0)
Simulationsparameter Δt	sim_dt	> 0.0	0.1 sek
Simulationsparameter ΔT	sim_dts	> 0.5	2.0 sek
Simulationsfaktor Dämpfung	sim_fac_d	> 0.0	0.0 (aus)
Simulationsfaktor Feldstärke	sim_fac_f	> 0.0	10.0
Simulationsfaktor Federsteifigkeit	sim_fac_s	> 0.0	1.0
Randbehandlungsparameter	boundary_num	1..obj_num	50
Saat-Periode	seed_period	> 1	50
Render Transferfunktion Max	render_t_max	> 0.0	10.0

Tabelle 5.1: Format der Parameterdateien (.par)

objekte werden an den spezifizierten Orten angezeigt (Abb. 5.3). Die Bedienung des Programms erfolgt mit der Tastatur und der Maus. In Tabelle 5.2 sind die Tastenkürzel der Hauptfunktionen aufgelistet. Die Sicht auf das Feld kann, wie bei anderen 3D-Programmen auch, mit der Maus geändert werden. Der *Rendermodus* legt das Aussehen der Versuchsobjekte fest. Gewählt werden kann zwischen einer Darstellung der Dreieckskanten (Wireframemodus) und der Objektoberflächen (Fillmodus). Zusätzlich läßt sich eine auf den Objektnormalen basierende Beleuchtungsberechnung aktivieren. Änderungen des Programmzustands werden über eine Meldung auf der Kommandozeile angezeigt. Der *Interaktionsmodus* legt fest, welche Objekte verschoben werden können. Im *Auto-Modus* ist dies davon abhängig, ob ein Saatpunkt ausgezeichnet ist oder Objekte selektiert sind. Über einen Tastendruck ist es außerdem möglich, die aktuellen Simulationstexturen und Parameter auf der Festplatte zu sichern.

Auch die Selektion und Interaktion mit den Versuchsobjekten ist über eine Kombination aus Tastatur- und Maussteuerung möglich (Tab. 5.3). Die meisten Objektoperationen beschränken sich auf die aktuelle Selektion, die blau eingefärbt ist. Es ist möglich, komplette Objekte oder festgelegte Vertexmengen zu verschieben. Hierfür gibt es für jeden Objekttyp zwei ausgezeichnete Vertexbereiche A und B. Bei Schlauchobjekten sind das jeweils die Vertices der beiden Enden. Bei Kugeln ist dagegen nur der Mittelpunkt markiert. Werden die Objektvertices der Mengen fixiert, dann werden sie weiss eingefärbt und Ihre Position ändert sich auch bei laufender Simulation nicht. Beide Mengen sind entweder aktiv oder

Funktion	**Taste**
Simulation starten/stoppen	Leertaste
Schrittweise Simulation (1, 25 Schritte)	(-, +)
Programm beenden	q
Feld Rotieren	Ctrl + linke Maustaste
Feld Zoom	Ctrl + rechte Maustaste
Feld Verschieben	Ctrl + mittlere Maustaste
Interaktionsmodus ändern (Auto, Selection, Fixed)	m
Rendermodus überblenden (Flat...Normal)	b,B
Rendermodus ändern (Fill, Wireframe)	r
Pseudotransparenz (an, aus)	t
Randbehandlungsparameter (-,+)	(j, J)
Transferfunktion Schwellwert (-,+)	(c, C)
Simulationsdaten in Texturen sichern	w
Aktuelle Parameter in default.par schreiben	W
Allgemeine Parameter anzeigen	1
Saatmodus Parameter anzeigen	2
Simulationsparameter anzeigen	3

Tabelle 5.2: Die Hauptfunktionen des Programms

Funktion	**Taste**
Auswahl neu	linke Maustaste
Auswahl verkleinern	mittlere Maustaste
Auswahl vergößern	rechte Maustaste
Auswahlkonfiguration sichern	s
Auswahlkonfiguration wiederherstellen	S
Auswahl abbrechen	ESC
Auswahl bewegen (x,y)	Shift + linke Maustaste
Auswahl bewegen (z)	Shift + rechte Maustaste
Auswahl bewegen (aktive fixierte Vertices) (x,y)	Shift + linke Maustaste
Auswahl bewegen (aktive fixierte Vertices) (z)	Shift + rechte Maustaste
Auswahl Fixierung Bereich A (ein/aus)	(i, I)
Auswahl Fixierung Bereich B (ein/aus)	(o, O)
Auswahl Fixierung Bereich A aktivieren (ein/aus)	p
Auswahl Fixierung Bereich B aktivieren (ein/aus)	P
Saatpunkt auswählen	Rechtsklick auf Saatpunkt
Saatpunkte deselektieren	Rechtsklick neben Objekte

Tabelle 5.3: Interaktion mit den Versuchsobjekten

(a) Saatpunkte

Funktion	Taste
Saatmodus initialisiern	l
Saatmodus beednen	L
Saatpunkt hinzufügen	k
Saatperiode +1, +10	x, X
Saatperiode -1, -10	y, Y

(b) Simulationsparameter

Funktion	Taste
Zeitschrittweite Δt (-,+)	(g, G)
Dämpfungsfaktor (-,+)	(d, D)
Federsteifigkeitsfaktor (-,+)	(h, H)
Feldstärkefaktor (-,+)	(f, F)
Massefaktor (-,+)	(a, A)

Tabelle 5.4: Tastenkürzel für den Saatmodus und für die Manipulation der Simulationsparameter

inaktiv, wobei nur *aktivierte Vertexmengen* verschoben werden können. Dies ermöglicht beispielsweise das separate Fixieren und Bewegen der beiden Schlauchenden.

Geraten Objekte bei laufender Strömungsvisualisierung an den Rand oder außerhalb des Feldes, werden sie automatisch an ihre ursprünglichen Positionen zurückgesetzt. Zusätzlich lassen sich die Objekte auch von Hand zurücksetzen (*Auswahlkonfiguration wiederherstellen*) und die Rücksetzpositionen können durch die aktuellen Vertexpositionen ersetzen werden (*Auswahlkonfiguration sichern*).

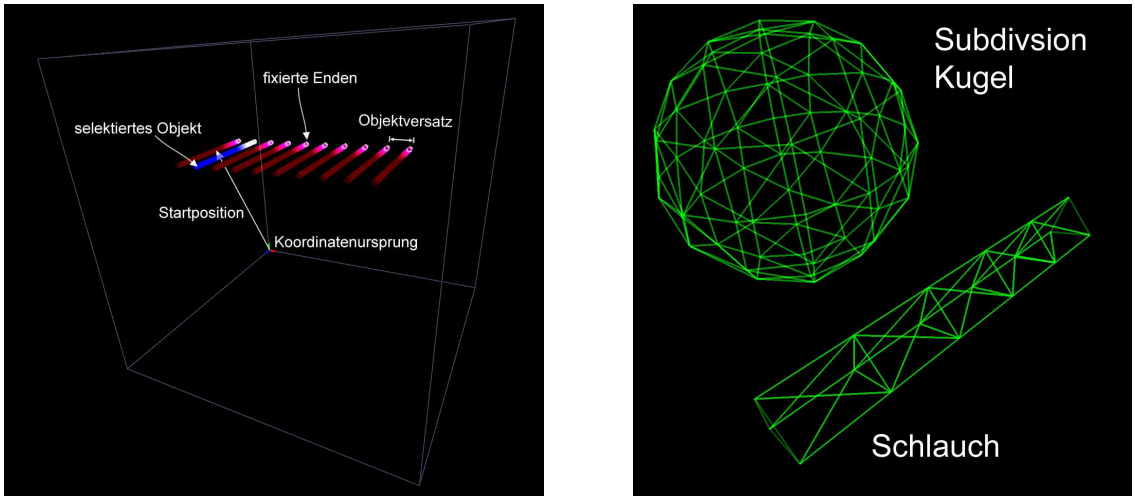


Abbildung 5.3: Programm nach dem Starten (links), geometrische Versuchsobjekte (rechts)

Wechselt man in den Saatmodus (Tab.5.3(a)), dann erscheint zunächst ein einzelner Saatpunkt, an dem periodisch Objekte freigesetzt werden. Daraufhin ist es möglich, weitere Saatpunkte zu generieren und die Periodendauer einzustellen. Außerdem erlaubt es das Programm, verschiedene Simulationsparameter interaktiv zu manipulieren (Tab. 5.3(b)).

5.3 Geometrieproben

Die Grundidee bei der Geometrieadvektions-Methode ist, eine große Menge gleichartiger, relativ einfach aufgebauter Versuchsobjekte in einem Strömungsfeld freizulassen. Aus der Bewegung und Verfor-

mungen der Objekte können Informationen über das zugrundeliegende Feld gewonnen werden. Befinden sich zu viele Objekte in einem kleinen Raumbereich, dann ist die Wahrscheinlichkeit für gegenseitge Verdeckungen und Überschneidungen groß. Die Platzierung und Verteilung der Objekte ist deshalb entscheidend für eine gelungene Visualisierung. Eine Kollisionsbehandlung der Objekte untereinander bringt hier keinen weiteren Nutzen, da die Eigenschaften des Feldes dargestellt werden sollen und nicht die Kollision von Teilchen. Ein anderes Problem betrifft jedoch die einzelnen Versuchsobjekte selber. Durch die Einwirkung des äußeren Strömungsfeldes können Selbstdurchschneidungen auftreten, was zu unerwünschten Artefakten führt. Eine seperater Test auf Überschneidungen von Geometrieprimitiven und deren Behandlung erscheint zu aufwändig und den Anforderungen unangemessen. Ein alternativer Ansatz ist, die Objektgeometrie und somit die Massepunkte und Federn so zu wählen, dass allein durch das physikalische Feder-Masse-Modell Artefakte verhindert werden. Beispielsweise können Dreiecke nicht zu einer Linie degenerieren und im dreidimensionalen Fall erfüllen Tetraeder die selben Anforderungen. Allerdings ist die korrekte Tetraedrisierung von geometrischen Objekten im allgemeinen Fall eine sehr komplizierte Aufgabenstellung. Eine optimale Zerlegen der Objektvolumen in Tetraeder wurde deshalb nicht angestrebt.

In den folgenden Unterkapiteln werden, ausgehend von der Geometriegenerierung, die GPU-Datenstrukturen für die Objektspeicherung und die Interaktion mit den Proben behandelt. Am Ende wird die Implementation der Saatpunkte und die automatische Platzierung von Versuchsobjekten besprochen.

5.3.1 Objektgenerierung

Implementiert wurden parametrische Geometriegeneratoren für schlauchförmige Objekte und für Kugeln. Andere Geometrien können durch das Ableiten einer weiteren Geometrie-Kindklasse in das Programm integriert werden. Im Hinblick auf den Integrationsalgorithmus sollte darauf geachtet werden, dass die Anzahl der von den Vertices ausgehenden Kanten innerhalb eines Objektes möglichst konstant ist. Aufgrund der GPU-Architektur kann ein einzelner Vertex mit größerer Nachbarzahl zu einer Engstelle werden und die Performance drücken.

Schläuche

Schläuche können sich durch ihre längliche Form an die Strömung anpassen. Durch Änderungen in der Länge werden Divergenzen im Strömungsfeld aufgezeigt, und durch lokale Deformationen im Durchmesser werden räumlich begrenzte Feldeigenschaften sichtbar. Ein Schlauch (Abb. 5.4) ist aus mindestens einem Segment aufgebaut und besteht an den Segmenträndern mindesten aus drei Vertices, die auf einem Kreis liegen. Die Anzahl der Segmente und Vertices im Querschnitt sowie der Schlauchdurchmesser und seine Länge können als Parameter angegeben werden. Bei Schläuchen mit mehr als drei Vertices im Querschnitt kann das Problem der Geometrie-Zusammenfaltung auftreten. Ist dies unerwünscht, müssen zusätzliche Stützvertices im Inneren des Objektes eingeführt werden. Das Deformationsverhalten im Querschnitt und längs des Schlauchs kann voneinander unabhängig über die Wahl verschiedener Federsteifigkeiten der Querschnittskanten und Längskanten eingestellt werden.

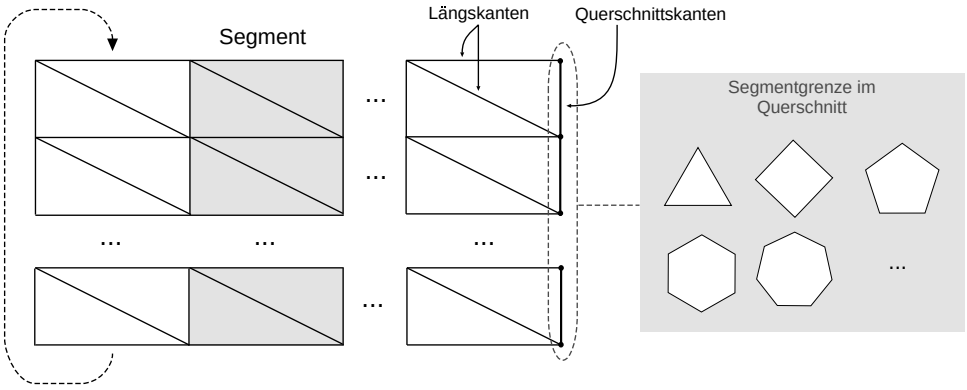


Abbildung 5.4: Geometrischer Aufbau der Schlauchobjekte

Kugeln

Die isotrope Kugelgeometrie eignet sich gut für die Visualisierung des Strömungsfeldes. Lokale Eigenschaften des Feldes werden durch Verformung der Oberfläche optisch dargestellt. Gleichzeitig läßt sich die Stärke des Feldes an der Geschwindigkeit der Kugeln erkennen.

