

Inhaltsverzeichnis

1	Einleitung	3
2	Visualisierung im 2D	4
2.1	Grundlagen	4
2.2	Anforderungen und Probleme	5
2.3	Verfahren	6
2.3.1	Linienintegralfaltung (LIC)	7
2.3.2	Schnelle Linienintegralfaltung (Fast LIC)	10
3	Visualisierung im 3D	15
3.1	Grundlagen	15
3.2	Anforderungen und Probleme	15
3.3	Verfahren	15
3.3.1	Beleuchtete Feldlinien	16
3.3.2	Strömungsflächen	18
3.3.3	Raycasting von Vectorfeldern	20
3.3.4	3D LIC	22
3.3.5	Visualisierung Mittels einer Sonde	24
3.3.6	Texturesplats	26
3.3.7	Particle Splatting	27
4	Implementierung 3D LIC	34
4.1	Klassenhierarchie	34
4.2	Optimierungen	35
4.2.1	Beschleunigung	35
4.2.2	Verdeckungsproblem	36
4.3	GUI Einbindung	37
5	Implementierung Particle Splatting	38
5.1	Klassenhierarchie	38
5.2	Optimierungen	39
5.2.1	Beschleunigung	39
5.2.2	Verdeckungsproblem	41
5.2.3	Animation	41
5.3	GUI Einbindung	41
6	Ergebnisse	43
6.1	3D LIC	43
6.2	Particle Splatting	46
A	Methoden der 3D LIC Klassen	51
A.1	Vectorfield	51
A.2	Integrator	52
A.3	Texture	52
A.4	WeightFunction	53
A.5	SimpleLIC	53

B	Methoden der Particle Splatting Klassen	55
B.1	Vectorfield	55
B.2	Streamline	56
B.3	ParticleSpawner	56
B.4	Viewer	57

1 Einleitung

Vektorfelder kommen häufig aus dem Bereich der Computational Fluid Dynamics und werden mit der Zahl der Anwendungen und der steigenden Leistungsfähigkeit der Computer immer komplexer. Diese Diplomarbeit beschäftigt sich mit der Darstellung dreidimensionaler Vektorfelder und den damit entstehenden Problemen. Dabei wird im Besonderen die Visualisierung globaler Eigenschaften des Vektorfeldes betrachtet. In dieser Diplomarbeit werden zwei ausgewählte Verfahren betrachtet, die auf völlig verschiedenen Ansätzen beruhen.

Das erste Verfahren (siehe Abbildung 1) basiert auf der bisher am weitesten verbreiteten Visualisierung zweidimensionaler Vektorfelder, der Linienintegralfaltung (LIC) und liefert eine 3D-Textur, die zur Darstellung nur auf die Bildebene projiziert werden muß. Bei diesem Ansatz ist die Verdeckung eines der Hauptprobleme, das bereits in anderen Veröffentlichungen betrachtet wurde. Das zweite Verfahren (siehe Abbildung 2) basiert auf der Darstellung kleiner Partikel, die lokale Eigenschaften des Vektorfeldes visualisieren. Eine Visualisierung globaler Eigenschaften ist dabei nur durch eine genügend hohe Dichte dieser Partikel möglich.

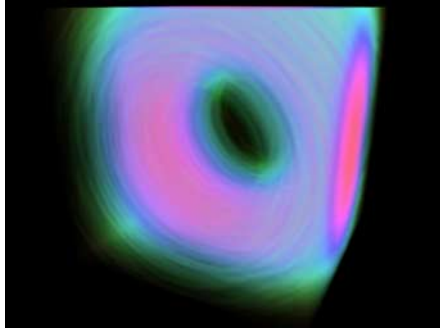


Abbildung 1: 3D LIC

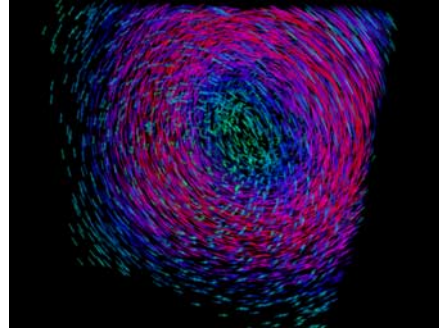


Abbildung 2: Particle Splatting

Obwohl die Ansätze grund verschieden sind, vermitteln sie beide einen Eindruck über die globalen Eigenschaften des Vektorfeldes. Die Verdeckung innerhalb des Vektorfeldes bleibt jedoch ein zentrales Problem, das von beiden Verfahren unterschiedlich gut gelöst wird.

2 Visualisierung im 2D

Die Visualisierung zweidimensionaler Vektorfelder wurde in letzter Zeit eingehend diskutiert. Die meisten erschienenen Veröffentlichungen basieren dabei auf dem Verfahren der Linienintegralfaltung (Line Integral Convolution, kurz LIC) von Cabral und Leedom [CL93], da es sowohl eines der elegantesten als auch eines der besten Verfahren darstellt.

2.1 Grundlagen

In einem zweidimensionalen Vektorfeld ist zu jedem Punkt ein Vektor, also eine Richtung definiert. Die meisten Datensätze verfügen neben dem Vektorfeld auch über einen oder mehrere skalare Werte, zu denen unter anderen auch die Länge der Vektoren zählt. Die einfachste Methode besteht darin, an ausgewählten Stellen des Vektorfeldes einen Pfeil in Richtung und Länge des darunterliegenden Vektors darzustellen. Diese Vektorpfeile (siehe Abbildung 3) haben jedoch einige entscheidende Nachteile.

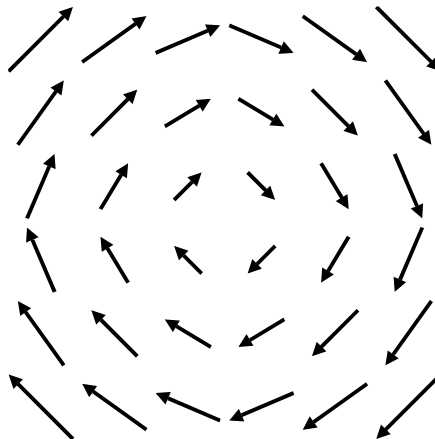


Abbildung 3: Visualisierung eines 2D Vektorfeldes mittels Vektorpfeilen.

Zwar kann man einfache Vektorfelder auf sehr intuitive Weise verstehen, aber mit zunehmender Komplexität muß auch die Zahl der Vektorpfeile steigen und das Verständnis kann leicht verloren gehen wenn, man nicht entsprechende Verbesserungen anwendet.

Die Datensätze verfügen neben den Vektor- und Skalarwerten noch über eine weitere Eigenschaft, die Topologie des Vektorfeldes. Diese Topologie besteht hauptsächlich aus kritischen Punkten oder Linien. Dabei wird ein Punkt kritisch genannt, wenn die Länge des Vektors an dieser Stelle 0 ist, d.h. keine Richtung definiert ist, aber in seiner unmittelbaren Nachbarschaft Vektoren der Länge größer 0 existieren. Das Vektorfeld aus Abbildung 3 hat also in der Mitte des Rotationszentrums einen kritischen Punkt. Ein Vektorfeld bei dem alle Vektoren die Länge 0 haben hat jedoch nach der vorherigen Definition keinen kritischen Punkt, da es in der Umgebung keines Punktes einen Vektor der Länge größer 0 gibt.

Das einfachste Verfahren zur Visualisierung solcher kritischen Punkte oder Linien ist die Darstellung sogenannter Feldlinien. Diese Feldlinien (siehe Abbildung 4) beschreiben dabei den Weg, den ein Teilchen im Vektorfeld zurücklegen würde. In der Nähe der kritischen Punkte nähern sich die Feldlinien einander immer mehr an und treffen sich dann theoretisch im kritischen Punkt, wobei dort keine Feldlinie definiert ist. Also ist ein kritischer Punkt bezogen auf die Feldlinien durch jeden beliebigen Punkt eine Definitionslücke. Ein kritischer Punkt kann allerdings auch ein Rotationszentrum wie in Abbildung 3 sein, das durch die Feldlinien ebenso gut zu erkennen ist.

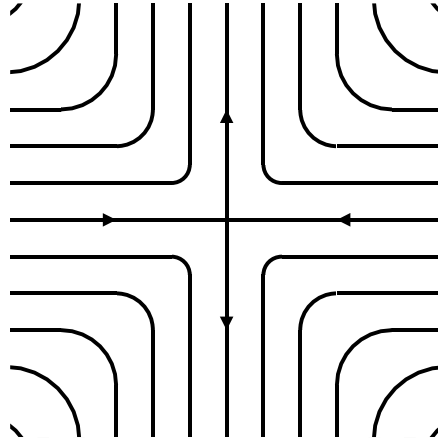


Abbildung 4: Visualisierung eines 2D Vektorfeldes mittels Feldlinien. In der Mitte befindet sich ein Sattelpunkt.

2.2 Anforderungen und Probleme

Eine gute Visualisierung sollte bestimmte Anforderungen erfüllen, die kurz nach der Einführung des LIC Verfahrens durch Cabral und Leedom [CL93] in einer Liste zusammengefaßt wurden.

- Hohe Datendichte
- Visuell intuitiv
- Dimensionale Generalität
- Gute Performance
- Geometrieunabhängig
- Eingabeunabhängig

Eine hohe Datendichte sorgt dafür, daß möglichst wenig Vektorinformation verloren geht. Dies ist besonders wichtig, wenn ein Vektorfeld nur kleine regional stark begrenzte kritische Punkte oder Linien besitzt. Bei einer zu niedrigen Datendichte könnten diese Regionen sonst verloren gehen.

Die erhaltenen Bilder müssen visuell intuitiv sein, das heißt man muß sowohl das Vektorfeld als auch alle kritischen Punkte oder Linien sofort erkennen und verstehen können. Eine Visualisierung, die zwar einen Überblick über das Vektorfeld und seine kritischen Bereiche liefert, ohne dabei den Vektor an jedem Punkt zu visualisieren, ist also vor allem auch im Zusammenhang mit der ersten Forderung nicht ausreichend. Die ersten beiden Anforderungen stellen aber schon jede Visualisierung eines dreidimensionalen Vektorfeldes vor ein großes Problem. Wie soll eine dichte Repräsentation, die letztendlich auf ein oder mehrere zweidimensionale Bilder reduziert wird, übersichtlich und damit visuell intuitiv bleiben?

Eine Visualisierung soll über dimensionale Generalität verfügen, also sich leicht vom Zweidimensionalen ins Dreidimensionale übertragen lassen, damit man nicht für verschiedene Dimensionen grundsätzlich verschiedene Verfahren benutzen muß. Jedoch stellt diese Anforderung eines der größten Probleme dieser Anforderungsliste dar, wie wir im weiteren Verlauf dieser Diplomarbeit sehen werden. Umgekehrt bedeutet es aber auch, daß Visualisierungen für dreidimensionale Vektorfelder auf zweidimensionale reduzierbar sein sollen. Diese Anforderung ist jedoch meist sehr einfach zu erfüllen, da im letzten Schritt immer zweidimensionale Bilder erzeugt werden.