Die Kugelgeometrie wird mit Hilfe eines Subdivisionsalgorithmus generiert. Gestartet wird mit einem Oktaeder, dessen sechs Vertices auf der Einheitskugel liegen. Ein Subdivisionsschritt fügt in der Mitte der Kanten des aktuellen Netzes neue Vertices ein, die dann über neue Kanten verbunden werden (Abb. 5.5). Die Vertices werden anschliessend auf den Einheitskreis projiziert. Die Zahl der Dreiecke nimmt dabei mit jedem Unterteilungsschritt um den Faktor vier zu. Nach wenigen Schritten ist eine ausreichende Approximation der Kugelform erreicht. Obwohl die wenigen Vertices des ursprünglichen Oktaeders einen Grad von vier haben und die neu eingefügten einen Grad von sechs, sind die Dreiecke ausreichend gleichförmig verteilt. Würde man eine Ikosaeder als Startobjekt wählen, dann wären alle Dreiecke gleich groß.

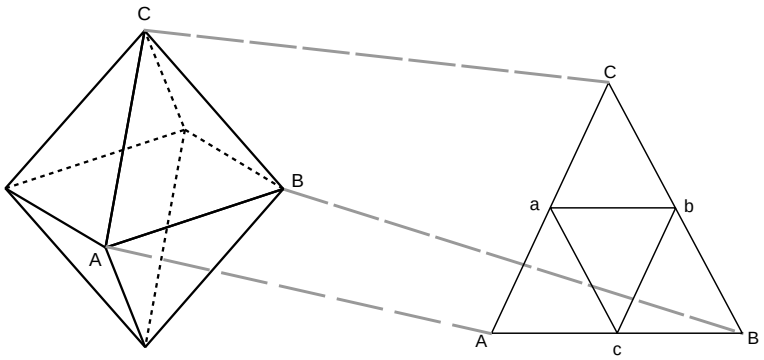


Abbildung 5.5: Tesselierung einer Kugel mittels Subdivision. $\Delta(A,B,C)$ wird ersetzt durch $\Delta(A,a,c)$, $\Delta(a,C,b)$, $\Delta(a,b,c)$ und $\Delta(c,b,B)$

Aufgrund des Selbstdurchschneidungsproblems ist es ab einer Unterteilung von mehr als 2 Schritten sinnvoll in der Mitte des Netzes einen zusätzlichen Vertex einzufügen, der über Kanten mit den äußeren Vertices verbunden ist. Ein zu großer Kantengrad beeinträchtigt aber die Performance der Vertexintegration. Deshalb kann es von Vorteil sein, den Mittelpunkt nur mit einer Teilmenge der Oberflächenvertices zu verbinden. Ein alternativer Ansatz schachtelt mehrere Kugelnäherungen mit zunehmender Unterteilungsstufe ineinander, wobei der Kugeldurchmesser mit dem Grad der Unterteilung zunimmt. Danach werden nahe beieinander liegende Vertices benachbarter Schalen miteinander verbunden.

5.3.2 Texturlayout

Die elementaren Grundobjekte und deren Topologie werden, wie im GPU-Kapitel besprochen, in drei verschiedenen Texturobjekten gespeichert. In den Vertex-Texturen sind die Vertexpositionen jedes einzelne Objektduplikates in einem eigenen Rechteck abgespeichert. Die Breite der Textur in x-Richtung entspricht dabei der Anzahl der Vertices pro Objekt und ist durch die maximale Texturbreite von 4096 Pixeln beschränkt. Größere Objekte werden auf mehrere Zeilen aufgeteilt. Da die Texturhöhe ebenso eingeschränkt ist, können maximal 4096 Kopien des Grundobjekts generiert werden. Für jede Objektkopie und all seine Vertices müssen Nachbarschaftsinformationen gespeichert werden. Die Struktur der Nachbarschafts-Textur entspricht der der Vertex-Textur. In der FB-Textur werden die Nachbarblöcke zeilenweise eingetragen, wobei die maximale Breite von 4096 Pixeln ausgenutzt wird. Da nur die RGB-Komponenten der Vertex-Texturen verwendet werden, kann in den Alpha-Komponenten ein zusätzliches Teilchenattribut gespeichert werden.

5.3.3 Interaktion mit den Proben

Die Geometrieadvektionsmethode visualisiert die Feldeigenschaften nur an den Stellen im Raum, an denen sich ein Objekt befindet. Es ist deshalb wichtig, dass der Benutzer die Positionen der virtuellen Versuchsobjekte manipulieren kann. Dank der Echtzeitsimulation auf der GPU sind dabei die Auswirkungen, die das Verschieben und Manipulieren hat, sofort sichtbar. Problematisch dabei ist allerdings, dass der GPU-Ansatz den Simulationzustand komplett auf der Grafikkarte speichert und ein Zurücklesen der Daten in den Hauptspeicher aus Performancegründen möglichst vermieden werden soll. Diese Einschränkung erschwert Modifikationen an der Objektgeometrie, die vom Benutzer ausgehen und erfordert deshalb spezielle GPU-basierte Methoden für das Selektieren der Geometrieobjekte, die Speicherung der Selektion und für die restlichen Modifikationsoperationen.

Das Programm realisiert das Selektieren der Geometrie mit Hilfe von *OpenGL Occlusion Queries*. Hierbei wird bei einem auf den Mauscursor oder den selektierten Bereich eingeschränkten Viewport jedes Objekt einzeln ohne Bildschirmausgabe gerendert und das Resultat der Occlusion-Query abgefragt. Da die Anzahl der gerenderten Fragmente zurückgegeben wird, kann ein Ergebnis größer Null als Treffer für das entsprechende Objekt gewertet werden. Die CPU merkt sich die gefundenen Objekte, indem sie die jeweilige Identifikationsnummer in einer Liste speichert. Würde man die Selektionsinformationen jedoch nur auf der CPU bzw. im Hauptspeicher verwalten, dann wären separate und zeitaufwändige Render-, Interaktions- und Simulationsaufrufe für jedes einzelne Objekt erforderlich, sobald eine auf die selektierten Objekte beschränkte Operation durchgeführt wird. Die volle Leistung der GPU kann aber erst genutzt werden, wenn ein einziger OpenGL-Aufruf für die Modifikation aller selektierten Objekte ausreicht.

Hierzu wird die Tiefentest-Funktionalität der GPU ausgenutzt. Gespeichert wird die Selektion auf der GPU in Tiefentexturen, die im Folgenden als Vertexmasken bezeichnet werden, und in der Größe und dem Aufbau genau den Vertex-Texturen entsprechen. Die Grundidee ist, jedem selektierten Objektvertex einen speziellen Tiefenwert zuzuweisen, der sich von dem Wert der nichtselektierten Vertices unterscheidet. Da die Objekte in rechteckigen Blöcken gespeichert sind ist es möglich, die Tiefenwerte kompletter Objekte mit einem einzigen Aufruf zu setzen. Bei folgenden Berechnungen und Operationen auf den Vertex-Texturen muss dann nur noch der Tiefentest aktiviert und entsprechend initialisiert werden, um die Auswirkungen der Operation auf die selektierten oder nichtselektierten Objekte einzuschränken:

1. Es wird festgelegt, welche Vertices von der Operation betroffen sind. Entweder sind dies alle Objektevertices, nur die maskierten Vertices oder nur die nichtmaskierten Vertices (Zeilen 2-14).

2. Eine Vertexmaske wird geladen, indem die entsprechende Tiefentextur an ein Framebuffer-Objekt gebunden wird (Zeilen 16-21).

3. Am Ende kann eine beliebige Operation auf den Vertex-Texturen durchgeführt werden. Der Tiefentest stellt dabei sicher, dass in der Ausgabetextur nur die gewünschten Vertexpositionen aktualisiert werden.

Listing 5.1: Operationen auf maskierten Vertex-Texturen

```
1
2   // 1. aktiviere und initialisiere den Tiefentest
3   switch(op_mode)
4   {
5   case OP_ALL:                    // alle Objekte
6     glDisable(GL_DEPTH_TEST);
7     break;
8   case OP_MASKED:                 // maskierte Objektvertices
9     glEnable(GL_DEPTH_TEST);
10    glDepthFunc(GL_GEQUAL);
11  case OP_UNMASKED:               // unmaskierte Objektvertices
12    glEnable(GL_DEPTH_TEST);
13    glDepthFunc(GL_LESS);
14  }
15
16  // 2. aktiviere FBO und binde die gewünschte Maske daran
17  glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, _fbo_id);
18
19  glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
20                            GL_DEPTH_ATTACHMENT_EXT,
21                            GL_TEXTURE_2D, maske, 0);
22
23  // 3. aktiviere das Shaderprogramm,
24  // binde Vertex-Texturen für die Eingabe und die Ausgabe
25  // und zeichne das Rechteck, dass die Operation anstößt.
```

Neben einer Tiefenmaske für die aktuelle Selektion (*Selektions-Maske*) verwaltet das Programm zusätzlich eine Maske für fixierte Vertices (*Vertex-Fix-Maske*) und mehrere Masken für den Saatpunktmodus.

Beispiel: Translationsoperation

Die selektierten Geometrieproben können vom Benutzer interaktiv im Raum verschoben werden. Ein simples Fragmentprogramm addiert auf die Positionen der maskierten Vertices (Selektions-Maske) einen Translationsoffset, der sich aus der Mausbewegung berechnet. Die neuen Positionen müssen sowohl in der aktuellen als auch in der Vertex-Textur des letzten Zeitschritts gespeichert werden. Wird nur die aktuelle Textur aktualisiert, dann treten Artefakte auf, da die alten Positionen beim nächsten Integrationsschritt verwendet werden. Alternativ können mit der Vertex-Fix-Maske die fixierten Vertices verschoben werden.

5.3.4 Saatpunkte

Particle-Trace-Methoden gehören zu den elementaren Verfahren der Strömungsvisualisierung. Dabei werden Teilchen im Feld freigelassen und die durchlaufenen Trajektorien grafisch dargestellt. Die physikalische Kopplung der Geometrieobjekte an das Feld bietet sich für die Implementation dieses Konzepts geradezu an. Hierzu wird eine Vielzahl von Objekten gleichen Typs an einem oder mehreren vorgegebenen Saatpunkten in zeitlich konstanten Abständen freigelassen. Je kürzer diese Abstände sind, umso stärker bilden sich Linenstrukturen im Feld heraus, da die Objekte eines Saatpunkts bei stationären Strömung immer wieder die gleichen Wege durchlaufen. Die Verwaltung der einzelnen Objekte auf der GPU ist allerdings nicht trivial, weil die verwendete fixe Texturdatenstruktur eine Generierung von neuen Objekte praktisch unmöglich macht. Deshalb werden Objekte, die das Randgebiet der Simulation erreichen wiederverwendet, indem sie an die Position ihres Saatpunktes zurückgesetzt werden. Die Identifikation und Behandlung solcher Objekte wird im Abschnitt Randbehandlung besprochen.

Zu Beginn gibt es nur einen Saatpunkt. Der Benutzer kann aber interaktiv weitere Punkte erzeugen. Werden k Saatpunkte benötigt, so wird die Menge der Geometrieobjekte $O = (o_1, o_2, ..., o_n)$ in k disjunkte Saatmengen S_i gleicher Mächtigkeit partitioniert $(i = 1...k)$. Wegen der zyklisch arbeitenden Routine zur Randbehandlung sollten die rechteckigen Objektbereiche der Vertex-Texturen möglichst gleichmäßig auf die Saatpunkte aufgeteilt sein (siehe Abb. 5.6).

Ein Objekt o_i kann sich in zwei Zuständen befinden:

1. *frei* - das Objekt o_i wurde freigelassen und bewegt sich durch das Strömungsfeld
2. *wartend* - das Objekt o_i wartet auf den Zeitpunkt des Freilassens

Die wartenden Objekte werden in k Wartelisten W_i von der CPU verwaltet. Sobald ein Objekt von der Randbehandlungsroutine zurückgesetzt wird, wird in einer Tabelle seine Saatmenge nachgeschlagen und die Objektnummer an das Ende der zugehörigen Warteliste angehängt. Außerdem werden die wartenden Objekte in der Vertex-Fix-Maske markiert, um dadurch zu verhindern, dass sich ihre Positionen bei den folgenden Simulationsschritten ändern. Zusätzlich zu den Wartelisten im Hauptspeicher gibt es für jeden Saatpunkt eine Tiefenmaske D_i im Grafikspeicher. Die Masken D_i markieren die zu den Saatpunkten gehörigen Objekte und werden beim Verschieben einzelner Saatpunkte benötigt.

Das Freilassen der Objekte wird in der Hauptschleife des Programms angestoßen. Unabhängig voneinander ist für jeden Saatpunkt die Periodendauer P_i von gerenderten Bilder festgelegt, nach der ein Objekt am Saatpunkt freigelassen wird. Beim Freilassen müssen die Wartelisten und die Vertex-Fix-Maske aktualisiert werden. Sind genügend Objekte in den Wartelisten vorhanden, ist ein gleichförmiger

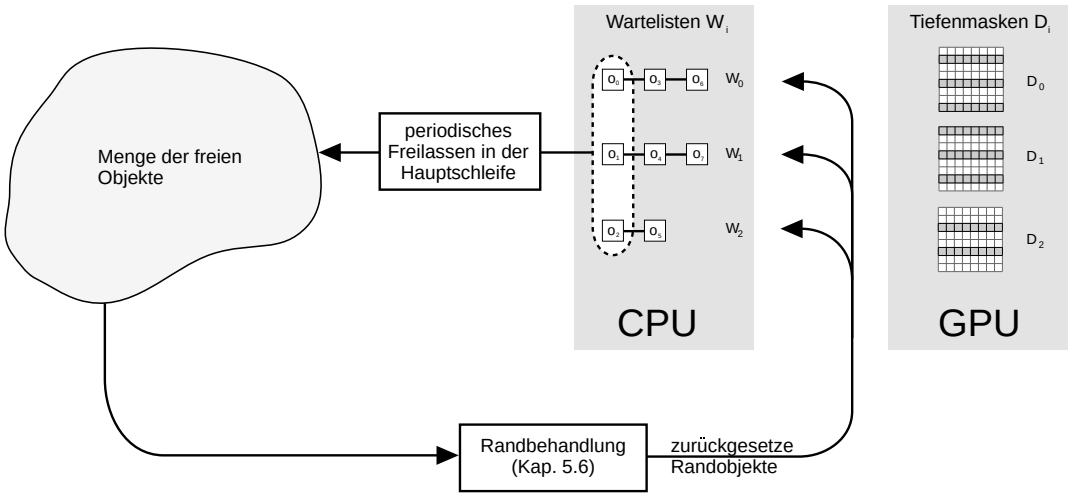


Abbildung 5.6: Beispiel für eine Anordnung mit drei Saatpunkten und insgesamt acht Objekten. Die Gleichmäßige Aufteilung der Objekte auf die Saatpunkte ist an den Objektindices und den Markierungen in den Tiefenmasken erkennbar.

Objektstrom garantiert. Da der Füllgrad der Listen direkt an die Verweildauer im Feld gekoppelt ist kann er abhängig von der Position des Saatpunktes stark variieren. Ist die Menge der wartenden Objekte eines Saatpunkts erschöpft, dann wird automatisch seine Saatperiode erhöht, bis sich wieder ein Gleichgewicht eingestellt hat.

5.3.5 Automatische Platzierung der Versuchsobjekte

In der Regel sind die interessanten Gebiete innerhalb eines Strömungsfeldes nicht im Voraus bekannt. Im Gegensatz zu dichten Visualisierungsverfahren, wird man mit Hilfe der Geometrieadvektion erst nach einigem Experimentieren einen Eindruck vom Feld gewinnen. Eine automatische Platzierung der Versuchsobjekte kann diesen Vorgang beschleunigen. Angedacht wurde eine Positionierung an zufallsgenerierten Orten, die durch eine dreidimensionale Wahrscheinlichkeitsverteilungen festgelegt sind. Dies ermöglicht die gleichmäßige Verteilung der Proben im gesamten Feld oder in räumlich eingeschränkten Unterbereichen des Strömungsvolumens. In der Programmversion, die dieser Arbeit zugrunde liegt, ist diese Methode nicht implementiert.

5.4 Simulation

5.4.1 Integration

Der Simulationszustand ist in drei Vertex-Texturen gespeichert. Zwei davon enthalten die Vertexpositionen der letzten beiden Zeitschritte und in die dritte Textur werden die Positionen des neuen Zeitschritts geschrieben. Die Funktion der drei Texturobjekte als Eingabe- und Ausgabetextur wechselt dabei zyklisch mit jedem Integrationsschritt. Der Ablauf und die Zusammenhänge sind in Abbildung 5.7 darge-

stellt. Ein einziger Renderpass behandelt die komplette Textur und damit alle Objekte. Die Positionen der fixierten Vertices, die in der Vertex-Fix-Maske markiert sind, werden in der Ausgabetextur nicht aktualisiert. Im Integrationsshader wird zusätzlich zur Teilchenintegration die Farbcodierung für die Vertices in der ungenutzten w-Komponente aktualisiert.

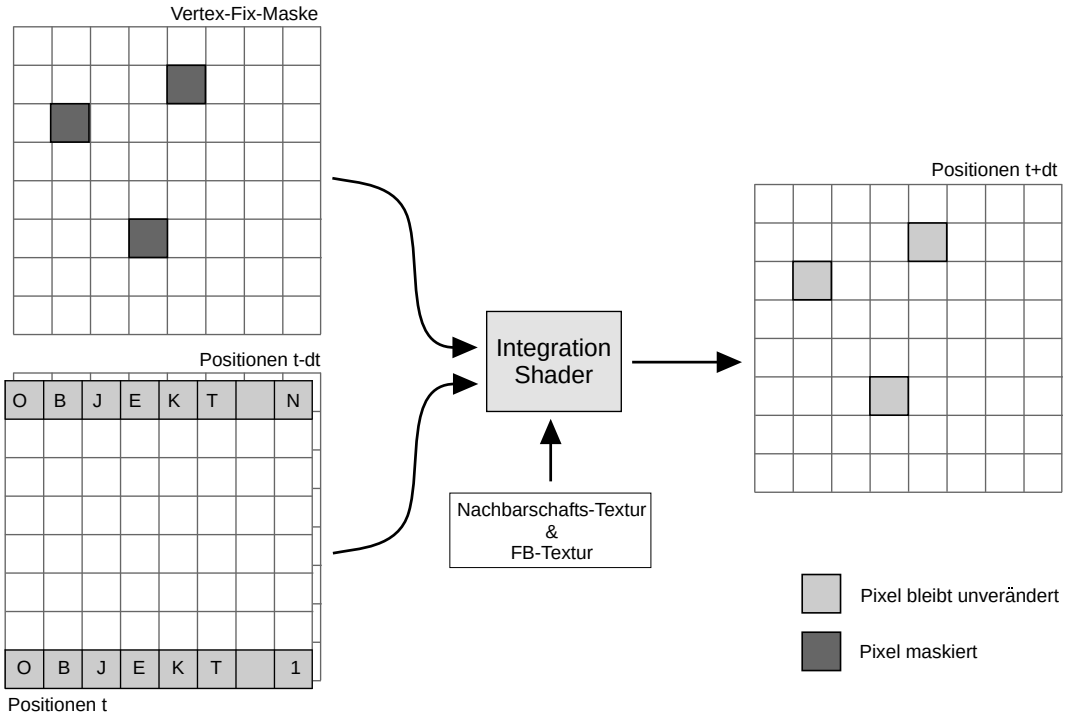


Abbildung 5.7: Bei der Integration der Teilchenpositionen stellt die Vertex-Fix-Maske sicher, dass die fixierten Vertices nicht aktualisiert werden.

5.4.2 Simulationsparameter

Strömungsfelder können sehr unterschiedliche Eigenschaften besitzen, was eine angepasste Wahl der Simulationsparameter erfordert. Eine ungünstige Wahl führt schnell zu unbrauchbaren Resultaten bei der numerischen Integration und der daraus folgenden Visualisierung. Allein schon die große Zahl der frei wählbaren Parameter macht die Auswahl und Abstimmung zu einer schwierigen Aufgabe:

- Größe des Zeitschritts Δt
- Dauer des Interpolationsschritts ΔT zwischen zwei Vektorfeldsamples $\boldsymbol{V_t}$ und $\boldsymbol{V}_{t+\Delta t}$
- Faktor f_{fac} für die Stärke des Strömungsfeldes
- Faktor d_{fac} für die geschwindigkeitsabhängige Dämpfung
- Faktor s_{fac} für die Federkonstanten

Wird der *Zeitschritt* Δt zu groß gewählt, dann treten numerische Instabilitäten auf oder es werden bei der Integration der Teilchenpositionen interessante Bereiche des Feldes übersprungen. Bei der Visualisierung instationärer Felder kommt erschwerend hinzu, dass zusätzlich zur inhärenten Zeiteigenschaft des Vektorfeldes, eine weitere Zeitkomponente durch die Änderung des Feldes hinzukommt. Die Abstimmung der beiden Paramter $(\Delta t, \Delta T)$ aufeinander kann problematisch sein.

Die *Dämpfung* verhindert das Aufschwingen und unerwünschte Beschleunigen der Geometrievertices. Da die Objekte auch durch die Reibung des Feldes abgebremst werden (Stokes'sche Kopplung), sollte ein zu großer Wert vermieden werden, weil sonst jegliche Bewegung schon im Anfangsstadium unterbunden wird. Wird der Faktor für die *Federkonstante* zu groß gewählt, so kann dies negative Auswirkungen auf die Stabiltität des Algorithmus haben, wohingegen ein zu kleiner Wert die Zerstörung der geometrischen Form der Probe durch die Strömung begünstigt.

Feldanalyse

Um eine erste Näherung für die Simulationsparameter zu bekommen, werden die Samples $\boldsymbol{V}_s$ der Vektorfeldzeitschritte nach dem Programmstart statistisch analysiert und dabei charakteristische Werte W wie die Divergenz und die Stärke des Feldes extrahiert. Neben dem Minimum, dem Maximum und dem Durchschnitt wird jeweils auch die Standardabweichung bestimmt. Die gewonnenen Daten werden automatisch in einer Textdatei gespeichert, weshalb bei erneutem Laden des selben Feldes der zeitaufwändige Analyseschritt entfällt. Aus den Teilwerten der Statistik $\boldsymbol{W}$ werden die Startparameter berechnet.

$$\boldsymbol{W} = (W_{\min}, W_{\max}, \overline{W}, \delta W) \tag{5.1}$$

$$W_{\min} = \min_{v \in V_s} f(v) \tag{5.2}$$

$$W_{\max} = \max_{v \in V_s} f(v) \tag{5.3}$$

$$\overline{W} = (\sum_{v \in V_s} f(v)) / |V_s| \tag{5.4}$$

$$\delta W = (\sum_{v \in V_s} (f(v) - \overline{W})^2) / |V_s| \tag{5.5}$$

Der mittlere Wert der *Vektornorm N* entspricht der mittleren Stärke des Feldes $(W = N, f = norm(v))$. Zusammen mit dem Zeitschritt Δt ergibt sich die Beziehung 5.5, die dafür sorgt, dass im Durchschnitt keine Feldsamples übersprungen werden. Um dies im gesamten Strömungsgebiet sicherzustellen, kann auch der maximale Wert N_{max} statt des Durchschnitts verwendet werden.

$$\Delta t \cdot \overline{N} \cdot f_{\text{fac}} < 2 \tag{5.6}$$

In die Divergenz-Statistik $\boldsymbol{\nabla}$ geht der Betrag der Divergenz am Ort $\boldsymbol{x}$ der Feldsamples ein:

$$f(v) = |\nabla \boldsymbol{V}(\boldsymbol{x}(v))| = |\boldsymbol{V}_x + \boldsymbol{V}_y + \boldsymbol{V}_z|_{\boldsymbol{x}} \tag{5.7}$$

An Stellen großer Divergenz besteht bei zu kleinem Federfaktor s_{fac} oder Dämpfungsfaktor d_{fac} die Gefahr einer starker Überdehnung der Versuchsobjekte (angedeutet in Abb. 5.8), bis hin zum totalen

Verlust der ursprünglichen Form. Diese Faktoren sollten deshalb mit steigendem $\overline{\nabla}$ auch größer gewählt werden. Im Folgenden sind beispielhaft die implementierten Formeln für die automatische Wahl in Abhängigkeit der Divergenz- und der Normstatistik angegeben. Zusätzlich wird darauf geachtet, daß Gleichung 5.5 eingehalten wird.

$$s_{\text{fac}} = (0.1 * (\nabla_{max} - \nabla_{min}) + 0.9 * \overline{\nabla}) * 70.0 \tag{5.8}$$

$$\Delta t = 0.01/(0.2 * N_{max} + 0.8 * \overline{\nabla}) \tag{5.9}$$

Die Formeln wurden empirisch durch Versuche mit verschiedenen Feldern gewonnen. Eine besondere Bedeutung kommt ihnen nicht zu. Für die meisten Versuchsfelder liefern sie einigermaßen brauchbare Resultate. Allerdings sind stark inhomogene Felder weiterhin kritisch, was eine sorgfältige Parameterabstimmung durch den Benutzer notwendig macht.

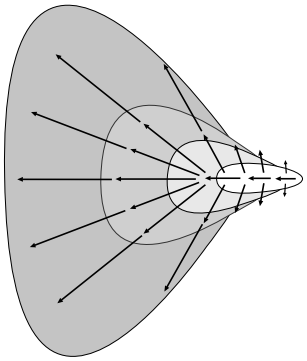


Abbildung 5.8: Extreme Expansion eines Versuchsobjekts in einem divergenten Strömungsgebiet

5.5 Randbehandlung

Ausserhalb des eigentlichen Simulationsgebietes wird das Vektorfeld künstlich auf Null gesetzt. Versuchobjekte, die an den Rande des Gebietes gelangen, werden somit in der Regel dort als unschöne Artefakte verweilen (Abb. 5.9). Da die GPU darauf ausgelegt ist, große Menge von Daten auf einmal zu verarbeiten und die Geometriepositionen gesammelt in einer Textur vorliegen entsteht ein Problem, sobald Entscheidungen auf Grund der Positionen einzelner Objekte getroffen werden sollen. Die Randbehandlung kann daher nicht mit der Integration kombiniert werden. Dort werden Massepunkte einzeln, aber keine Objekte als Ganzes betrachtet. Ein eigenständiger Algorithmus, der zu einem großen Teil auf der GPU implementiert ist und kein Rücklesen der Positionen erfordert, identifiziert Randobjekte und leitet entsprechende Maßnamen ein.