Wie jeder Algorithmus sollte auch die Visualisierung eines Vektorfeldes mit möglichst guter Performance entwickelt werden oder, wenn die Berechnung für jedes Vektorfeld nur einmal ausgeführt werden muß, wenigstens eine annehmbare Komplexität besitzen. Gerade bei den LIC Verfahren ist die Performance ein besonders wichtiger Punkt, da für jeden Bildpunkt ein Integral über eine Feldlinie des Vektorfeldes erfolgt und es sich dabei um sehr aufwendige Berechnungen handelt. Im Allgemeinen sollen so wenig wie möglich Berechnungen für jeden Bildpunkt und nach Möglichkeit keine doppelten Berechnungen durchgeführt werden. Die Berechnungen auf einen kleinen Bereich um den Bildpunkt zu beschränken, also lokal zu halten, ist die einfachste Lösung um eine gute Performance zu erhalten.

Das Verfahren soll geometrieunabhängig sein, also mit allen möglichen Repräsentationen eines Vektorfeldes arbeiten können, die zu jedem Punkt einen Vektor und, falls vorhanden, die zusätzlichen skalaren Daten liefern können. Dabei dürfen keine Entscheidungen vom Benutzer oder von einem Algorithmus getroffen werden, die zum Verlust von Daten führen könnten.

Die letzte wichtige Anforderung besagt, daß die Visualisierung eingabeunabhängig sein muß, also keine Annahmen über das Vektorfeld machen darf. Ein Verfahren, das nur in einem bestimmten Bereich von Vektorlängen arbeitet oder nur mit Vektorfeldern bei denen keine Vektoren am Rand nach aussen oder innen zeigen, ist zum Beispiel nicht geeignet. Also sind keine Verfahren für nur eine bestimmte Menge an Vektorfeldern erwünscht. Eine Visualisierung nur einer speziellen Art von Vektorfeldern reicht jedoch manchmal schon aus, so daß diese Anforderung nur für Verfahren gedacht ist, die alle möglichen Vektorfelder visualisieren will.

2.3 Verfahren

Wie bereits erwähnt findet bei zweidimensionaler Vektorfelder fast nur noch die Visualisierung mit Linienintegralen Anwendung. Bei diesem Verfahren wird ein Eingabebild, meist ein einfaches Rauschen, entlang der Feldlinien integriert, so-

daß der Kontrast entlang dieser Linien sehr gering, senkrecht dazu jedoch sehr groß ist.

2.3.1 Linienintegralfaltung (LIC)

Das Linienintegral für jeden Punkt p im Vektorfeld $V(p)$ wird nach Cabral und Leedom [CL93] folgendermaßen berechnet:

Eine parametrisierte Kurve $P(p, s)$ mit $P'(p, s) = V(P(p, s))$, also ein Stück einer Feldlinie, wird erzeugt, die dem Vektorfeld ein Stück der Länge L vorwärts und rückwärts von p aus folgt. Die Funktion $F(p)$ liefert zu jedem Punkt p den entsprechenden Wert aus dem Eingabebild. Also liefert $F(P(p, s))$ den entsprechenden Wert für einen Punkt auf der parametrisierten Kurve. Diese Funktion wird nun mit dem symmetrischen Filterkern $k(s)$ gefaltet und mit der Fläche unter dem Filter normiert, um unabhängig vom Filter immer im gleichen Helligkeitsbereich zu bleiben. Für die normierte Funktion $\hat{F}(p)$ ergibt sich damit:

$$\hat{F}(p) = \frac{\int_{-L}^L F(P(p, s))k(s) ds}{\int_{-L}^L k(s) ds} \quad (1)$$

Abhängig von der Form des Filterkerns, dem Eingabebild, einigen Kontrollparametern und dem Vektorfeld selbst kann eine große Anzahl verschiedener Bilder erzeugt werden (siehe [CL93]).

Um jedoch einen Algorithmus zur Bilderzeugung anzugeben, müssen alle benötigten Formeln ins Diskrete übertragen werden. Dazu muß die parametrisierte Kurve in einzelne Punkte P_i umgewandelt werden und der Filterkern muß für diese diskreten Punkte berechnet werden. Für die Formel 1 ergibt sich dann mit dem diskretisierten Filterkern h_i :

$$\hat{F}(p) = \frac{\sum_{i=-l}^l F(P_i)h_i}{\sum_{i=-l}^l h_i} \quad (2)$$

Um die Funktion $\hat{F}(p)$ nach Formel 2 zu berechnen müssen folgende Probleme angesprochen werden. Wie berechnet man die Feldlinie oder die Punkte P_i der Feldlinie zu p ? Wie berechnet man die Faltung mittels der Gewichte h_i und in welcher Reihenfolge werden diese beiden Schritte berechnet?

Um nun die Feldlinien numerisch zu berechnen, bietet sich ein Euler Verfahren mit variabler Schrittweite an. Um die folgenden Formeln zu vereinfachen, wird das Vektorfeld auf die Größe des Ausgabebildes skaliert und mit den Pixelkoordinaten adressiert. Dabei wird jeder Teil des Vektorfeldes unter einem Pixel als Zelle mit konstantem Vektor betrachtet. Die Koordinate P_0 des Punktes p an der Stelle $(x; y)$ ist dann:

$$P_0 = (x + 0,5; y + 0,5) \quad (3)$$

Und für die dazugehörigen Vektoren:

$$v_i = V(P_i) \quad (4)$$

Bei dieser Methode wird die Feldlinie von der aktuellen Position aus bis zum Rand der Zelle, vorwärts oder rückwärts entlang des Vektors dieser Zelle, weiter verfolgt (siehe Abbildung 5). Bei dieser Methode wird die Feldlinie von der Position P_i bis zum Rand der Zelle in der aktuellen Richtung (Vorwärts oder Rückwärts) verfolgt (siehe Abbildung 5). So wird durch den Punkt P_i , den dazugehörigen Vektor v_i und den Abstand zur naheliegendsten Kante in Vektorrichtung Δs_i der Punkt P_{i+1} folgendermaßen definiert:

$$P_{i+1} = P_i + \frac{v_i}{\|v_i\|} \Delta s_i \quad (5)$$

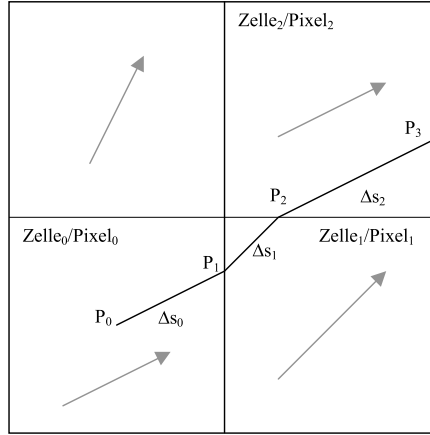


Abbildung 5: Stückweise lineare Annäherung einer Feldlinie vom Punkt P_0 aus.

Δs_i ist dabei der minimale Abstand zum Rand der aktuellen Zelle in Richtung des Vektors, also das positive Minimum der Abstände zu allen vier Rändern $\Delta s_{oben, unten, rechts, links}$. Durch die Richtung des Vektors v_i und dem Punkt P_i können jedoch alle bis auf eine der folgenden Formeln ausgeschlossen werden. Für die Abstände im einzelnen gilt:

$$\begin{aligned} \Delta s_{oben} &= ((y+1) - P_{i,y}) \frac{\|v_i\|}{v_{i,y}} \\ \Delta s_{unten} &= (y - P_{i,y}) \frac{\|v_i\|}{v_{i,y}} \\ \Delta s_{rechts} &= ((x+1) - P_{i,x}) \frac{\|v_i\|}{v_{i,x}} \\ \Delta s_{links} &= (x - P_{i,x}) \frac{\|v_i\|}{v_{i,x}} \end{aligned} \quad (6)$$

Bei einem Nullvektor bleibt die Feldlinie in der entsprechenden Zelle des Vektorfeldes stehen, so daß $P_i = P_{i-1}$ gilt.

Da $\|v_i\|$ sowohl im Zähler von Δs_i als auch im Nenner von Gleichung 5 steht, kann man damit kürzen, falls man nur die neue Position P_i benötigt. Jedoch muß für den Faltungskern $k(s)$ jedes Δs_i in normierter Form vorliegen, da durch Δs_i das jeweilige Stück auf s repräsentiert werden muß.

In der selben Weise wie die Feldlinie in positiver Richtung berechnet wurde, wird sie vom selben Startpunkt aus auch in negativer Richtung also rückwärts berechnet. Die so erhaltenen Punkte werden mit negativen Indizes, also mit P_{-i} bezeichnet.

Bis jetzt haben wir eine Feldlinie stückweise durch Linien angenähert, was jedoch nicht unbedingt nötig ist. Ein anderer Algorithmus bräuchte nur die Punkte P_i zu berechnen, ohne für jeden Wert auf der Feldlinie einen Punkt zu liefern.

Als nächstes muß nun das Integral des Filterkerns über dieser stückweise linear angenäherten Feldlinie berechnet werden. Das Integral kann dabei stückweise über C^1 stetige Funktionen angenähert werden, wenn das Eingabebild als kontinuierliche Funktion von x und y betrachtet wird. h_i wird für jedes Liniensegment i berechnet durch Integration des Filterkerns $k(s)$ von s_i bis s_{i+1} , den parametrischen Koordinaten der Endpunkte von P_i :

$$h_i = \int_{s_i}^{s_{i+1}} k(s) ds \quad (7)$$

Dabei gilt:

$$\begin{aligned} s_0 &= 0 \\ s_{i+1} &= s_i + \Delta s_i \end{aligned} \quad (8)$$

Mit $s_l \leq L < s_{l+1}$ beziehungsweise $s_{l'} \leq L < s_{l'+1}$ erhält man dann für die gesamte LIC Approximation:

$$\hat{F}(p) = \frac{\sum_{i=-l}^l F(P_i) h_i}{\sum_{i=-l}^l h_i} \quad (9)$$

Um die Werte h_i berechnen zu können, wird eine Tabellen k mit 4095 Einträgen angelegt:

$$k_i = \int_0^{i/2047 * L} k(s) ds \quad i \in [-2047..2047] \quad (10)$$

Daraus folgt dann für h_i :

$$h_i \approx k_{\lfloor s_{i+1}/L * 2047 \rfloor} - k_{\lfloor s_i/L * 2047 \rfloor} \quad (11)$$

Die Länge der Feldlinie ist durch die Verfolgung des Vektorfeldes in beide Richtungen doppelt so groß, wie der Wert L . Wird diese Länge L zu groß gewählt, liegen die Farbwerte aller Pixel zu eng bei einander und das gesamte Bild verschwimmt. Wird L jedoch zu klein gewählt, wird zu wenig gefiltert und das Vektorfeld wird nicht ausreichend visualisiert. Da die Rechenzeit stark von L

abhängt wird der kleinste effektive Wert benötigt. Ein gebräuchlicher Wert ist dabei $L = 10$, so daß eine Feldlinienlänge von 20 Pixeln erzeugt wird.

Nach der Übertragung der Formel 1 ins Diskrete und mit der Näherung 11 für den Filterkern, beginnt nun der letzte Schritt dieses Verfahrens, die Bilderzeugung. Nach der Filterung ist zu beachten, daß die Kontraste des gefilterten Bildes von der Filterlänge L abhängen. Um so größer L wird, um so kleiner werden die Kontraste im erzeugten Bild und sie müssen daher entweder während der Filterung korrigiert werden, oder im Nachhinein auf den vollen Bereich skaliert werden. Die Skalierung während der Filterung ist nur möglich, wenn man die genauen Eigenschaften des Filterkerns und des Eingabebildes kennt.

Der letzte Punkt zur Bilderzeugung ist nun noch die Wahl eines Filterkerns zur Faltung. Dabei kann die Wahl für einen bestimmten Filter aus sehr unterschiedlichen Motiven heraus fallen und hat auch direkten Einfluß auf die Filterlänge L und die Qualität des erzeugten Bildes, wobei dies meistens gegenläufig ist.

Der einfachste Filter ist der sogenannte Box-Filter für den $k(s) = 1$ gilt. Dieser Filter benötigt zwar eine sehr kleine Filterlänge L , aber im erzeugten Bild sind starke Aliaseffekte entlang der Feldlinien erkennbar. Ein Filter, der sowohl eine kleine Filterlänge benötigt als auch nur zu geringen Aliaseffekten entlang der Feldlinien führt, ist der Hanning-Filter mit $k(s) = (\cos(s * \pi/L) + 1)/2$.

Die erzeugten Bilder verfügen jedoch meist über sehr starke Aliaseffekte orthogonal zu den Feldlinien, da über ein Eingabebild mit sehr hohen Frequenzen nur in einer Richtung gefiltert wurde. Diese Effekte lassen sich wie alle Aliaseffekte auf zwei Arten mindern. Entweder man filtert das Eingabebild um die hohen Frequenzen zu unterdrücken oder das Ausgabebild wird höher abgetastet.

Cabral und Leedom haben jedoch noch ein weiteres Verfahren gefunden, um Aliaseffekte zu unterdrücken, das sehr gut funktioniert jedoch nur schwer zu analysieren ist. Bei diesem Verfahren wird das erzeugte Bild ein zweites Mal mit dem selben Vektorfeld gefiltert. Man könnte annehmen, daß dadurch das selbe Bild erzeugt wird als wenn man auf den Feldlinien zweimal gefiltert hätte. Wenn man das Verfahren jedoch genauer betrachtet erkennt man, daß viel kompliziertere Dinge passieren, da im zweiten Durchlauf auch Pixelwerte verwendet werden, die durch benachbarte Feldlinien entstanden sind.

2.3.2 Schnelle Linienintegralfaltung (Fast LIC)

Wie schon im vorherigen Abschnitt erwähnt, steigt die benötigte Zeit zur Visualisierung des Vectorfeldes sowohl linear mit der Länge des verwendeten Filters als auch mit der Anzahl der zu berechnenden Pixel. Die beiden zeitaufwendigsten Abschnitte der Visualisierung mit Linienintegralen sind die Berechnung der Feldlinien und der Faltungsintegrale.

Hege und Stalling präsentieren in [HS98] ein schnelles Verfahren, das die Abhängigkeit von der Filterlänge für stückweise polynomielle Filter durch die Ausnutzung eines der Faltungstheoreme stark reduziert und auf dem Verfahren von Battke etc. [SH95] beruht.

Um die Berechnung der Feldlinien zu reduzieren, werden sie mit einer Länge L' berechnet, die während der Berechnung an das aktuelle Vektorfeld und an die Anzahl der bereits berechneten Pixel angepaßt wird. Die Faltung entlang dieser Linie der Länge L' erfolgt nun unter Ausnutzung des Faltungstheorems:

$$\begin{aligned}
(f * h)(x) &= \int_{-\infty}^{\infty} f(y)h(x-y)dy & (12) \\
&= \int_{-\infty}^{\infty} F(y)h'(x-y)dy \\
&= (F * h')(x)
\end{aligned}$$

Um die weitere Schreibweise zu vereinfachen definieren wir unter der Annahme, daß f mindestens n -mal integrierbar und h mindestens n -mal differenzierbar ist:

$$F_n(x) = \int_{-\infty}^x \int_{-\infty}^{x_1} \dots \int_{-\infty}^{x_{n-1}} f(x_1) dx_1 \dots dx_{n-1} dx_n \quad (13)$$

Daraus ergibt sich folgende Formel, die auch dann noch gültig ist, wenn die n -te Ableitung $h^{(n)}$ eine Delta Distribution ist, also nur an einer endlichen Zahl von x -Werten ungleich 0 ist.

$$f * h = F_n * h^{(n)} \quad (14)$$

Stückweise polynomielle Funktionen können in einer Basis angegeben werden, in der jedes Basiselement durch (mehrfaches) Ableiten zu einer Delta Distribution wird. Für eine streng monoton steigende Folge von Knoten $\xi := (\xi_i)_{i=1 \dots l}$ mit $\xi_i \in \mathbb{R}$ und einer Menge von Polynomen $S_i, i = 1 \dots l$ jeweils höchstens der Ordnung k definieren wir eine stückweise polynomielle Funktion der Ordnung k mit:

$$f(x) := \begin{cases} 0, & x < \xi_1 \\ S_i(x), & \xi_i \leq x < \xi_{i+1}, \quad i = 1 \dots l-1 \\ S_l(x), & x \geq \xi_l \end{cases} \quad (15)$$

Zur Verwendung dieser Funktion als Filter muß $S_l = 0$ gelten, da der Filter über einen kompakten Träger verfügen muß. Wie man leicht erkennen kann, bilden die Polynome k -ter Ordnung mit einer festen Knotenfolge ξ einen linearen Raum $\mathbb{P}_{k,\xi}$. Wir definieren für diesen Raum eine Basis mit Hilfe der abgeschnittenen Potenzfunktionen $(x)_+^r$ für $r \in \mathbb{R}_0$.

$$(x)_+^r := \begin{cases} 0, & x < 0 \\ x^r, & x \geq 0 \end{cases} \quad (16)$$

Wir definieren nun für jeden Knoten i eine Folge von Basisfunktionen mit:

$$\phi_{ij}(x) = \frac{(x - \xi_i)_+^j}{j!} \quad \text{für } i = 1 \dots l \text{ und } j = 0 \dots k-1 \quad (17)$$

Die ϕ_{ij} sind stückweise polynomielle Funktionen der Ordnung $j+1$ mit nur einem Knoten ξ_i . Da wir $S_l = 0$ nicht verlangt haben, sind sie Elemente von $\mathbb{P}_{k,\xi}$. Außerdem gilt

$$\frac{d}{dx} \phi_{ij} = \phi_{i,j-1} \quad \text{für } j = 1 \dots k-1 \quad (18)$$

und

$$\frac{d^j}{dx^j} \phi_{ij} = \phi_{i,0} \quad \text{für } j = 0 \dots k-1 \quad (19)$$

wobei $\phi_{i,0}$ eine Funktion ist, die bei ξ_i von 0 auf 1 springt. Obwohl B-Splines für die meisten numerischen Verfahren wesentlich besser geeignet sind, kann man mit dieser Basis das Faltungsintegral auf eine sehr elegante Weise umschreiben.

Wenn der Filterkern als stückweise polynomielle Funktion mit $h = \sum_{ij} c_{ij} \phi_{ij}$ gegeben ist, ergibt sich mit der Formel 18 und dem Faltungstheorem (Formel 14) folgende Darstellung des Faltungsintegrals:

$$\begin{aligned} (f * h)(x) &= \int_{-\infty}^{\infty} f(y)h(x-y)dy & (20) \\ &= \sum_{ij} c_{ij} \int_{-\infty}^{\infty} f(y)\phi_{ij}(x-y)dy \\ &= \sum_{ij} c_{ij} \int_{-\infty}^{\infty} F_{j+1}(y)\delta(x-\xi_i-y)dy \\ &= \sum_{ij} c_{ij} F_{j+1}(x-\xi_i) \end{aligned}$$

Die Berechnung des Faltungsintegrals reduziert sich also auf die Berechnung einer gewichteten Summe von wiederholten Integralen oder, im diskreten Fall, von wiederholten Summen. Dabei sollte die Darstellung in unserer Basis möglichst einfach sein. Diese Bedingung erfüllen die symmetrischen B-Splines k -ter Ordnung mit $k+1$ uniform verteilten Stützstellen (siehe Tabelle 1).

Für die Faltung mit einem B-Spline wird nur ein einziges F_n benötigt. Im Allgemeinen können auch mehrere F_n benötigt werden, wodurch die Berechnung etwas aufwendiger wird.

Die Diskretisierung des Integrals wird ähnlich wie beim normalen LIC Verfahren berechnet, so daß sich mit

$$F(k\Delta s) = \int_0^{k\Delta s} f(x)dx \approx \Delta s \sum_{i=0}^{k-1} f(i\Delta s) \quad (21)$$

folgende Formel für die höheren Integrale ergibt:

$$F_n(k\Delta s) = \int_0^{k\Delta s} F_{n-1}(x)dx \approx \Delta s \sum_{i=0}^{k-1} F_{n-1}(i\Delta s) \quad (22)$$

Im einzelnen benötigt der Algorithmus zur Berechnung der Linienintegralfaltung nun folgende Schritte. Darstellung des Filterkerns mit Hilfe der Basisfunktionen $h = \sum_{ij} c_{ij} F_j(k\Delta s - \xi_i)$. Berechnung der wiederholten Summen an der Stelle $k\Delta s$ bis zur Ordnung n . Diskrete Approximation des Faltungsintegrals $\tilde{I}(k\Delta s)$ mit:

$$\tilde{I}(k\Delta s) = \sum_{ij} c_{ij} F_j(k\Delta s - \xi_i) \quad (23)$$

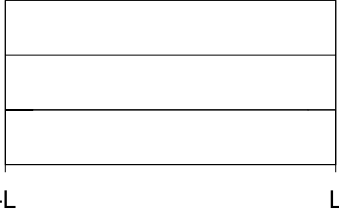
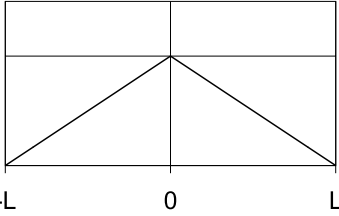
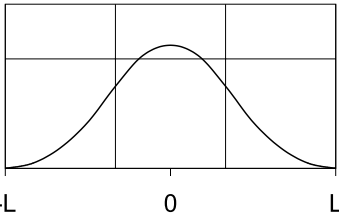
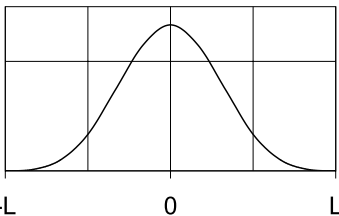
Filterkern h	Definition	Faltung $f * h$
	$S_1 = \frac{1}{2L}$	$\frac{1}{2L}(F(x+L) - F(x-L))$
	$S_1 = \frac{1}{L^2}(x+L)$ $S_2 = \frac{1}{L^2}(L-x)$	$\frac{1}{L^2}(F_2(x+L) - 2F_2(x) + F_2(x-L))$
	$S_1 = \frac{27}{16L^3}(x+L)^2$ $S_2 = \frac{9}{8L} - \frac{17}{8L^3}x^2$ $S_3 = \frac{27}{16L^3}(L-x)^2$	$\frac{27}{8L^3}(F_3(x+L) - 3F_3(x + \frac{L}{3}) + 3F_3(x - \frac{L}{3}) - F_3(x-L))$
	<p>Faltung mit einem B-Spline Filterkern n</p> $(f * h)(x) = \left(\frac{n}{2L}\right)^n \sum_{i=0}^n (-1)^i F_n\left(x + \frac{n-2i}{n}L\right)$	

Tabelle 1: Filterkerne von B-Splines, die durch mehrfache Faltung mit dem Boxfilter entstehen.