Soll ein Objekt darauf getest werden, ob es sich am Rand befindet, dann wird auf seinen Vertexpositionen (Abb. 5.9, links) ein Fragmentprogramm ausgeführt. Dieses verwirft die einzelnen Fragmente, falls die zugeordnete Teilchenposition außerhalb des Strömungsgebietes liegen. Über eine Occlusion-Query läßt sich daraufhin die Anzahl der gerenderten Fragmente abfragen. Wird ein vorgegebener Schwellenwert unterschritten, dann setzt das Programm das Objekt an die Positionen zurück, die in einer Reset-Textur gespeichert sind. Diese Textur wird bei Programmbeginn mit den Startpositionen der Geometrie

initialisiert, kann aber auch vom Benutzer manipuliert werden. Da das Testen aller Objekte relativ zeitaufwändig ist, wird pro gerendertem Bild nur eine beschränkte Anzahl k von Objekten getestet. Die Liste der Objekte wird dabei zyklisch durchlaufen. Je nach Wahl des Parameters k kann es deshalb mehrere Durchläufe der Hauptschleife dauern, bis ein Randobjekt identifiziert und zurückgesetzt ist.

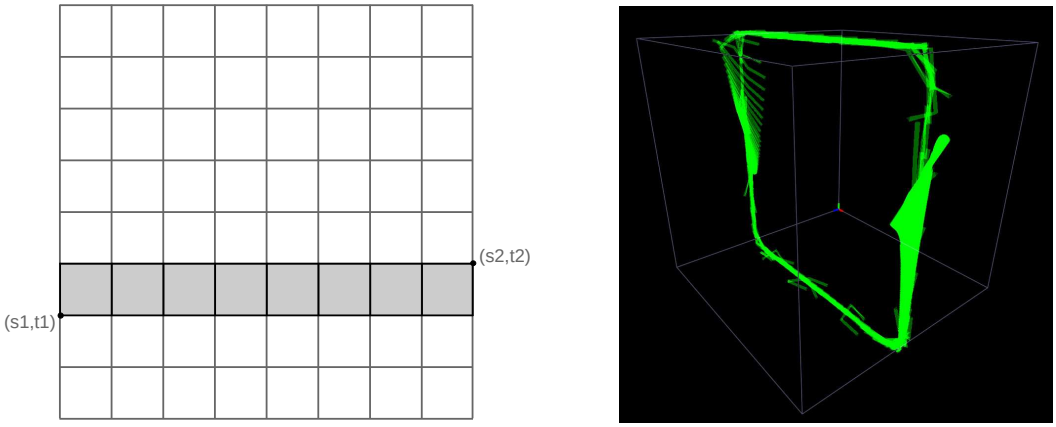


Abbildung 5.9: Der Bereich eines einzelnen Objekts in der Vertex-Textur (links), Artefakte (rechts)

Eine Behandlung innerhalb des Feldes, an Stellen großer Divergenz, wurde nicht implementiert. Das Funktionsprinzip eines solchen Algorithmus ist aber dasselbe wie bei der Randbehandlung. Allerdings reduziert eine Berechnung der Divergenz während der Simulation die Performance zu stark, weshalb sich ein Vorverarbeitungsschritt anbietet, der die berechneten Werte in der w-Komponente der Vektorfeld-Textur speichern könnte.

5.6 Rendering

Die grafische Darstellung der sich bewegenden Versuchsobjekte ist die Schnittstelle zwischen den Simulationsdaten und dem Benutzer, der diese interpretieren muss. Dem Rendern kommt deshalb eine wichtige Bedeutung zu. Bei einem möglichst hohen Informationsgehalt soll die Darstellung übersichtlich und interpretierbar bleiben. Das Programm rendert die Vorder- und Rückseiten der Objektoberflächen, die sich aus Dreiecksnetzen zusammensetzen. Dreiecke, die Stützvertices enthalten, werden aber nicht dargestellt. Es sind drei Rendermodi implementiert. Neben einem einfachem *Flat-Shading*, gibt es eine *Wireframe-Darstellung* und eine auf den Vertexnormalen basierende *Beleuchtungsberechnung*.

Die Liste der Oberflächendreiecke aller Objekte ist als *Index-Array* in einem statischen *Vertex Buffer Object* abgelegt. Pro Dreieck werden drei Indices angegeben. Die Indices verweisen in ein *Vertex-Array*, in welchem statt der Vertexposition die Texturkoordinaten (u,v) abgespeichert sind. Die wirkliche Vertexposition erhält das Rendering-Vertexprogramm durch einen Texturlookup in der aktuellen Vertex-Textur. Da sich die beiden Arrays während des Programmlaufs nicht verändern, können sie beim Programmstart durch einen einmalige Kopieroperation auf der Grafikkarte gespeichert werden. Das Rendern aller Objekte kann deshalb durch einen einzigen effizienten OpenGL-Aufruf angestoßen werden.

Die Vertex-Farbe berechnet sich aus der w-Komponente des nachgeschlagenen Texels:

- w = -2 Vertex ist selektiert (blau)
- w = -1 Vertex ist fixiert (weiss)
- w >= 0 Farbkodierung eines physikalischen Parameters am Ort des Vertex

Liegt die Kodierung eines physikalischen Parameters vor, dann wird, basierend auf seinem Wert zwischen roter und grüner Farbe linear interpoliert. Rote Vertices können so zum Beispiel starke Kräfte oder hohe Geschwindigkeiten visualisieren. Wenn sich eine große Anzahl von Objekten im Versuchsgebiet befinden, dann kommt es unumgehbar zu gegenseitigen Verdeckungen und Überschneidungen. Eine transparente Darstellung, basierend auf *Alpha Blending*, kann in diesem Fall für mehr Übersichtlichkeit sorgen. Üblicherweise müssen die transparenten Objekte vor dem Rendern in aufsteigender Entfernung zum Betrachter sortiert werden. Für die große Menge der Objekte ist dies allerdings nicht praktikabel. Aufgrund der variierenden Form der Probegeometrie müssten sogar alle Dreiecke sortiert werden. Implementiert wurde deshalb eine Art *Pseudotransparenz*, bei der die einzelnen Objektfragmente unabhängig von der Reihenfolge additiv geblendet werden. Der dreidimensionale Eindruck leidet dabei allerdings etwas.

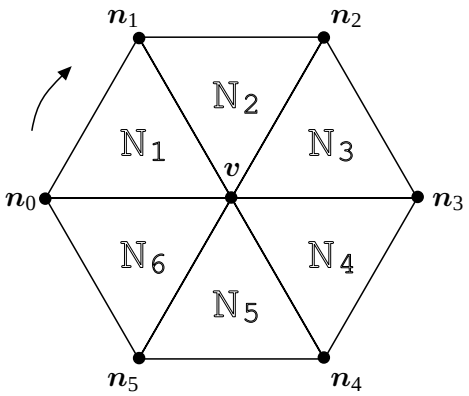
Bei der Wireframe-Darstellung werden nur die Kanten der Dreiecke dargestellt. Verwendet wird hierfür der eingebaute OpenGL-Mechanismus. Im Flatshading-Modus ergeben sich die Farben der inneren Dreiecksfragmente aus der linearen Interpolation der Vertex-Farben des Dreiecks. Erst eine zusätzliche Beleuchtungsberechnung, die die Normalen der Objektoberfäche verwendet, ermöglicht einen wirklichen 3D-Effekt. Die Normale eines Vertex entspricht dem Mittelwert der Normalen aller angrenzenden Dreiecke. Aufgrund der Formveränderung der Geometrie müssen die Normalen für jeden Zeitschritt neu bestimmt werden. Es bietet sich an, die Berechnungen im Integrationsshader durchzuführen. Dort werden die benötigten Positionen der benachbarten Vertices schon für die Integration nachgeschlagen. Wählt man die Reihenfolge der Nachbarvertices in der FB-Textur korrekt, dann kann die Berechnung der Normalen (siehe Abb. 5.10) in der selben Schleife durchgeführt werden, in der auch die Federkräfte aufsummiert werden. Dadurch spart man sich erneute Texturlookups für die Vertexpositionen. Die Positionsintegration findet erst nach der Schleife statt, deshalb werden die Normalen des letzten Zeitschritts berechnet. Aufgrund der hohen Bildraten und der winzigen Positionsänderungen kann dieser Effekt aber vernachlässigt werden. Problematischer ist die Tatsache, dass für die Normalenberechnung an den Schlauchenden und bei Objekten mit inneren Vertices gesonderte Ausnahmebehandlungen nötig sind. Die endgültige Fragmentfarbe wird im Fragmentprogramm des Renderers berechnet:

$$\text{frag}_{farbe} = blend_{\text{fac}} * \text{farbe} + (1 - blend_{\text{fac}}) * \text{farbe} * (\boldsymbol{N}_v \cdot \underbrace{(0,0,1)}_{\text{Lichtvektor}}) \tag{5.10}$$

Die Normalen werden dazu vorher in das Koordinatensystem des Betrachters transformiert. Über einen Blendfaktor kann der Benutzer fliessend zwischen einfachem Flatshading und dem beschriebenen Beleuchtungsmodell wählen.

5.7 Vektorfeld-Streaming

Instationäre Vektorfelder $\boldsymbol{V}(\boldsymbol{x},t)$ sind als eine Reihe von N Vektorfelddatensätzen $\boldsymbol{V}_i$ gespeichert, die das veränderliche Feld bei einer konstanten Abtastrate approximieren. Bei den Geometrieadvektions-Berechnungen auf der GPU werden zwei aufeinanderfolgende Zeitschritte des Feldes benötigt, da die



$$\boldsymbol{N}_1 = (\boldsymbol{n}_1 - \boldsymbol{v}) \times (\boldsymbol{n}_0 - \boldsymbol{v})$$
$$\boldsymbol{N}_2 = (\boldsymbol{n}_2 - \boldsymbol{v}) \times (\boldsymbol{n}_1 - \boldsymbol{v})$$
$$\dots$$
$$\boldsymbol{N}_6 = (\boldsymbol{n}_0 - \boldsymbol{v}) \times (\boldsymbol{n}_5 - \boldsymbol{v})$$
$$\boldsymbol{N}_{\boldsymbol{v}} = (\boldsymbol{N}_1 + \dots + \boldsymbol{N}_6)/6$$

Abbildung 5.10: Normalenberechnung im Integrations-Fragmentprogramm. Die Reihenfolge ist durch die for-Schleife festgelegt.

Feldsamples linear über die Zeit interpoliert werden. In Abbildung 5.11 ist die Integrationsschrittweite Δt und die Interpolationsdauer ΔT zu erkennen, nach der spätestens ein neuer Vektorfeldschritt vorliegen muss, damit die Berechnung nicht ins Stocken gerät. Da die üblichen Texturupload-Methoden von OpenGL den Aufrufer blockieren, muss der Datensatz in kleinen Blöcken in die Textur geladen werden. Würde man die komplette Textur in einem Schritt aktualisieren, dann würde das komplette System kurz stocken. Dies ist bei einer interaktiven Visualisierung mit möglichst konstanter Bildrate unerwünscht. Weil das Nachladen der Daten von der Festplatte in den Hauptspeicher auch synchron abläuft, wurde ein zweigeteiltes Producer-Consumer-System entworfen. Dieses besteht aus dem *Streamer*, der direkt in der Hauptschleife des Programms aufgerufen wird und dem *Lader*, der davon unabhängig in einem seperaten Thread verwaltet wird. Die beiden Komponenten teilen sich eine festgelegt Anzahl von Feldpuffern, die sie beide zyklisch verwenden. Für die nötige Synchronisation sorgt dabei ein Zählsemaphore. Als Thread-API wurde die plattformunabhängige pThread-Bibliothek ausgewählt [22]. Im Folgenden wird die Funktionsweise des Streamers und des Laders anhand zweier Ablaufdiagramme erläutert, die in Abbildung 5.12 zu sehen sind.

Lader

Der Lader kopiert die Felddaten von der Festplatte in den nächsten freien Feldpuffer im Arbeitsspeicher und führt dabei eine Konvertierung in das fp16-Format durch. Ist eine Zeitschritt $\boldsymbol{V}_i$ vollständig geladen und das Semaphore blockiert nicht, so wird sofort mit dem Laden des nächsten Zeitschritt $\boldsymbol{V}_{i+1}$ begonnen. Sollte das zeitliche Ende des Datensatzes erreicht werden, dann wird wieder beim ersten Zeitschritt $\boldsymbol{V}_1$ angefangen.

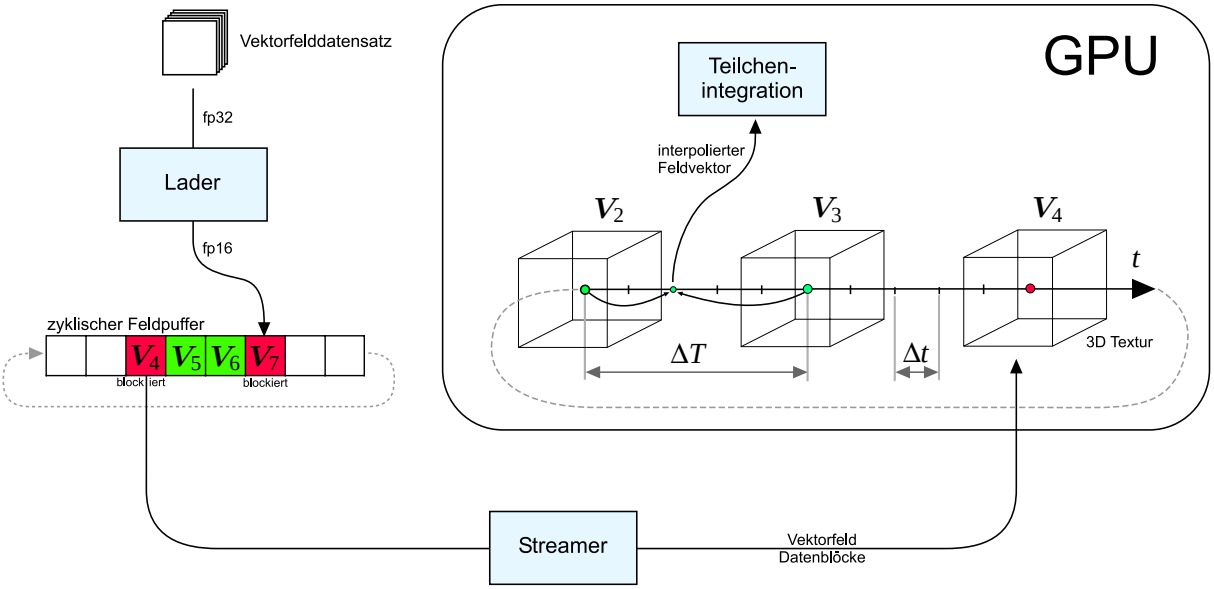


Abbildung 5.11: Zusammenhänge beim Streaming der Vektorfelddaten instationärer Strömungsfelder. Zu erkennen sind auch die zeitlichen Zusammenhänge bei der Simulation.

Streamer

Auf der GPU stehen drei 3D-Texturen im fp16-Format zur Verfügung, von denen zu jeder Zeit eine vom Streamer mit den Daten des neuen Zeitschritts befüllt wird. Mit jedem Durchlauf der Hauptschleife kann genau eine Zustandsänderung des Streamers hervorgerufen werden. Dabei wird jeweils die Aktion ausgeführt, die in den abgerundeten Kästchen 5.12 angegeben ist:

Der Streamer befindet sich zunächst im Zustand ’gestoppt’. Der Textur-Upload kann gestartet werden, sobald ein neuer Zeitschritt im Hauptspeicher vorliegt und ein freies Texturobjekt auf der Grafikkarte zur Verfügung steht. Die Größe der Datenblöcke, die pro Aufruf hochgeladen werden, ist abhängig von der aktuellen Bildrate, der Größe des Datensatzes eines Zeitschrittes und beträgt immer ein Vielfaches der x-Auflösung des Feldes. Durch die geschickte Wahl einer geeigneten Größe (1) versucht der Streamer, den Upload möglichst gleichmäßig über die Interpolationsdauer ΔT der zwei letzten Vektorfeldzeitschritte aufzuteilen. Dies sorgt für eine möglichst konstante Bildrate und garantiert eine flüssige Animation.

Befindet sich das System im Zustand ’gestartet’, dann wird pro Durchlauf der Hauptschleife ein Datenblock auf die Grafikkarte geladen (2). Dies wird durch *OpenGL Pixel Buffer Objekte* realisiert, die das Kopieren einzelner Texturblöcke in einen vom Grafikkartentreiber verwalteten Speicherbereich ermöglichen. Der C-Code eines solchen Kopiervorgangs ist im Listing 5.2 in den Zeilen 3-5 angegeben. Da die Texturdaten durch normale Speicheroperationen in den gebundenen Speicherbereich geschrieben werden, müssen die Vektorfelddaten im fp16-Format der 3D-Texturen vorliegen. Durch den Aufruf von *glTexSubImage3D* in Zeile 8 werden die Daten im Grafikkartenspeicher aktualisiert. Sobald ein Zeitschritt vollständig hochgeladen wurde, geht der Streamer wieder in den Zustand ’gestoppt’ über (3).

Ursprünglich wurde der Aufruf der Zeilen 7 und 8 statt für jeden Datenblock, einmal am Ende eines

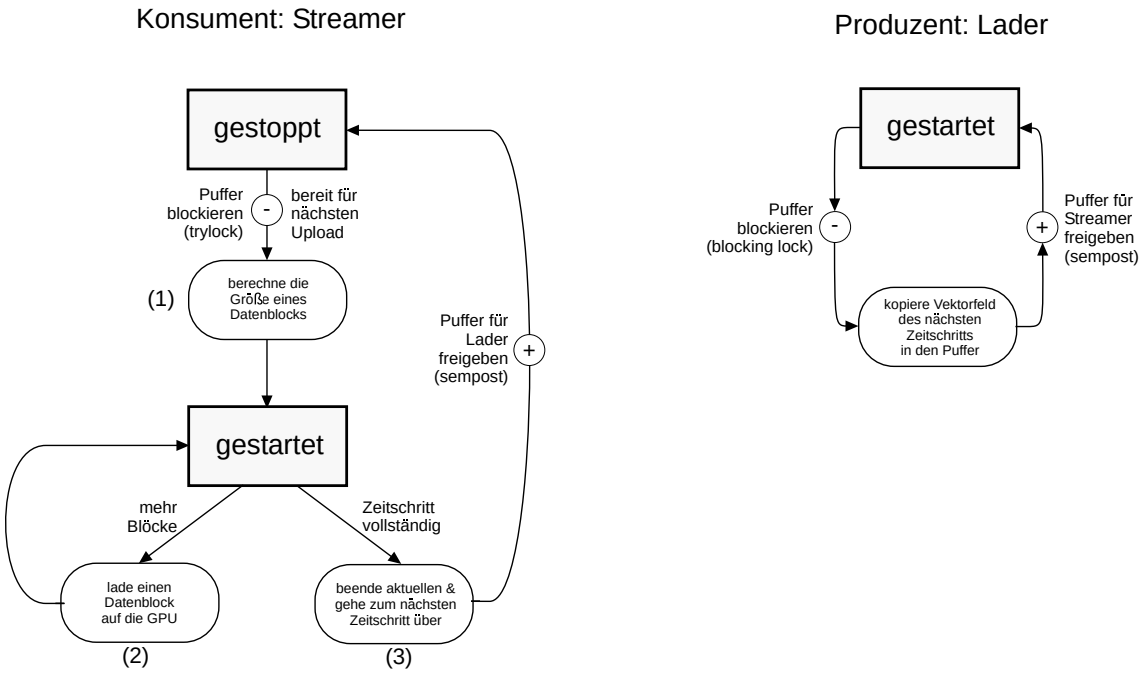


Abbildung 5.12: Ablaufgraph des Streamers und des Laders

Zeitschritts für den vollständigen Datensatz aufgerufen. PBOs versprechen bei einem glTexSubImage-Aufruf Zeitvorteile gegenüber einem direkten Texturupload der Daten ohne gebundenen Grafikspeicher. Trotzdem dauerte die Ausführdauer exakt gleich lange. Die dabei auftretende Latenz hat sich jeweils am Ende eines Zeitabschrittes durch Stocken bemerkbar gemacht. Deshalb wurde der Aufruf auf die einzelnen Datenblöcke verteilt. Leider funktioniert das Streaming auch so nicht perfekt. Ist es aktiviert, dann stockt das System in längeren, aber nicht vorhersagbaren Zeitabständen. Zeitmessungen haben ergeben, dass der Simulationsteil während der problematischen Bilder bis zu einhundert mal länger braucht. Möglicherweise stehen die Texturdaten trotz des Aufrufs von glTexSubImage3D nicht sofort auf der Grafikkarte zur Verfügung.

Listing 5.2: Upload eines Datenblocks im Streamer

```
1  glBindBuffer(GL_PIXEL_UNPACK_BUFFER_EXT, _vf_pbo[_streamer_buffer]);
2
3  glMapBuffer(GL_PIXEL_UNPACK_BUFFER_EXT, GL_WRITE_ONLY),
4  memcpy(kopiere einen Datenblock);
5  glUnmapBuffer(GL_PIXEL_UNPACK_BUFFER_EXT),
6
7  glBindTexture(GL_TEXTURE_3D, _vf_textures[_streamer_buffer]);
8  glTexSubImage3D(GL_TEXTURE_3D, Auswahl des kopierten Texelbereichs,
9                  GL_RGB, GL_HALF_FLOAT_ARB, Offset in der 3D Textur);
10
11 glBindBuffer(GL_PIXEL_UNPACK_BUFFER_EXT, 0);
```

6 Auswertung

6.1 Einleitung

In diesem Kapitel werden die Ergebnisse der Geometrieadvektions-Methode besprochen. Im Abschnitt Performancemessung wird die Geschwindigkeit des Programms untersucht, worauf eine Analyse der Visualisierungsresultate anhand mehrerer Beispiele erfolgt.

6.2 Performancemessung

Die Messungen der Performance wurden auf einem PC durchgeführt, der mit 1024 MB Arbeitsspeicher, einem Intel Core 2 Duo E6300 Prozessor und einer Nvidia Geforce 7900 GS mit 256 MB Grafikspeicher ausgestattet ist. Beide Kerne dieser CPU sind mit 1867 Mhz getaktet. Der verwendete Grafikchip verfügt über 20 Renderingpipelines, die sich die 20 Fragmenteinheiten und 7 Vertexeinheiten teilen. Seine Leistung entspricht etwa der einer Geforce 7900 GT. Bei den Messungen wurde ein stationäres Vektorfeld mit einer Auflösung von $128 \times 128 \times 128$ Samples verwendet und die Streamingkomponente ausgeschaltet. Es wurden vier Versuchskonfigurationen A, B, C und D mit einer variierenden Zahl von Kugelobjekten und Subdivision-Schritten ausgewählt.

Konfiguration	**A**	**B**	**C**	**D**
Anzahl der Kugeln	128	1024	2048	1024
Subdivision Schritte	1	1	1	4
Vertices pro Objekt	18	18	18	258
Vertices Gesamt	2304	18432	36864	264192
Anzahl der Federn	6144	49152	98304	786432

Tabelle 6.1: Es wurden vier Versuchskonfigurationen untersucht. Jede Federverbindung zwischen den Vertices wurde nur einmal gezählt. Die FB-Textur enthält die doppelte Zahl an Einträgen.

Für Konfiguration D benötigen die Vertex-Texturen, die Nachbar-Textur und die Reset-Textur zusammen 20 MB Grafikspeicher. Dazu kommen 24 MB Speicherbedarf der FB-Textur und 16 MB für die Vektorfelddaten. Zusätzlich müssen noch die Vertex-Arrays und die Tiefentexturen gespeichert werden. Dies bedeutet, dass das Programm in dem betrachteten Fall etwa 65 MB Grafikspeicher verwendet. Bei der Visualisierung instationärer Strömungen muss beachtet werden, dass zwei zusätzliche 3D-Texturen für die Vektorfelddaten benötigt werden.

Für die Zeitanalyse wurden die Objekte gleichmäßig im Strömungsgebiet verteilt und die Randbehandlung der Objekte zunächst ausgeschaltet. Gemessen wurde die Gesamtzeit eines Durchlaufs der Hauptschleife t_{ges} sowie die dabei benötigte Zeit für das Rendering t_{render} und den Integrationsschritt t_{int}. Außerdem wurden für jede Konfiguration drei zusätzliche Meßungen bei aktivierter Randbehandlung durchgeführt, wobei die Anzahl der dabei getesteten Objekte pro Simulationsschritt zwischen 1, 10 und 50 Objekten variierte. In der folgenden Tabelle sind die Ergebnisse aufgelistet, wobei die Zeiten der Randtestmessungen $\Delta t_{\text{Randtest}}$ additiv zur Gesamtzeit zu sehen sind.

Konfiguration	**A**	**B**	**C**	**D**
t_{ges}	0.38 ms	1.53 ms	2.85 ms	24.26 ms
t_{int}	0.23 ms	0.82 ms	1.50 ms	14.57 ms
t_{render}	0.15 ms	0.71 ms	1.35 ms	9.69 ms
$\Delta t_{\text{Randtest}}(1)$	+0.14 ms	+0.12ms	+0.14 ms	+0.25 ms
$\Delta t_{\text{Randtest}}(10)$	+0.17 ms	+0.15ms	+0.18 ms	+0.27 ms
$\Delta t_{\text{Randtest}}(50)$	+0.18 ms	+0.26ms	+0.30 ms	+0.40 ms

Tabelle 6.2: Resultate der Zeitmessung

Bei den kleineren Systemen werden bei eingeschalteter Integration mehrere hundert Bilder pro Sekunde berechnet. Dies garantiert eine schnelle und flüssige Animation der Objektbewegungen. Auch im Fall des größten Systems berechnet und visualisiert das Programm noch 40 Zeitschritte pro Sekunde. Allerdings muss die Schrittweite der Integration mit zunehmender Vertexzahl erhöht werden, da sonst die Positionsänderungen pro Zeitschritt zu klein sind und die Objekte sich dementsprechend langsam bewegen. Trägt man die benötigte Gesamtzeit t_{ges} und die für den Randtest zusätzlich benötigte Zeit $\Delta t_{\text{Randtest}}$ über der Anzahl der Vertices auf, dann sieht man, dass das System linear skaliert, wobei die Zeit für die Randbehandlung nur schwach von der Objektzahl abhängt.