Durch die bereits angesprochene Wiederverwendung einer Feldlinie für mehrere Pixel des Ausgabebildes wird die Anzahl der Berechnungen für jeden weiteren Punkt der Feldlinie, zum Beispiel für den Box Filter, auf eine Addition und eine Subtraktion reduziert.

Als letzter Punkt bleibt dann noch die Wahl der Startpunkte für eine neue Feldlinie und ihrer Länge. Für die Wahl der Startpunkte könnte man einfach nach dem Scanline Verfahren vom nächsten unbesetzten Pixel aus eine Feldlinie

in beide Richtungen berechnen, was jedoch häufig zu einer sehr großen Anzahl an mehrfach berechneten Punkten führt. Als bestes Verfahren hat sich eine quasi Zufallsfolge, um genauer zu sein, eine zufällige Permutation aller Pixel, die durch den Sobol Algorithmus ([Sob67]) mit der Implementierung von Bratley und Fox ([BF88]) erzeugt wird, mit der adaptiven Anpassung der Feldlinienlänge herausgestellt. Die Länge wird dabei so angepaßt, daß auf Grund der bisherigen Ergebnisse des Algorithmus die Kosten, das heißt die Zeit, die ein Pixel zur Berechnung benötigt hat, minimiert werden. Die Kosten einer Feldlinie, mit der eine Linie der Länge L_s im Ausgabebild mit der Filterlänge L_f erzeugt wird, ist dabei:

$$K_{feldlinie}(L_s) = \alpha(L_f + L_s) \quad (24)$$

Die eher geringen Kosten der Initialisierung werden ignoriert. $P(l)$ die Anzahl der noch nicht besetzten Punkte einer Feldlinie der Länge l . $P(l)$ steigt zwar monoton mit l , ist aber nicht dazu proportional. Für die Kosten eines neuen Punktes folgt damit:

$$K_{pixel}(L_s) = \frac{\text{Kosten der Feldlinie}}{\text{Anzahl neuer Punkte}} = \frac{\alpha(L_f + L_s)}{P(L_s)} \quad (25)$$

Die Anzahl der neuen Pixel wird aus den bisherigen Werten abgeschätzt und damit auch das Minimum dieser Kostenfunktion. Auf diese Weise werden die Gesamtkosten zur Berechnung des Ausgabebildes reduziert, obwohl die Anzahl der benötigten Feldlinien steigt.

3 Visualisierung im 3D

Auch die Visualisierung dreidimensionaler Vektorfelder wurde in letzter Zeit stark diskutiert, jedoch wurde, im Gegensatz zum Zweidimensionalen, bisher keine optimale Lösung gefunden. Obwohl sich die meisten Verfahren von zwei- auf dreidimensionale Vektorfelder erweitern lassen, stößt man dabei häufig auf Probleme, die im Zweidimensionalen nicht auftreten können.

3.1 Grundlagen

Wie auch im zweidimensionalen Fall ist zu jedem Punkt im Raum ein Vektor und somit auch eine Topologie definiert. Doch gerade diese Topologie zum Beispiel ist sehr viel komplexer als im Zweidimensionalen. Es gibt nicht mehr nur kritische Punkte und Linien sondern auch Oberflächen und durch die dritte Dimension auch wesentlich mehr Freiheitsgrade für solche kritische Bereiche. Ein weiterer Punkt bei dreidimensionalen Vektorfeldern ist, daß diese häufig auf einem geometrischen Objekt gegeben sind und interpoliert werden müssen, also es im Gegensatz zu zweidimensionalen Vektorfeldern, die meist auf einem regelmäßigen Gitter gegeben sind, auch nicht definierte Punkte existieren. Diese Definitionslücken können aber für die meisten Verfahren durch Einfügen von Nullvektoren behoben werden.

3.2 Anforderungen und Probleme

Die Anforderungen an eine Visualisierung dreidimensionaler Vektorfelder sind denen zweidimensionaler Vektorfelder sehr ähnlich. Im Unterschied zum zweidimensionalen Fall spielt jedoch auch die Verdeckung eine wichtige Rolle. Gerade bei Verfahren mit hoher Datendichte wird die Verdeckung von Informationen im Inneren des Vektorfeldes zum zentralen Problem, da beide Anforderungen gegenläufig sind. Auf der anderen Seite ist ein Verfahren mit hoher Datendichte und wenig Verdeckung, also eine sehr transparente Repräsentation mit sehr hoher Transparenz nicht mehr visuell intuitiv, da zu viele Daten auf einem Punkt visualisiert werden und das entstandene Bild dann unübersichtlich wird.

Alle diese Probleme entstehen, da der Mensch mit seinen zwei Augen nur in der Lage ist pseudo-dreidimensionale Bilder zu sehen. Man kann zum Beispiel nicht hinter ein Objekt sehen, da man nur ein zweidimensionales Bild mit zusätzlichen Tiefeninformationen sieht, also keine echte dreidimensionale Szene. Genauer gesagt, bleibt vor allem das Problem der Verdeckung selbst dann bestehen, wenn man über einen Bildschirm verfügt, der ein dreidimensionales Bild anzeigen kann. Durch eine Animation kann dieses Problem jedoch meistens stark reduziert werden.

3.3 Verfahren

Die hier vorgestellten Verfahren können nicht alle Anforderungen gleichzeitig erfüllen, da einige Anforderung, die ursprünglich für zweidimensionale Vektorfelder aufgestellt wurden, im Dreidimensionalen gegenläufig sind. Die Verfahren versuchen dabei auf sehr unterschiedliche Weise einen möglichst ausgeglichenen Mittelweg zu finden, ohne eine Anforderung völlig zu vernachlässigen oder versuchen einige Probleme zum Beispiel mit Animation zu umgehen.

Von den hier präsentierten Verfahren wurde das 3D LIC Verfahren und das Particle Splatting implementiert. Da 3D LIC die besten Ergebnisse versprach, aber während der Implementation einige Probleme mit der Visualisierung auftraten, wurde das Particle Splatting entwickelt um durch einen völlig anderen Ansatz Einblick in ein dreidimensionales Vektorfeld zu erhalten.

3.3.1 Beleuchtete Feldlinien

Eine sehr einfache Methode um das Problem der Verdeckung zu reduzieren besteht darin, nur einzelne Feldlinien zu visualisieren, jedoch kann OpenGL oder auch die meisten anderen Grafikschnittstellen keine Linien beleuchten. Ohne Beleuchtung bleibt diese Art der Visualisierung im Dreidimensionalen jedoch sehr unübersichtlich und ist damit nicht visuell intuitiv.

Zöckler, Stalling und Hege zeigen mit ihrem Verfahren [ZSH96] jedoch eine sehr schnelle und effektive Art der Beleuchtung von Feldlinien. Als erstes bleibt jedoch zu klären, wie eine Linie überhaupt zu Beleuchten ist.

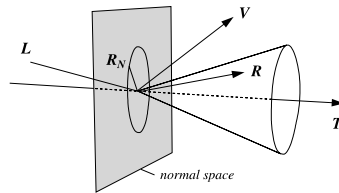


Abbildung 6: Ebene mit Normalen einer Feldlinie und möglichen reflektierten Strahlen R . [ZSH96]

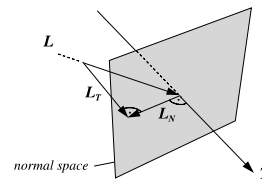


Abbildung 7: Zerlegung des Vektors \vec{L} in die zwei orthogonalen Komponenten \vec{L}_T und \vec{L}_N . [ZSH96]

Um einen möglichst guten Eindruck vom Verlauf der einzelnen Linien zu bekommen, verwendet man zur Beleuchtung das Modell von Phong [Pho75]. Jede Fläche kann dabei lokal durch ihre Normale \vec{N} charakterisiert werden. Wenn nun \vec{L} die Richtung der Lichtquelle, \vec{V} die Blickrichtung und \vec{R} der reflektierte Strahl ist, läßt sich die Intensität eines Punktes auf einer Fläche folgendermaßen angeben:

$$\begin{aligned} I &= I_{ambient} + I_{diffus} + I_{spekular} \\ &= k_a + k_d \vec{L} \cdot \vec{N} + k_s (\vec{V} \cdot \vec{R})^n \end{aligned} \quad (26)$$

Der Faktor n im spekularen Anteil gibt dabei die Breite des Highlights, also des reflektierenden Bereichs an. Bei Linien gibt es jedoch nun das Problem, daß kein genereller Normalen- und Reflektionsvektor angegeben werden kann, statt dessen können nur zweidimensionale mannigfaltige Flächen angegeben werden. Genauer gesagt eine Ebene für die Normale und ein Kegel für die Reflektion. Da Linien in \mathbb{R}^3 keinen Normalenvektor, sondern eine Normalenebene besitzen, kann das von Banks [Ban94] für Normalen höherer Dimension erweiterte Beleuchtungsmodell angewendet werden. Für Linien in \mathbb{R}^3 ist das Ergebniss recht

offensichtlich. Von allen möglichen Normalen kommt nur die in Frage, die mit \vec{L} und der Tangente \vec{T} in einer Ebene liegt. Auch der reflektierte Strahl \vec{R} muß in der selben Ebene liegen. Nun zerlegen wir die Richtung der Lichtquelle in den Teil entlang der Normalen und der Tangente $\vec{L} = \vec{L}_N + \vec{L}_T$. Damit erhalten wir:

$$\vec{L} \cdot \vec{N} = |\vec{L}_N| = \sqrt{1 - |\vec{L}_T|^2} = \sqrt{1 - (\vec{L} \cdot \vec{T})^2} \quad (27)$$

Durch eine ähnliche Umformung kann man das Produkt $\vec{V} \cdot \vec{R}$ nur durch \vec{L} , \vec{V} und \vec{T} ausdrücken oder einfacher gesagt ohne die Normale \vec{N} . Dabei ist zu beachten, daß $\vec{R}_N = -\vec{L}_N$ und $\vec{R}_T = \vec{L}_T$ gilt.