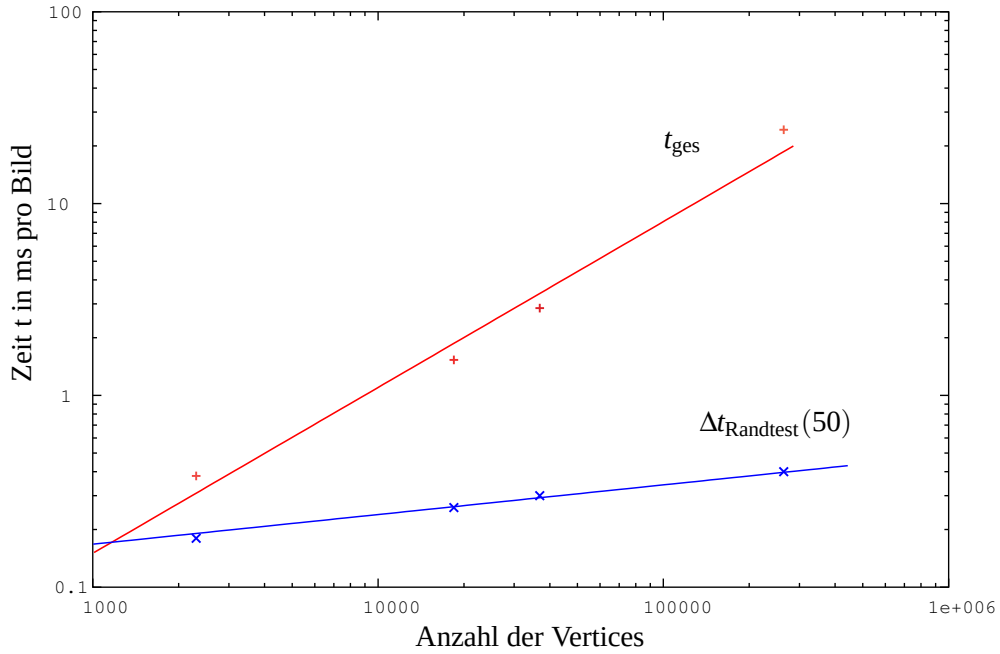


Abbildung 6.1: Die Dauer eines Zeitschritts skaliert linear mit der Größe des simulierten Systems.

Abgesehen von gelegentlichem Stocken funktioniert auch die Visualisierung inhomogener Felder, wobei die Bildrate gegenüber stationärer Felder etwas sinkt. Allerdings ist das Programm hierbei auf einen Prozessor mit mehreren Kernen angewiesen, da sich sonst das Nachladen der Vektorfeldzeitschritte im Lader-Thread durch eine stark reduzierte Performance bemerkbar macht.

6.3 Resultate der Visualisierung

Im Folgenden werden die Resultate der Visualierung vorgestellt. Diese hängen stark von den initialen Positionen, des Typs und der räumlichen Größe der Objekte ab. Den Visualisierungen liegt das stationäre Strömungsfeld eines Tornados zugrunde, das in einem 128^3 Datensatz gespeichert ist. Neben einem stark divergenten Bereich mit einer hohen Geschwindigkeit in der Nähe des Wirbels, der in Richtung der z-Achse (blau) ausgerichtet ist, ist das Feld am Rand vergleichsweise schwach und weniger divergent. Dies stellt die Visualisierung der Objekte durch Deformationen vor eine große Herausforderung. Einen wirklichen Eindruck über die Stärke und die Richtung der Strömung erhält man nur bei einer animierten Darstellung am Computer. Auf den gedruckten Bildern kann die Bewegungsrichtung anhand der Ausrichtung und Verformung der Objekte nur erahnt werden. Außerdem muss beachtet werden, dass die Objekte auch zu unterschiedlichen Zeiten im Feld freigelassen worden sein können. Zusätzlich zu der geometrischen Form der Objekte ist die Strömungsstärke an der Farbe der Objekte zu erkennen. Alternativ kann man andere Größen, wie die Beschleunigungskraft oder die Federkräfte farblich kodieren.

Kugeln

Im ersten Versuch wird eine große, hochaufgelöste Kugel in der Mitte des Feldes platziert und die Simulation gestartet. In Abbildung 6.8 ist der Ablauf der Animation in einer Serie von Bildern festgehalten. Bei einer transparenten Darstellung lassen sich auch Bereiche der Kugeloberfläche erkennen, die dem Benutzer nicht zugewandt sind. Man sieht, wie sich die Geometrie auf der Achse des Tornados zu verformen und zu verwirbeln beginnt. Nach und nach wird das komplette Objekt in den Wirbel gezogen. Über die Verdrillung und die farbliche Kodierung ist der Wirbel und die Stärke des Feldes gut erkennbar. Auf einen stabilisierenden Vertex im Inneren der Kugel wurde hier bewusst verzichtet, da sich die Oberfläche sonst kaum verformt. Außerdem bewegt sich ein solches Objekt nur sehr langsam, da aufgrund seiner Größe verschiedene, entgegengesetzte Kräfte an der Oberfläche angreifen, die sich gegenseitig aufheben.

In Abbildung 6.2 (links) sind die Resultate eines Versuchs zu sehen, bei dem die Feldeigenschaften durch die Deformationen mehrer mittelgroßer Kugeln visualisiert werden. An der Verformung der Kugeln ist die Divergenz des Feldes zu erkennen. In den äußeren, homogeneren Bereichen ähnelt die Darstellung der von Streambubbles. Wie an den inneren Objekten zu sehen ist, kommt es aber auch vor, dass die Kugelform zerstört wird.

Alternativ zu wenigen großen Objekten kann man auch eine große Anzahl von kleineren Objekten im Feld freilassen. Entweder werden die Objekte vom Strudel erfasst oder sie treiben in schwächere Aussenbereich des Feldes, wo sie dann am Rand des Strömungsgebietes ankommen und zurückgesetzt werden. Bei einem solchen Versuchsaufbau läßt sich die Feldstruktur vor allem aus der Bewegung der Objekte ablesen. Wird aber das Feld über die Konstante f_{fac} verstärkt und die Federkonstante s_{fac} relativ klein gewählt, dann ergeben sich Resultate wie in Abbildung 6.2 (rechts). Hier ist die starke Überdehnung der Kugelobjekte durch die hohe Divergenz im inneren Feldbereich sowie der grobe Verlauf der Strömungslinien gut zu erkennen.

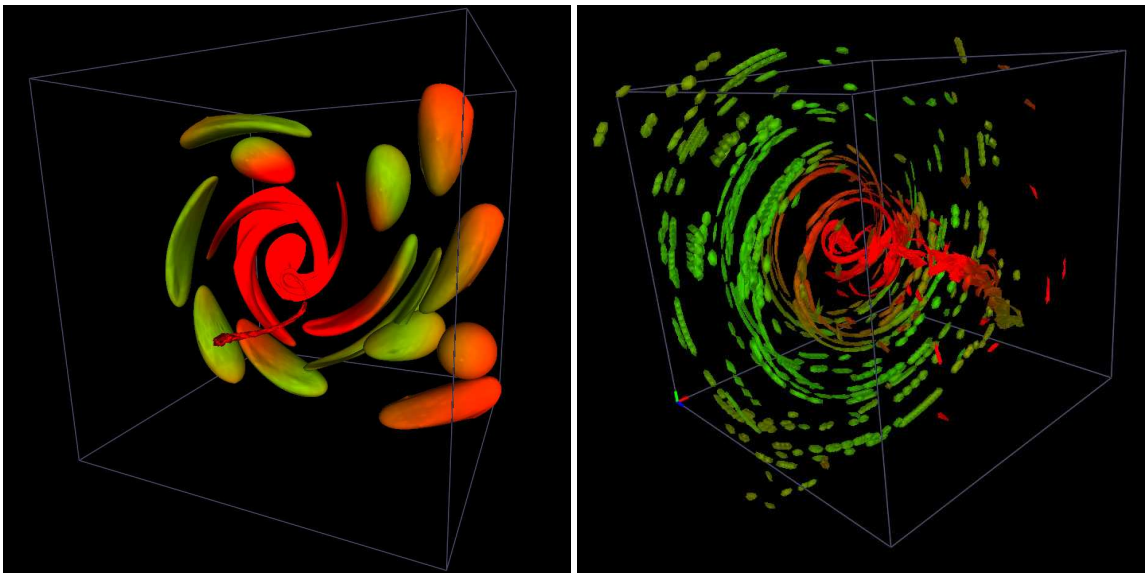


Abbildung 6.2: Die Visualisierung hängt sehr von der Anzahl und Größe der Versuchsobjekte ab. Im linken Bild sind 15 Kugelobjekte mittlerer Größe und rechts mehrere hundert kleine Kugeln zu sehen.

Im letzten Versuch mit Kugeln werden diese an mehreren Saatpunkten freigesetzt. Die Größe der Saatperiode unterscheidet sich dabei in den beiden Bildern der Abbildung 6.3. Da die Objekte in einem stationären Feld immer wieder die gleichen Bahnen durchlaufen, ergeben sich Strukturen, die mit zunehmender Zahl freier Objekte verstärkt hervortreten und Stromlinien ähneln. Beim Betrachten der animierten Darstellung verschmelzen die benachbarten Teilchen eines Saatpunktes zu einem sich bewegenden Gesamtobjekt. Wird dagegen die Saatfrequenz verkleinert, dann verkleinert sich auch die Zahl der freien Objekte im Strömungsfeld, wobei die einzelnen Teilchen und ihre Deformationen besser sichtbar werden. Im Versuchsaufbau wurden zehn Saatpunkte in der Nähe des Wirbelzentrums platziert, die als weisse Kugeln zu erkennen sind. Die Kugeln zweier Saatpunkte werden dabei in den Wirbel hineingezogen, wobei die Objekte eines dieser Saatpunkte selektiert wurden und blau eingefärbt sind. Dabei ist zu sehen, dass die Kugelform der Objekte im Inneren des Strudels aufgrund der extremen Kräfte, die auf das zugrundeliegende Feder-Masse-System wirken, zerstört wird. Dieser Effekt ist in der Abbildung 6.4 einer Nahansicht in Wireframe-Darstellung gut zu erkennen. Da in diesem Fall die kollabierten Objekte relativ schnell am Rand ankommen und zurückgesetzt werden, stellt dies kein großes Problem dar und kann sogar durch die extreme Verformung die Turbulenz des Feldes verdeutlichen. Von den restlichen acht Saatpunkten, deren Objekte nicht so stark deformiert werden, gehen Linien aus, die nach mehreren kreisförmigen Bewegungen um die Achse des Tornados letztendlich am Rand des Gebietes enden. Erhöht man interaktiv die Masse der Objekte, und damit ihre Trägheit, dann führen die Versuchsobjekte weniger Umdrehungen durch, da sie sich nicht so schnell an die Geschwindigkeitsänderung des Feldes anpassen können. Bei einem Massefaktor von 1 erhält man eine Visualisierung, die einer einfachen Advektion masseloser Teilchen nahe kommt. Für die Visualisierung in Abbildungen 6.5 wurde ein negativer Feldfaktor gewählt und die Anordnung der Saatpunkte geändert. Aufgrund der unterschiedlichen Position der Saatpunkte werden dabei die Form und die Ausmaße des Tornado-Rüssels besser sichtbar.

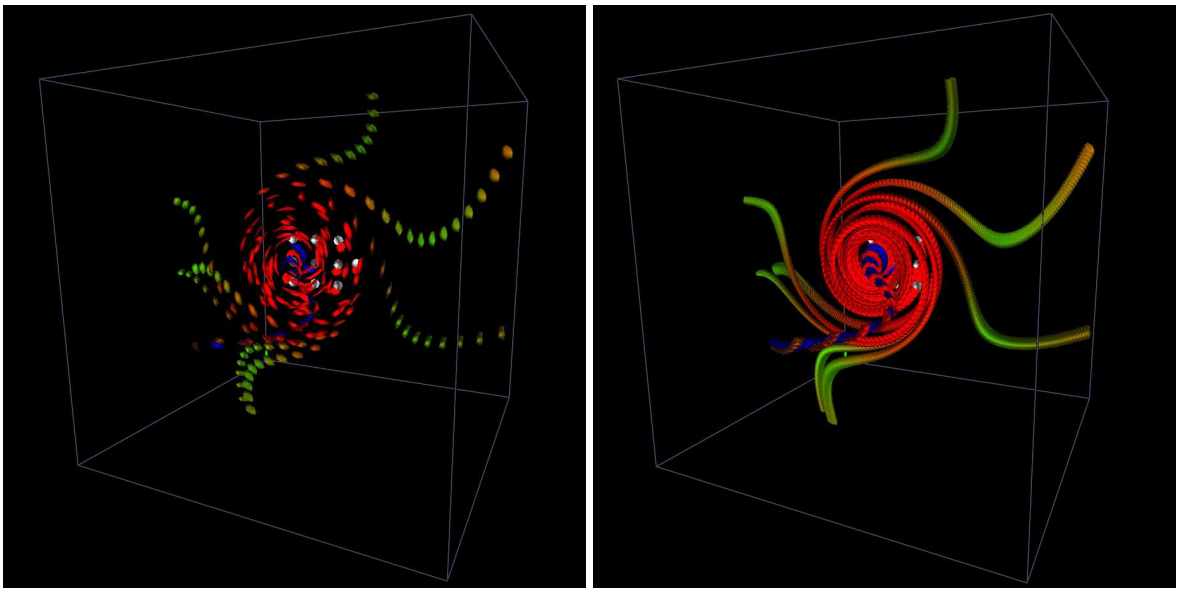


Abbildung 6.3: Kleine Kugelobjekte, die an zehn Saatpunkten freigelassen wurden. Die Bilder unterscheiden sich durch eine unterschiedliche Wahl der Periodendauer. Im linken Bild sind 194 freie Kugeln und im rechten 1500 freie Kugeln zu sehen.

Schläuche

Dank ihrer länglichen Form können sich die Schläuche besser als Kugeln am Feld ausrichten. Trotzdem dürfen sie nicht als Stromlinien missverstanden werden, die an jedem Punkt tangential zu den Feldlinien verlaufen. Der Vergleich mit Zeitlinien trifft es eher. Wieder sind die Resultate stark vom Versuchsaufbau abhängig.

In Abbildung 6.6 sind die Resultate eines Versuchs zu sehen, bei dem ein langer Saatschlauch in der Nähe des Tornadotrichters platziert wurde. Wegen der großen Menge der Objekte ist es schwer, die Form und die Ausmaße einzelner Schläuche zu erkennen. Selektiert man aber eines der Objekte, dann wird dieses blau eingefärbt und seine Verformungen und Bewegungen können leicht verfolgt werden. In der transparenten Darstellung ist der 3D-Eindruck nicht so gut, er kann aber verbessert werden, indem die Sicht auf die Visualisierung während der Simulation geändert wird.

Positioniert man eine große Zahl dünner Schläuche nebeneinander, dann approximieren diese beim Start eine Fläche. Das Resultat nach einigen Zeitschritten ist in Abbildungen 6.7 zu sehen. Da bei der gerenderten Darstellung der Schläuche Sampling-Artefakte auftreten und die Flächenstruktur in einem divergenten Feld relativ schnell verloren geht, wäre die Verwendung eines flächenförmigen Versuchsobjekts bei solchen Visualisierungen sinnvoller.

In Abbildung 6.9 ist die zeitliche Entwicklung eines Systems mehrerer Schlauchobjekte dargestellt, die an einem Ende fixiert sind. Dabei geraten die mittleren Schläuche in den Sog des Tornadotrichters.

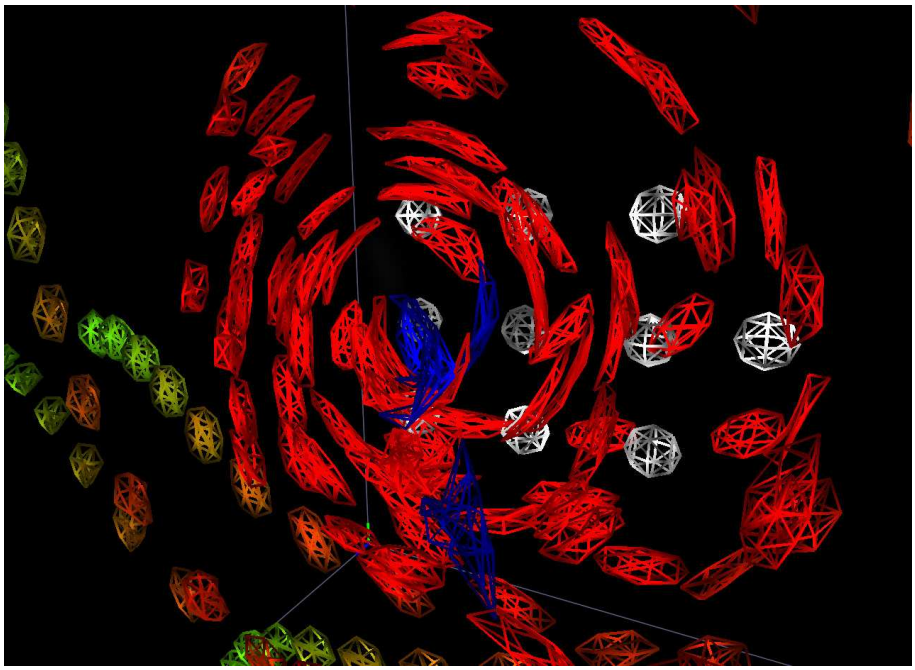


Abbildung 6.4: Die Deformation der Kugeln im stark divergenten Innenbereich des Tornadofeldes ist zu erkennen. Die blauen Objekte wurden vom Strudel eingesogen, ihre geometrische Kugelform ging dabei verloren. Die Objekte, die es aus dem Strudel herausschaffen, formen sich in ihre ursprüngliche Kugelform zurück.

Bestehende Probleme und Lösungsvorschläge

Wird die Federhärte zu klein gewählt, dann wird die Objektgeometrie in den stark divergenten Bereichen des Tornadofeldes schnell zerstört. Wird sie zu groß gewählt, dann verformen sich die Objekte im Randbereich wenig. Das Problem dabei ist, dass die verwendeten Geometrieobjekte hauptsächlich durch die Feder-Verbindungen ihrer Oberflächen stabilisiert werden. Eine gleichförmige Tetraedrisierung des vollständigen Objektvolumens könnte hier bessere Resultate liefern. Streambubbles hingegen teilen sich in divergenten Strömungsgebieten. Ein solcher Ansatz ist prinzipiell auch für die Geometrieadvektion geeignet, er läßt sich aber aufgrund der Einschränkungen der GPU und der verwendeten Datenstrukturen nicht umsetzen. Bei einigen der vorgestellten Visualisierung werden durch die zerstörte Oberflächenstruktur die starken Verwirbelungen in der Strömung sogar besser sichtbar. Die Erweiterung des Programms um eine Komponente, die geometrische Hindernisse in Strömungsdaten berrücksichtigt, wäre interessant.

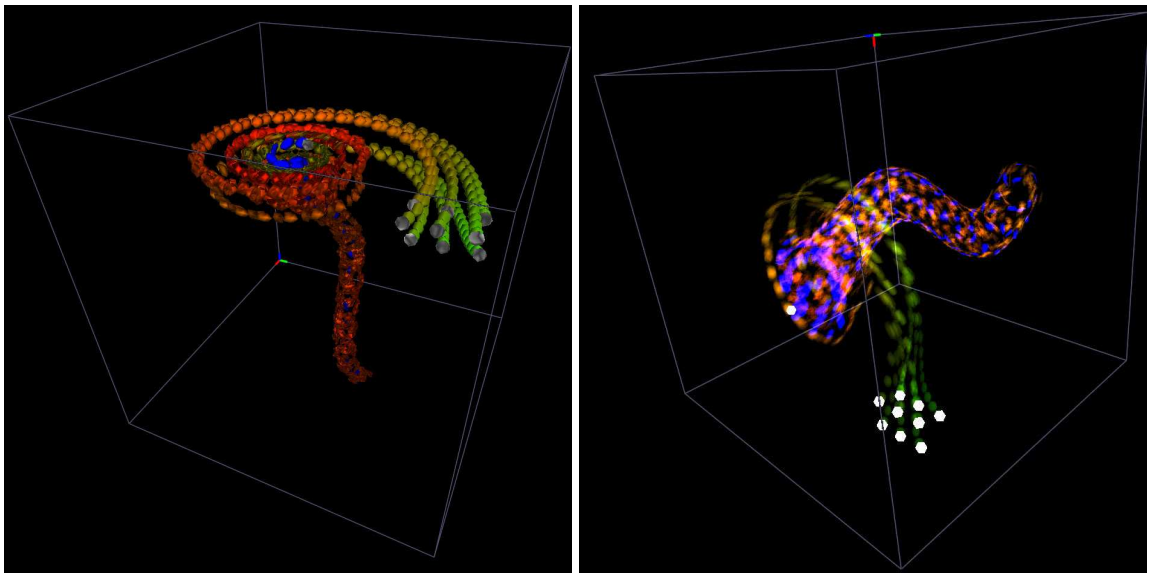


Abbildung 6.5: Bei der Wahl eines negativen Feldfaktors f_{fak} und der Positionierung der Saatpunkte am Rand des Gebietes, ergeben sich Bilder, die die Form des Tornados gut veranschaulichen. Im linken Bild ist im oberen inneren Bereich des Tornados ein schwächerer Strömungsbereich durch die grüne Färbung erkennbar.

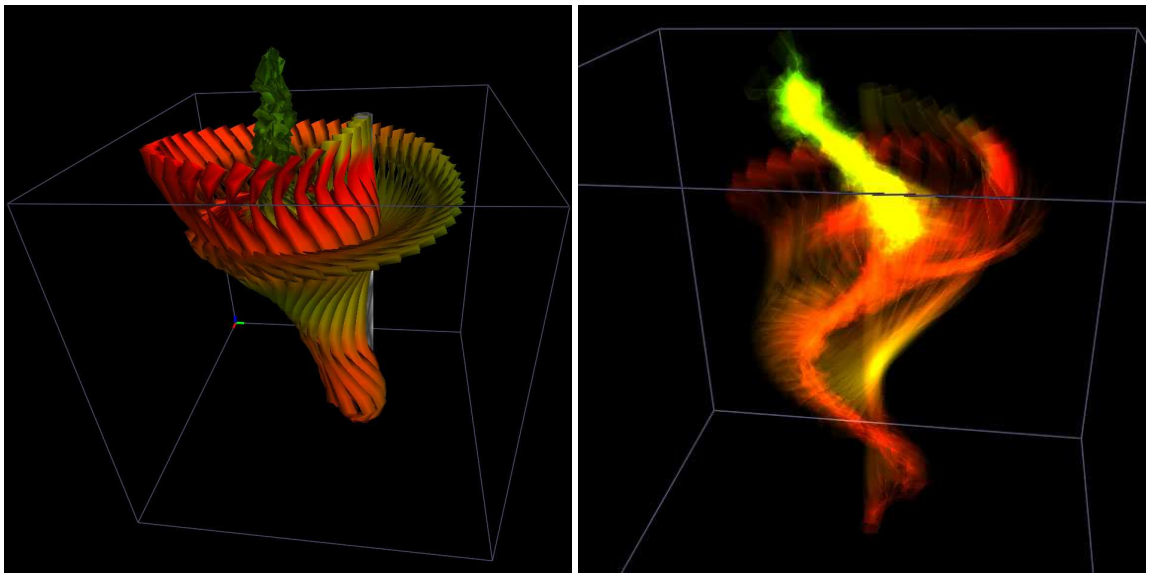
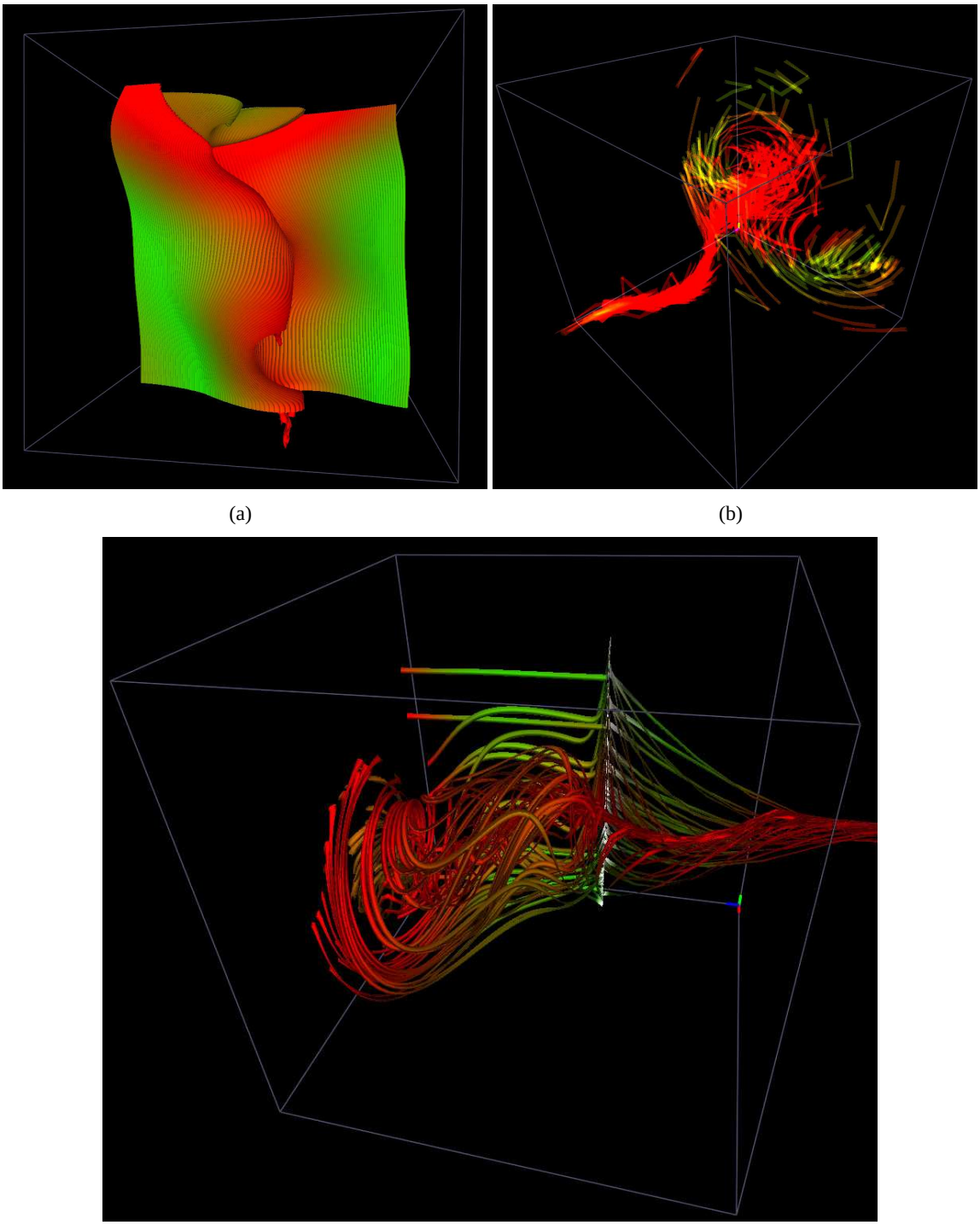