$$\begin{aligned} \vec{V} \cdot \vec{R} &= \vec{V} \cdot (\vec{L}_T - \vec{L}_N) \\ &= \vec{V} \cdot ((\vec{L} \cdot \vec{T})\vec{T} - (\vec{L} \cdot \vec{N})\vec{N}) \\ &= (\vec{L} \cdot \vec{T})(\vec{V} \cdot \vec{T}) - (\vec{L} \cdot \vec{N})(\vec{V} \cdot \vec{N}) \\ &= (\vec{L} \cdot \vec{T})(\vec{V} \cdot \vec{T}) - \sqrt{1 - (\vec{L} \cdot \vec{T})^2} \sqrt{1 - (\vec{V} \cdot \vec{T})^2} \end{aligned} \quad (28)$$

In OpenGL können Texturkoordinaten durch eine 4x4 Matrix transformiert werden, sodaß die Berechnung von $\vec{L} \cdot \vec{T}$ und $\vec{V} \cdot \vec{T}$ in Hardware geschehen kann. Mit der Transformationsmatrix \mathbf{M} werden sie dann direkt aus der Tangente \vec{T} erzeugt.

$$\mathbf{M} = \frac{1}{2} \begin{pmatrix} L_1 & V_1 & 0 & 0 \\ L_2 & V_2 & 0 & 0 \\ L_3 & V_3 & 0 & 0 \\ 1 & 1 & 0 & 2 \end{pmatrix} \quad (29)$$

Dadurch erhält man:

$$\begin{aligned} t_1 &= \frac{1}{2}(\vec{L} \cdot \vec{T} + 1) \\ t_2 &= \frac{1}{2}(\vec{V} \cdot \vec{T} + 1) \end{aligned} \quad (30)$$

Mit Hilfe dieser Texturkoordinaten kann nun eine Textur berechnet werden, sodaß die komplette Beleuchtung in Hardware geschehen kann. Durch Verwendung von Farben, um bestimmte Eigenschaften einer Feldlinie oder des darunter liegenden Vektorfeldes zu visualisieren oder durch Verwendung von Transparenz, um zum Beispiel die Richtung des Vektorfeldes zu zeigen oder die Sicht auf Feldlinien im Inneren zu ermöglichen, kann das Ergebniss weiter verbessert werden.

Die Verteilung der Feldlinien basiert auf einer einfachen Monte Carlo Methode. Das Vektorfeld wird in regelmäßige kleine Zellen unterteilt, die je nach Interesse gewichtet werden. Das Gewicht einer Zelle bestimmt dabei die Wahrscheinlichkeit, daß ein neuer Startpunkt in ihr gesetzt wird. Die Feldlinien werden dann von dort aus mittels eines Runge-Kutta Verfahrens in beide Richtungen über eine bestimmte Länge L berechnet. L kann dabei auch so angepaßt werden, daß die Divergenz des Vektorfeldes kompensiert wird.

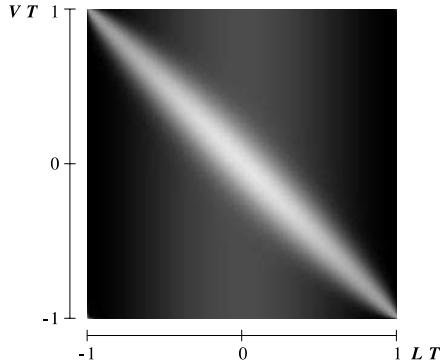


Abbildung 8: Textur zur Beleuchtung der Feldlinie, $k_a = 0,1$, $k_d = 0,3$, $k_s = 0,6$ und $n = 40$. [ZSH96]

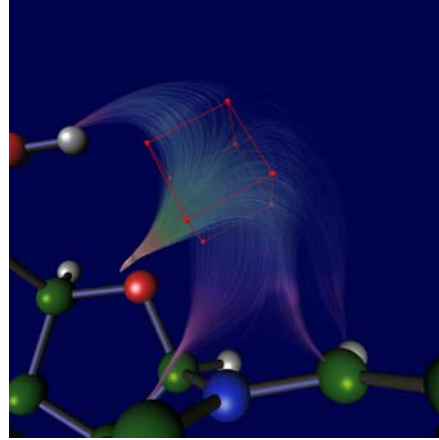


Abbildung 9: Visualisierung mittels beleuchteter Feldlinien. Die Startpunkte für die Feldlinien liegen alle innerhalb des roten Würfels. [ZSH96]

Bei allen Vorteilen, die dieses Verfahren besitzt, muß jedoch auf eine global dichte Repräsentation des Vektorfeldes verzichtet werden, da nicht in jedem Pixel des erzeugten Bildes und damit auch nicht in jedem Punkt des Vektorfeldes eine Feldlinie visualisiert werden kann. Trotz dieses Nachteils sind die erhaltenen Bilder visuell sehr intuitiv. Die fehlende globale Dichte wird durch eine interaktive Wahl eines interessanten Bereichs kompensiert, so daß dieses Verfahren für die meisten Vektorfelder gute Ergebnisse liefert.

3.3.2 Strömungsflächen

Im Gegensatz zu Feldlinien, die als Spur eines Partikel innerhalb des Vektorfeldes interpretiert werden können, handelt es sich bei Strömungsflächen um die Spur einer Linie oder Kurve.

Die einfachste Betrachtungsmöglichkeit für diese Flächen ist, sie als parametrisierte Flächen $\mathbf{r}(s, t)$ zu behandeln, wobei s der Parameter der ursprünglichen Kurve $\mathbf{r}(s, 0)$ und t der Parameter entlang der Feldlinien ist. Die Kurve $\mathbf{r}(\textit{konstant}, t)$ ist eine Feldlinie und $\mathbf{r}(s, \textit{konstant})$ eine Zeitlinie. Um die Strömungsfläche zu berechnen, wird nun einfach die ursprüngliche Kurve in die Punkte r_{i0} diskretisiert und eine diskrete Approximation der Feldlinien wird in r_{ij} berechnet. Dadurch erhält man ein regelmäßiges Gitter, das die Strömungsfläche approximiert.

So einfach dieser Ansatz auch ist, so katastrophal sind auch die erzielten Ergebnisse bei konvergenten und divergenten Vektorfeldern oder an kritischen Punkten (siehe Abblindung 10). Eine adaptive Anpassung der Integrationschritte und der Auflösung der Zeitlinie wie sie Hultquist [Hul92] vorschlägt, oder eine Berechnung mit Oberflächenpartikeln [vW93b] löst diese Probleme nur teilweise.

Van Wijk wählt einen anderen Ansatz, die implizite Strömungsfläche [vW93a].

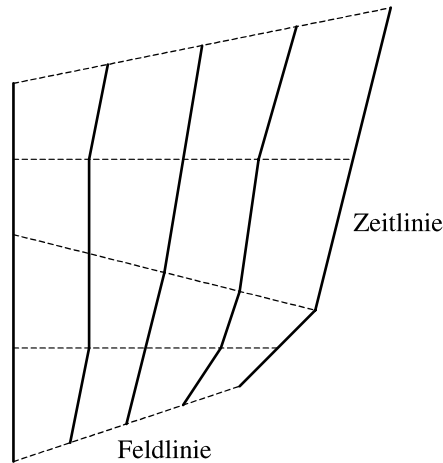


Abbildung 10: Strömungsfläche mit regelmäßigem Gitter approximiert.

Bei diesem Ansatz wird eine Funktion f berechnet, die mit $f(x) = C$ eine ganze Menge von Strömungsflächen berechnen kann. Diese Funktion muß folgende Forderung erfüllen:

$$\nabla f \cdot \vec{v} = 0 \quad (31)$$

Oder einfacher, die Normale von $f = C$ muß an jeder Stelle senkrecht auf dem entsprechenden Vektor stehen. Die Berechnung von f selbst erfolgt mit dem Algorithmus von Leonard [Leo79] und wird dabei auf drei Dimensionen erweitert, indem sie nacheinander interiert werden.

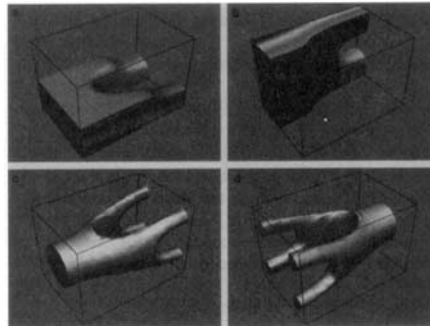


Abbildung 11: Beispiele impliziter Strömungsflächen. [vW93a]

Das größte Problem dieses Algorithmus ist die fehlende dichte Repräsentation. Es wird immer nur eine einzige Strömungsfläche visualisiert und das Verständnis des Vektorfeldes kann so nur durch mehrere Bilder erreicht werden. Jedoch erhält man durch eine zusätzliche Animation einen sehr guten Überblick über die Eigenschaften des Vektorfeldes.

3.3.3 Raycasting von Vectorfeldern

Ein Ansatz genau diese dichte Repräsentation zu erreichen verwendet das Raycasting. Beim eigentlichen Raycasting [Sab88] werden nur ein oder mehrere skalare Werte eines Datensatzes visualisiert. Durch eine zusätzliche Beleuchtung des Datensatzes in Abhängigkeit von den zu visualisierenden Vektoren kann auch das Vektorfeld dargestellt werden. Außer der hinzugefügten Beleuchtung wird der normale Raycastingalgorithmus verwendet.

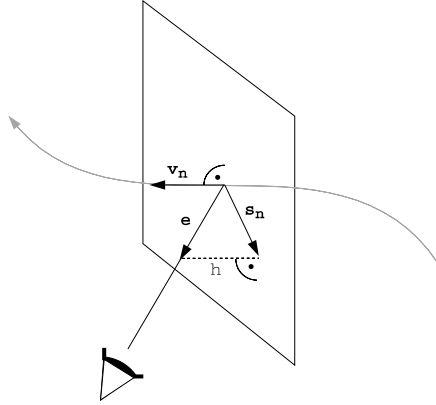


Abbildung 12: Berechnung der Normalen \vec{s}_n in Abhängigkeit von der Blickrichtung \vec{e} . [Frü96]

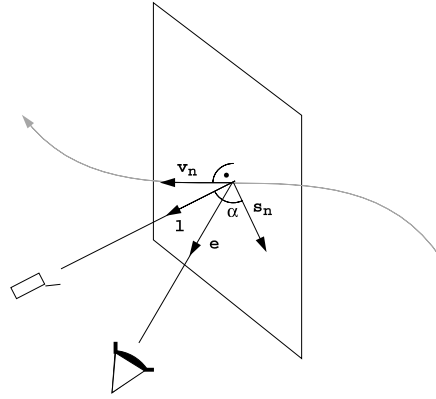


Abbildung 13: Beleuchtung in Abhängigkeit vom Winkel α zwischen \vec{l} und \vec{s}_n . [Frü96]

Frühauf [Frü96] wählt dabei eine Beleuchtung, die mit Hilfe der Feldlinien oder besser gesagt, dem lokalen Vektor arbeitet. Da man auf einer Feldlinie eine ganze Ebene von Normalen hat, wird die Normale in Richtung des Betrachters gewählt. Die Normale \vec{s}_n ist dann vom Betrachter \vec{e} aus gesehen:

$$\vec{s}_n = \vec{e} - (|\vec{h}| \cdot \vec{v}_n) \quad (32)$$

Mit \vec{v}_n als Einheitsvektor der Richtung der lokalen Feldlinie. Der Abstand h ist dabei:

$$h = \vec{e} \cdot \vec{v}_n \quad (33)$$

Frühauf rechnet dabei nur mit einer diffusen Beleuchtung einer direktionalen Lichtquelle mit der Richtung \vec{l} beziehungsweise mit Winkel α zur Normale. Damit ergibt sich für die beleuchtete Farbe eines Samplepunktes mit der Farbe c :

$$c_s = c \cdot \cos\alpha \quad (34)$$

Dadurch werden die Bereiche an denen der Vektor in Lichtrichtung zeigt am dunkelsten, während die Bereiche an denen der Vektor orthogonal zur Lichtrichtung steht am hellsten sind. Da die Ergebnisse stark von der Positionierung der Lichtquelle abhängen, ist ein guter erster Ansatz die Lichtrichtung in Richtung

des durchschnittlichen Vektors $v_m^{\vec{}}$ (siehe Abbildung 14) also der Hauptrichtung des Vektorfeldes fest zu legen. Dadurch werden abweichende Vektoren heller, während Vektoren in der Hauptrichtung dunkler werden.