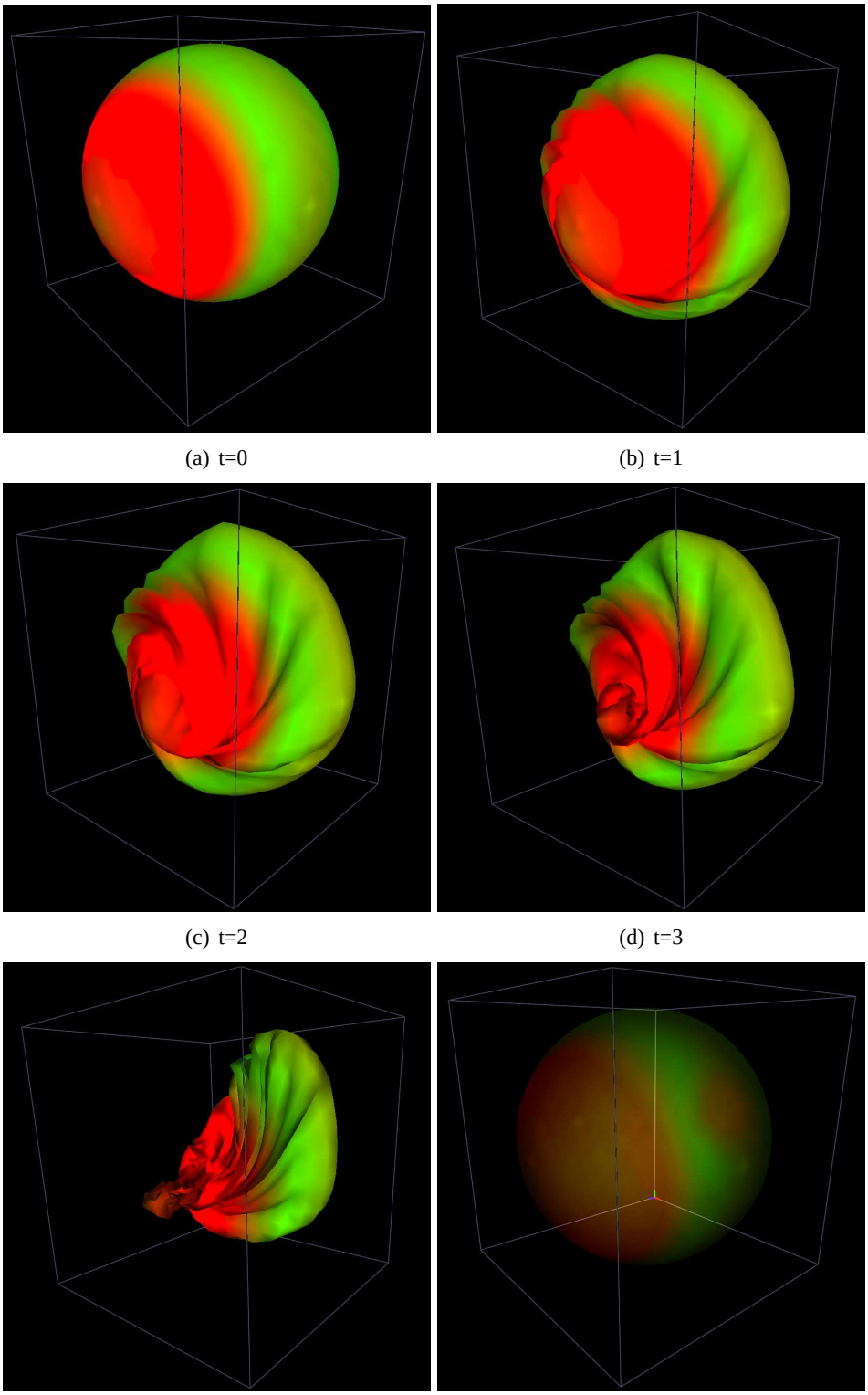
Abbildung 6.6: Platziert man einen Schlauch-Saatpunkt längs des Tornadotrichters, dann ergibt sich bei einer hohen Saatfrequenz eine flächenartige Struktur, die sich im Inneren des Wirbels auflöst. Die Schläuche werden dabei extrem verdrillt und von der Ranbehandlungsroutine zurückgesetzt.



(a) (b)

(c)

Abbildung 6.7: Je nach Positionierung der Schläuche ergeben sich ganz unterschiedliche Darstellungen: In (a) approximiert eine Vielzahl von Schläuchen eine Fläche, die sich aber wenige Zeitschritte später auflöst. In (b) sind einfache Schläuche aus drei Segmenten zu sehen und in (c) eine größere Menge langer Schläuche, die an einem Ende fixiert sind.

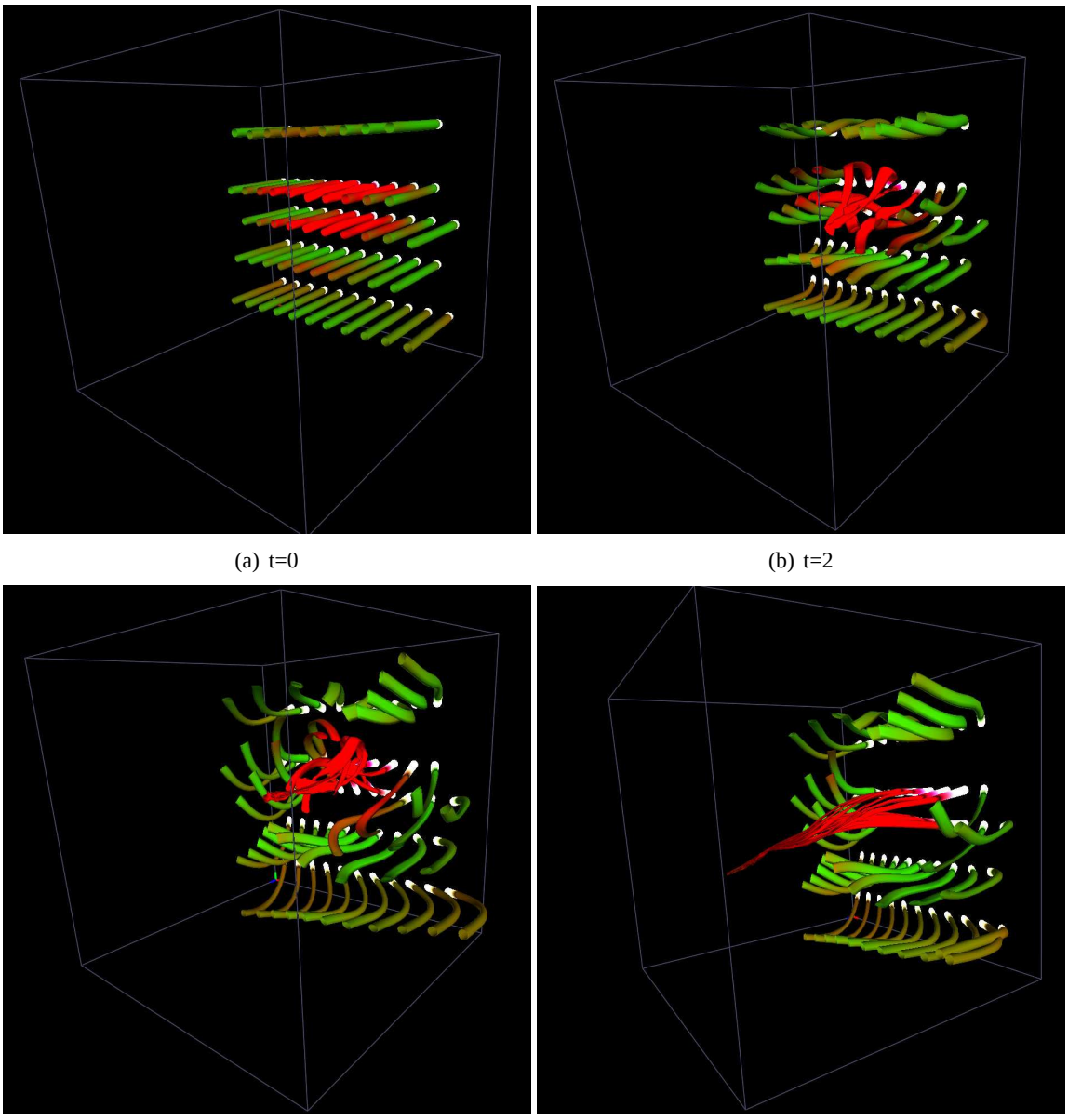


(a) t=0 (b) t=1

(c) t=2 (d) t=3

(e) t=4 (f) t=0

Abbildung 6.8: Zeitliche Abfolge der Verformung einer großen Kugel im Tornadowirbel (a)-(e). In (f) ist eine transparente Visualisierung der Kugel beim Startzeitpunk t_0 zu sehen.



(a) t=0 (b) t=2

(c) t=3 (d) t=4

Abbildung 6.9: Dargestellt ist eine größere Anzahl von Schläuchen, die an einem Ende fixiert sind. Über die Färbung läßt sich die Stärke des Feldes erkennen. Schläuche im Inneren Bereich werden vom Tornado langezogen.

7 Zusammenfassung

Es wurde eine neuartige Methode für die Visualisierung dreidimensionaler Strömungsfelder vorgestellt. Verschiedene Geometrieobjekte können vom Benutzer des implementierten Visualisierungstools interaktiv platziert und manipuliert werden. Dabei sind die Versuchsobjekte über eine physikalische Kopplung an das Feld gebunden und ihre Bewegungen und Verformungen in der Strömung werden animiert dargestellt, wobei sich die Verformungen in gleichförmigen Bereichen des Feldes zurückbilden können. Anhand der Objektdynamik können Informationen über das zugrundeliegende Strömungsfeld abgelesen werden. Dazu gehören die Richtung und die Stärke, sowie Divergenzen und Turbulenzen des Feldes. Ermöglicht wird das Verhalten bei Oberflächenverformungen durch das verwendete Objektmodell, das die Vertices der Geometrie als Massepunkte, und die Kanten, welche die Vertices verbinden, als Federn betrachtet. Die Speicherung des Zustand und die zeitliche Entwicklung des simulierten Objektsystems sowie seine graphische Darstellung, werden dabei vollständig von der GPU realisiert. Dies ermöglicht eine flüssige Simulation und Animation großer Objektsysteme, die aus vielen tausenden Einzelobjekten bestehen können.

Literaturverzeichnis

[1] Helman, J. L.; Hesselink, L.: *Visualizing Vector Field Topology in Fluid Flows*. IEEE Computer Graphics and Applications Vol. 11, 36–46 (1991)

[2] van Wijk, J. J.; Hin, A. J.; de Leeuw, W. C.; Post, F. H.: *Three ways to show 3d fluid flow*. IEEE Computer Graphics and Applications Vol. 14.5, 33–39 (1994)

[3] Lodha, S. K.; Renteria, J. C.; Roskin, K. M.: *Topology Preserving Compression of 2D Vector Fields*. In: Visualization 2000. Proceedings, Seiten 343–350. 2000

[4] Cabral, B.; Leedom, L. C.: *Imaging Vector Fields Using Line Integral Convolution*. In: Computer Graphics SIGGRAPH Proceedings, Seiten 263–272. August 1993

[5] van Wijk, J. J.: *Flow Visualization with Surface Particles*. IEEE Computer Graphics and Applications Vol. 13, 18–24 (1993)

[6] Zhang, B.; Pang, A.: *Stream Bubbles for Flow Visualization*. In: Computer Graphics and Applications, 2001. Proceedings. Ninth Pacific Conference, Seiten 169–177. 2001

[7] Hauser, H.; Laramee, R. S.; Doleisch, H.: *State-of-the-art report 2002 in flow visualization*, 2002

[8] Landau, L.; Lifschitz, E.: *Lehrbuch der Theoretischen Physik Band 1: Mechanik*. Verlag Harri Deutsch 1997

[9] Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; Vetterling, W. T.: *Numerical Recipes in C*. Cambridge University Press 1988

[10] Foley, J. D.; van Dam, A.; Feiner, S. K.; Hughes, J. F.: *Computer Graphics — Principles and Practice*. 2. Auflage Addision-Wesley 1990

[11] Schreiner, D.; Woo, M.; Neider, J.; Davis, T.: *OpenGL Programming Guide*. 5. Auflage Addison Wesley 2006

[12] Rost, R. J.; Kessenich, J. M.; Lichtenbelt, B.: *OpenGL Shading Language*. 2. Auflage Addison Wesley 2006

[13] WWW,: *Gpgpu, http://www.gpgpu.org*

[14] Fernado, R.: *GPU Gems*. Addison Wesley 2004

[15] Pharr, M.: *GPU Gems 2*. Addison Wesley 2005

[16] Liu, Y.; Liu, X.; Wu, E.: *Real-Time 3D Fluid Simulation on GPU with Complex Obstacles*. In: Proceedings of the Computer Graphics and Applications, Seiten 247–256. 2004

[17] Moreland, K.; Angel, E.: *The FFT on a GPU*. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Seiten 112–119. 2003

[18] Georgii, J.; Westermann, R.: *Mass-Spring Systems on the GPU*. Simulation Practice and Theory Vol. (2005)

[19] Georgii, J.; Echtler, F.; Westermann, R.: *Interactive Simulation of Deformable Bodies on GPUs*. In: Proceedings of Simulation and Visualization, Seiten 247–258. 2005

[20] Tejada, E.; Ertl, T.: *Large Steps in GPU-based Deformable Bodies Simulation*. Simulation Modelling Practice and Theory: Programmable Graphics Hardware Vol. 13, 703–715 (2005)

[21] Baraff, D.; Witkin, A.: *Large Steps in Cloth Simulation*. In: Computer Graphics Proceedings, Seiten 43–54. 1998

[22] Lewis, B.; Berg, D. J.: *PThreads Primer*. SunSoft Press 1996

Abbildungsverzeichnis

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfaßt und nur die angegebenen Quellen benutzt zu haben.

(Markus Üffinger)