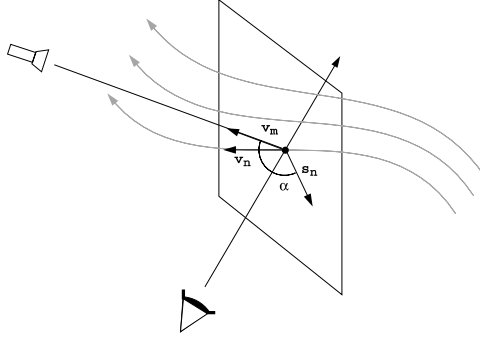


Abbildung 14: Hauptrichtung des Vektorfeldes $v_m^{\vec{}}$ als Richtung der Lichtquelle \vec{l} . [Frü96]

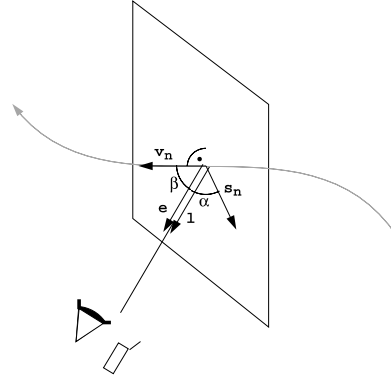


Abbildung 15: Blickrichtung \vec{e} als Richtung der Lichtquelle \vec{l} . [Frü96]

Eine andere mögliche Wahl ist, die Lichtquelle immer beim Betrachter zu setzen, wodurch sich die Berechnung stark vereinfacht. Für die diffuse Beleuchtung wird nicht die Normale $s_n^{\vec{}}$, sondern nur der Cosinus zwischen Lichtrichtung \vec{l} und der Normalen benötigt. Nach Abbildung 15 gilt:

$$\alpha = \frac{\pi}{2} - \beta \quad (35)$$

Wobei β der Winkel zwischen dem Vektor $v_n^{\vec{}}$ und der Blickrichtung \vec{e} ist. Die Gleichung 34 wird dann vereinfacht zu:

$$c_s = c \cdot \sin\beta \quad (36)$$

Durch diese Beleuchtung wird ein Vektor, der zum Betrachter oder von ihm weg zeigt, dunkler, während ein Vektor orthogonal zum Betrachter heller wird. Damit entsteht eine intuitive aber richtungsabhängige Visualisierung der Richtungen eines Vektorfeldes.

Leider kann man mit diesem Verfahren nur sehr einfache Vektorfelder erkennen, da selbst bei dem tornadoähnlichen Feld in den Abbildungen 16 bis 18 die Richtung des Vektorfeldes nur recht schwer zu erkennen ist, was auch durch eine zusätzliche Modifikation der Opazität durch den Winkel α kaum behoben wird. Ein weiterer Punkt, der in diesem Verfahren vernachlässigt wird, ist die Topologie des Vektorfeldes, die nur durch eine zusätzliche Kodierung der Vektorlänge auf die Farben der einzelnen Samplepunkte visualisiert werden kann. Doch durch Verwendung all dieser Verbesserungen kann eine recht gute Visualisierung der globalen Eigenschaften erreicht werden.

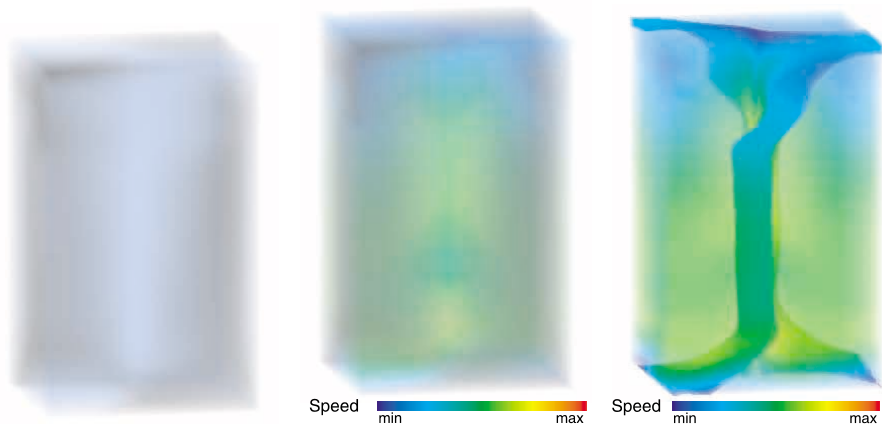


Abbildung 16: Beleuchtete Feldlinien ohne Farbkodierung, Lichtquelle beim Betrachter. [Frü96]

Abbildung 17: Beleuchtete Feldlinien mit Farbkodierung der Vektorlänge. [Frü96]

Abbildung 18: Beleuchtete Feldlinien, niedrige Werte von α mit hoher Opazität. [Frü96]

3.3.4 3D LIC

Volumen LIC ist die konsequente Erweiterung des LIC Verfahrens aus dem vorherigen Kapitel auf den dreidimensionalen Fall. Bei dieser Erweiterung treten jedoch einige nicht zu vernachlässigende Probleme auf, die den Einsatz dieser Methode stark eingeschränkt haben. Interrante und Grosch [IG97] versuchen die Hintergründe genauer zu betrachten, die dazu führen, daß man dichte Volumen nur sehr schwer verständlich darstellen kann und versuchen Lösungen für dieses Problem anzubieten.

Im ersten Schritt muß ein Ausgangsbild gewählt werden. Im zweidimensionalen Fall wurde einfach ein opaques weißes Rauschen benutzt. Um jedoch im Dreidimensionalen ein möglichst gutes Ergebnis zu erzielen, sollte das Ausgangsbild nur an sehr wenigen Stellen opaque sein und die Faltung erfolgt dann sowohl über die Farbwerte als auch über die Opazität. Die genaue Wahl dieser wenigen opaquen Stellen kann dabei auch an das Vektorfeld angepaßt werden oder rein zufällig sein.

Bei den bis jetzt erzeugten Bildern (siehe Abbildung 19) können zur Verbesserung der Darstellung noch die Opazitäten mit der Länge des darunterliegenden Vektors multipliziert werden, wodurch Bereiche mit kurzen Vektoren durchsichtiger werden.

Um die Wahrnehmung des Vektorfeldes zu verbessern und nicht den Eindruck zu erwecken, daß es sich um ein flaches Gebilde handelt, wurden von Interrante und Grosch einige Verfahren angewendet, die auch für die meisten anderen Visualisierungsverfahren von Vektorfeldern anwendbar sind.

Die einfachsten Möglichkeiten sind eine Änderung der Farbe entlang einer Achse des Vektorfeldes oder auch die Beleuchtung nach dem Verfahren von Phong [Pho75] und der Verallgemeinerung von Banks [Ban94]. Jedoch reichen alle diese kleinen Verfeinerungen nur selten aus, um wirklich einen dreidimensionalen Eindruck des Vektorfeldes zu vermitteln.

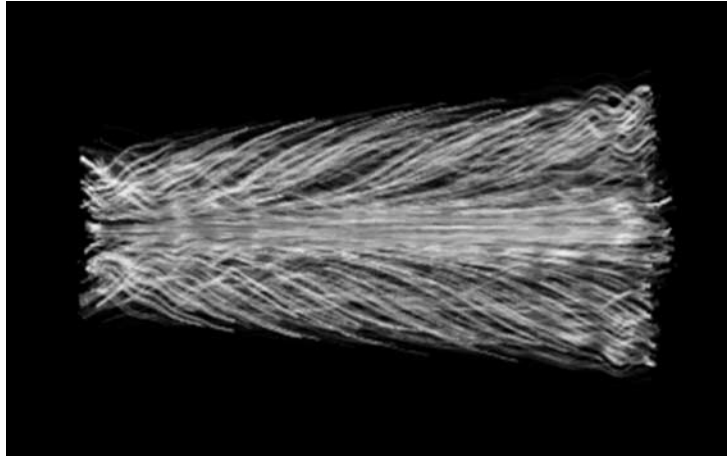


Abbildung 19: 3D LIC mit Beleuchtung nach Banks [Ban94], trotzdem fehlen wichtige Tiefeninformationen. [IG97]

Ein Unterschied in der Tiefe benachbarter Bereiche wird vom menschlichen pseudo dreidimensionalen Sehen durch einen kleinen Bereich, der nur in einem Bild zu sehen ist, erkannt. Daher schlugen Dooley und Cohen die Nutzung sogenannter Halo's [DC90] (schmale dunkle Umrandungen) vor, um eindeutig zu kennzeichnen, welche Objekte im Vordergrund liegen. Dieses Verfahren wird vor allem in Zeichnungen verwendet, die sonst keine Tiefeninformation enthalten und wurden in die Liniendarstellung dreidimensionaler Objekte übernommen (siehe Abbildung 20).

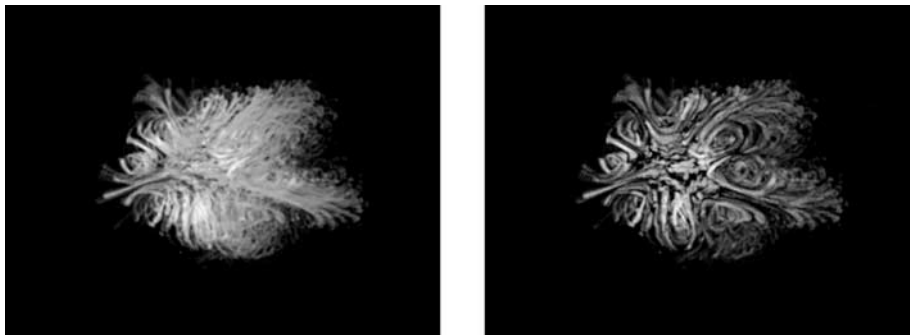


Abbildung 20: 3D LIC mit Halo zur Verbesserung der dreidimensionalen Wahrnehmung (rechts) und ohne Halo (links). [IG97]

Da es sich bei den Objekten im Ausgabebild nicht um binäre Gebilde handelt, also ein Punkt auch nur zum Teil in einer Feldlinie liegen kann, muß auch das Halo auf ähnliche Weise definiert sein. Dazu wird ein zweites Eingabebild benutzt, das dem Ersten genau entspricht, außer daß die opaquen Punkte etwas größer sind. Während nun weiterhin der Farbwert aus dem ursprünglichen Eingabebild verwendet wird, wird die Opazität mit Hilfe des Zweiten berechnet. Dadurch wird jede Feldlinie von einem schmalen dunklen Bereich umgeben.

Dabei muß jedoch beachtet werden, daß jeweils der erste Eintritt in ein Halo ignoriert werden muß, also darf nur beim Verlassen des Halo's eine Abschwächung stattfinden, da jede Linie von ihrem eigenen Halo umgeben wird.

Die nun erzeugten Bilder können dann noch, zum Teil sogar interaktiv, zur Darstellung angepasst werden, zum Beispiel mit Farben versehen werden oder nur ausschnittsweise betrachtet werden. Zuletzt bleibt noch zu erwähnen, daß die Berechnung von 3D LIC Bildern sehr aufwendig werden kann und je nach gewünschtem Ergebnis bis zu einer Stunde oder mehr in Anspruch nehmen kann.

3.3.5 Visualisierung Mittels einer Sonde

Als Gegenstück zu den bisherigen globalen Visualisierungen ist das Verfahren von Leeuw und Wijk [dLvW93] gedacht, da Mittels einer Sonde die lokalen Eigenschaften des Vektorfeldes visualisiert werden. Ein einfacher Pfeil zeigt jedoch nur die lokale Richtung und Länge des Vektorfeldes, obwohl manchmal gerade Eigenschaften, wie die lokale Änderung interessant sind. Diese lokale Änderung also die Ableitung des Vektorfeldes ist ein Tensor.

In einem Vektorfeld ist die Approximation erster Ordnung eines Punktes x in der Nähe von x_0 :

$$\mathbf{u}(x) = \mathbf{u}(x_0) + \mathbf{J}(x - x_0) \quad (37)$$

Im Dreidimensionalen ist die Ableitung oder Jacobi Matrix eines Vektorfeldes $\mathbf{u}(x) = (u, v, w)$ gegeben durch:

$$\mathbf{J} = \nabla \mathbf{u} = \begin{pmatrix} u_x & v_x & w_x \\ u_y & v_y & w_y \\ u_z & v_z & w_z \end{pmatrix} \quad (38)$$

Zur Visualisierung muß dieser Tensor jedoch in zwei Teile, einen symmetrischen und einen antisymmetrischen, aufgespalten werden (siehe [Bat67]):

$$\mathbf{J} = \mathbf{J}^{(s)} + \mathbf{J}^{(a)} \quad (39)$$

Dabei gilt:

$$\begin{aligned} \mathbf{J}^{(s)} &= \frac{\mathbf{J} + \mathbf{J}^T}{2} \\ \mathbf{J}^{(a)} &= \frac{\mathbf{J} - \mathbf{J}^T}{2} \end{aligned} \quad (40)$$

Der antisymmetrische Teil hat drei unabhängige Komponenten, die die Rotation beschreiben. Der symmetrische Teil des Tensors beinhaltet die Hauptachsen des Tensors und die entsprechende Streckung entlang dieser Achsen. Die Achse entlang der größten Streckung ist die Hauptachse des Tensors.

Ein Tensor kann als die Verzerrung eines infinitesimal kleinen Teilchens am Punkt, an dem der Tensor definiert wurde, interpretiert werden. Dadurch kann die Zerlegung des Tensors als Einschränkung auf zwei unterschiedliche Arten von

Teilchen interpretiert werden. Während sich ein Teilchen nur drehen kann, also nur der antisymmetrische Teil anwendbar ist, kann das andere Teilchen nicht rotiert sondern nur verzerrt werden, also ist nur der symmetrische Teil anwendbar. Eine weitere Zerlegung führt zu weiteren Einschränkungen der Teilchen.

Um den Tensor nun auf eine sinnvolle Weise visualisieren zu können, muß er in ein lokales Koordinatensystem transformiert werden, sodaß eine Unterscheidung zwischen Änderungen entlang des Vektors und parallel dazu möglich ist. In diesem lokalen Koordinatensystem, einem Frenet Koordinatensystem [Blo90], zeigt die x-Achse entlang des Vektors und die y-Achse ist parallel zur Krümmung der zum Vektor gehörigen Feldlinie. Der Krümmungsvektor \vec{c} einer Feldlinie $p(s)$ durch den Ursprung ist gegeben durch:

$$\vec{c} = \frac{d^2 p}{ds^2} \quad (41)$$

Nimmt man nun ein imaginäres Teilchen, das sich nach der Approximation erster Ordnung entlang der Feldlinie bewegt:

$$\mathbf{u}(t) = \mathbf{u}_0 + \mathbf{J}\mathbf{u}_0 t \quad (42)$$

und benutz die Tatsache, daß:

$$\frac{d\mathbf{p}}{ds} = \frac{\mathbf{u}(t)}{|\mathbf{u}(t)|} \quad (43)$$

gilt, kann man zeigen, daß für den Krümmungsvektor folgende Relation gilt:

$$\vec{c} = \frac{\mathbf{J}\vec{u}(\vec{u} \cdot \vec{u}) - \vec{u}(\vec{u} \cdot \mathbf{J}\vec{u})}{|\vec{u}|^3} \quad (44)$$

Die Basis des lokalen Koordinatensystems ist dann folgendermassen gegeben:

$$\left(\frac{\vec{u}}{|\vec{u}|}, \frac{\vec{c}}{|\vec{c}|}, \frac{\vec{u} \times \vec{c}}{|\vec{u} \times \vec{c}|} \right) \quad (45)$$

Falls die Feldlinie keine lokale Krümmung besitzt, kann jedes orthonormale Koordinatensystem verwendet werden, in dem die x-Achse in Richtung des Vektors zeigt.

Um den transformierten Tensor zu visualisieren wird er in zwei Teile zerlegt, die Änderung in Richtung des Vektorfeldes und die entgegen dem Vektorfeld. Die Komponenten in Richtung des Vektorfeldes beinhalten sowohl die Ableitung entlang der x-Achse (u_x , v_x und w_x) und des Vektors \vec{u} (u_x , u_y und u_z). Die Komponenten entgegen der Richtung des Vektorfeldes besteht aus einer 2×2 Matrix (v_y , v_z , w_y und w_z). Diese Matrix wird wieder in einen symmetrischen (Torsion) und einen antisymmetrischen (Konvergenz) Teil getrennt.

Der Tensor wird in fünf Komponenten aufgeteilt, Krümmung, Scheerung, Beschleunigung, Torsion und Konvergenz. Die Geschwindigkeit, also die Länge des Vektors, wird durch die Länge des durch die Krümmung gebogenen Pfeils visualisiert. Auf dem Pfeil sind außerdem zwei Markierungen für die Rotation angebracht. Die Beschleunigung wird durch einen halben Ellipsoid visualisiert,

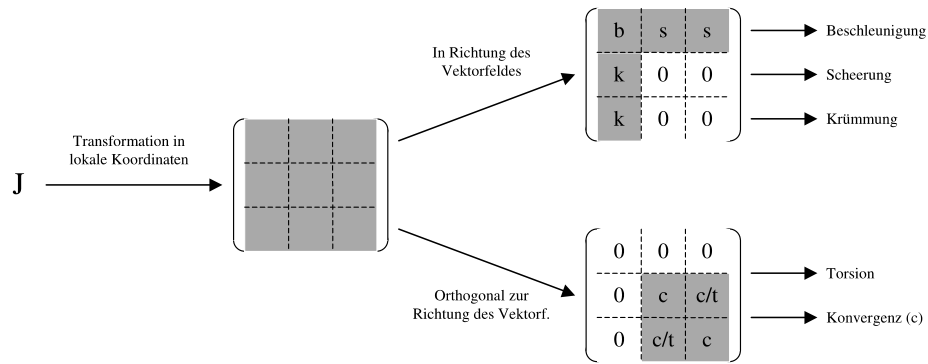


Abbildung 21: Zerlegung des Tensors für die Sonde.

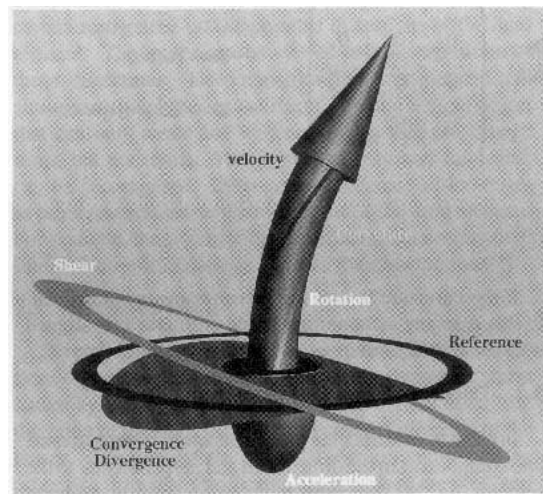


Abbildung 22: Komponenten der Sonde. [dLvW93]

der an der selben Stelle wie der Pfeil ansetzt. Konvergenz und Scheerung werden als gebogene, beziehungsweise verzerrte Kreisscheiben und einer Referenzscheibe visualisiert.

3.3.6 Texturesplats

Bei diesem Verfahren von Crawfis [CM93] handelt es sich um einen ähnlichen Grundgedanken, wie beim Raycasting von Vektorfeldern. Ein bestehender Algorithmus zur Volumenvisualisierung wird auf die Visualisierung dreidimensionaler Vektorfelder erweitert.

Bei den für das Splatting verwendeten Filterkernen beschränkt sich Crawfis auf die rotationssymmetrischen Filterkerne mit einer möglichst optimalen Rekonstruktionsfunktion. Die Rekonstruktionsfunktion ist die Funktion, die zur Erzeugung des Filterkerns verwendet wird. Eine optimale Rekonstruktionsfunktion liefert bei gleichmäßig verteilten Samplepunkten eine genaue Darstellung des zu visualisierenden Datensatzes. Die Rekonstruktionsfunktion wird in zwei

Polynome dritter Ordnung aufgeteilt, die C^1 -stetig am Punkt s in einander übergehen. Dabei ist zu beachten, daß der Filterkern im Ursprung auch C^1 -stetig sein muß, da ein rotationssymmetrischer Filterkern erzeugt wird:

$$\begin{aligned} p(r) &= a + br^2 + cr^3 \\ q(r) &= d(t-r)^2 + e(t-r)^3 \end{aligned} \quad (46)$$

Mittels eines Optimierungsalgorithmus [Gay83] wurde ein lokales Minimum für einen annähernd kleinen Wert von t gesucht und bei $t = 1,556228$ und $s = 0,889392$ gefunden. Nach der Skalierung der Rekonstruktionsfunktion bleibt dann:

$$g(r) = \begin{cases} 0,557526 - 1,157743r^2 + 0,671033r^3 & 0 \leq r < s \\ 0,067599(t-r)^2 + 0,282474(t-r)^3 & s \leq r < t \\ 0 & t \leq r \end{cases} \quad (47)$$

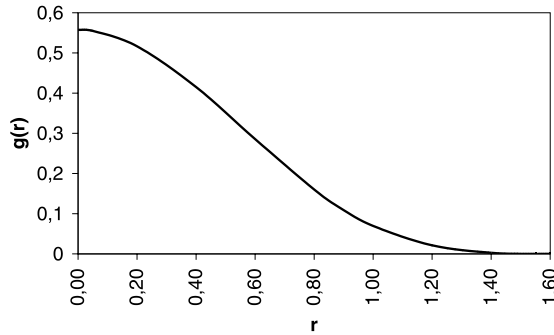


Abbildung 23: Filterkern mit möglichst geringem Fehler bei gleich verteilten Samplepunkten. Nur 1 Bit bei 8-Bit Auflösung.

Auf diesen Splat werden nun die kleinen Partikel, die in einer Richtung verwischt wurden, aufgetragen. Die Richtung der Verwischung ist für alle Splats in der xy -Ebene gleich, da sie durch Rotation verändert werden kann, nur die Länge der Verwischung in Richtung der z -Achse und damit auch der xy -Ebene wird variiert. Also kann sie nicht durch eine weitere Rotation oder Streckung erzeugt werden, ohne daß der Filterkern davon betroffen wird.

3.3.7 Particle Splatting

Hinter diesem neuen Verfahren steckt ein ganz ähnlicher Ansatz wie hinter dem Verfahren von Crawfis. Es werden jedoch nicht mehrere Partikel mit einer zusätzlichen Volumeninformation, sondern nur einzelne Partikel über ein dem Splatting sehr ähnlichen Verfahren auf ein zweidimensionales Bild projiziert.

Ziel dieses Verfahrens ist eine interactive und animierte Darstellung des Vektorfeldes zu erreichen, um so das Problem der Verdeckung so weit wie möglich zu umgehen und einen dreidimensionalen Eindruck des Vektorfeldes zu bekommen.

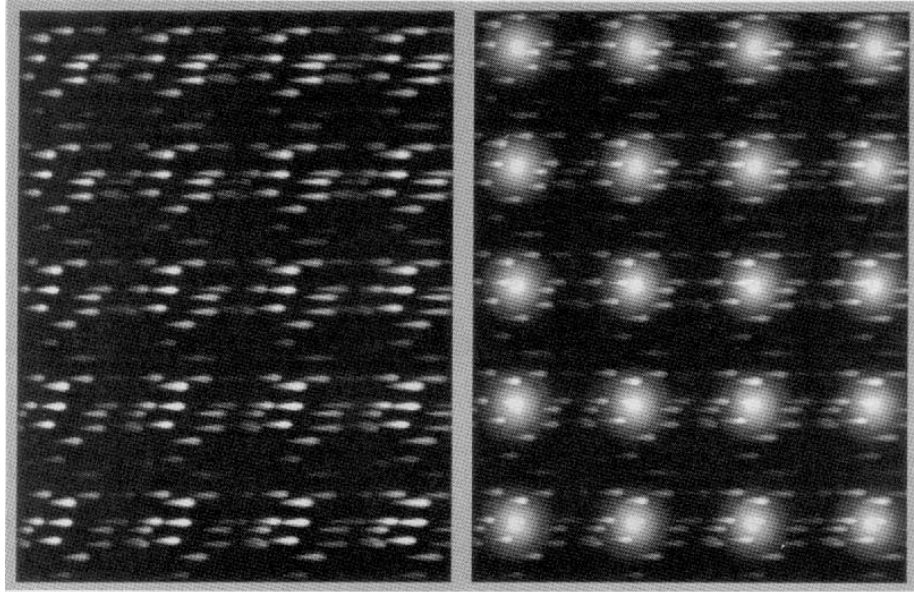


Abbildung 24: Textur für Texturesplats, links Intensität, rechts Opazität. [CM93]

Für diese Art der Darstellung werden jedoch Anforderungen an den Datensatz gestellt, die er nicht unbedingt von sich aus erfüllen kann.

Die Visualisierung wird dabei in verschiedenen Teile aufgespalten:

- Laden des Datensatzes
- Konvertierung in Octree
- Definition der Gewichtsfunktion
- Konstruktion des Octrees für Partikel
- Verteilung des Partikel
- Animation

Während der Animation werden die Partikel mit vorher berechneten Texturen visualisiert und entlang der Feldlinien bewegt. Außerdem werden nicht mehr benötigte Partikel entfernt und neue hinzu gefügt. Der große Vorteil dieses Ansatzes liegt darin, daß die Animation an sich nur sehr wenig Rechenzeit benötigt und dadurch interaktiv wird. Durch die nun dauernd wechselnde Verdeckung erhält man einen guten Einblick in das Vektorfeld und die sich bewegenden Partikel erzeugen einen sehr guten dreidimensionalen Eindruck.

Das Verfahren benötigt eine Datenstruktur, die einen möglichst schnellen Zugriff auf den Vektor an einer beliebigen Stelle des Datensatzes ermöglicht, ohne vor der Darstellung eine aufwendige Berechnung durchführen zu müssen. Schnelle und einfach zu berechnende Repräsentationen sind zum Beispiel ein reguläres Gitter oder ein Octree [Sed92]. Für das reguläre Gitter muß eine genügend hohe Auflösung gewählt werden, so daß das Vektorfeld in dieser Repräsentation

unter Umständen sehr viel Speicher benötigt. Auf der anderen Seite muß bei einem Octree eine Entscheidung bei jeder Zelle getroffen werden, ob sie weiter unterteilt werden muß oder ob die aktuelle Auflösung an dieser Stelle genügt.

Um für jedes Vektorfeld eine ausreichende Repräsentation ohne starke Unstetigkeiten zwischen unterschiedlich großen Zellen zu erhalten, benötigt man eine gute Fehlerberechnung. Eine exakte Berechnung des Fehlers scheidet jedoch aus, da man die Differenzfunktion zwischen dem eigentlichen Vektorfeld und dem durch den Octree erzeugten Vektorfeld innerhalb einer Zelle berechnen müsste. Diese Differenzfunktion muß dann quadriert und über die gesamte Zelle integriert werden. Da diese Berechnung viel zu lange dauern würde, braucht man eine möglichst stabile Fehlerabschätzung. Dazu wird die Differenz an insgesamt 19 Stellen der aktuellen Zelle berechnet. Es wird an jeder Mitte einer Kante, einer Fläche und in der Mitte der Zelle verglichen und der quadratische mittlere Fehler berechnet. Die Eckpunkte werden für die Rekonstruktion des Vektorfeldes innerhalb der Zelle benutzt und sind daher exakt und für die Fehlerabschätzung nicht interessant.

Durch diese Methode der Umwandlung des Vektorfeldes können jedoch wichtige Features, wie zum Beispiel dicht benachbarte kritische Punkte und der genaue Rand des Datensatzes, verloren gehen. Um dieses Problem zu umgehen kann man beim regulären Gitter die Auflösung bestimmen und beim Octree den maximal zulässigen Fehler. Als noch schnellere, aber auch leider noch fehleranfälliger Repräsentation kann noch ein Octree verwendet werden, in dem in jeder Zelle nur ein einziger Vektor gespeichert wird, also keine Interpolation verwendet wird. Bei der Fehlerabschätzung werden nun auch die Eckpunkte benötigt. Dieser Octree verfügt bei gleichem Fehler immer über eine höhere Tiefe und damit auch zum Teil über einen höheren Speicherbedarf. Daher ist diese Repräsentation in den meisten Fällen nicht von Nutzen.

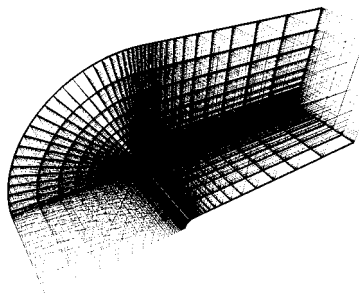


Abbildung 25: Curvilineares Gitter des Blunt Fin Datensatzes

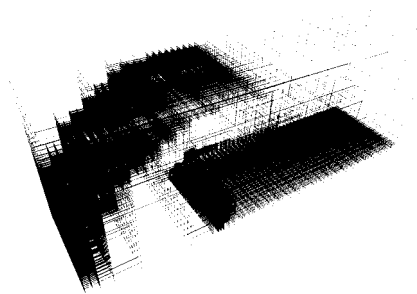


Abbildung 26: Octreedarstellung des Blunt Fin Datensatzes

Doch für eine Animation wird noch ein schnelles Verfahren zur Berechnung der Feldlinien, entlang denen sich die Partikel bewegen, benötigt. Trotzdem sollte die Feldlinie auch möglichst genau sein, also einen geringen Fehler hoher Ordnung besitzen. Um diese Anforderungen zu erfüllen wurden mehrere Integrationsverfahren mit variabler Schrittweite getestet. Als einfachste Methode bietet sich ein Euler Verfahren an. Zwar bekommt man mit diesem Verfahren in sehr schneller Zeit einen sehr geringen Fehler, aber da es sich um ein Verfahren erster Ordnung handelt, erhält man einen Fehler zweiter Ordnung. Als nächstes

wurde ein Runge Verfahren zweiter Ordnung getestet. Die benötigte Rechenzeit bei gleichem Fehler steigt nur gering. Die Ordnung des Fehler wird auf die dritte Ordnung erhöht. Als letztes Verfahren mit annehmbarer Rechenzeit bleibt dann noch ein Runge-Kutta Verfahren vierter Ordnung. Trotz der nur geringfügig gestiegenen Rechenzeit erreicht man einen Fehler fünfter Ordnung.

$$\begin{aligned} \mathbf{d}_n &= \text{dir}(\mathbf{p}_n) \\ \mathbf{p}_{n+1} &= \mathbf{p}_n + s\mathbf{d}_n \end{aligned} \tag{48}$$

Für einfache Vektorfelder oder die Repäsentation durch einen Octree ohne Interpolation reicht das Euler Verfahren meistens aus. Wird ein Vektorfeld jedoch durch einen Octree mit Interpolation oder durch ein reguläres Gitter repräsentiert kann eine Verwendung des Runge Verfahrens nicht nur die Ordnung des Fehler erhöhen, sondern unter Umständen auch die benötigte Rechenzeit reduzieren.

$$\begin{aligned} \mathbf{d}_n^{(0)} &= \text{dir}(\mathbf{p}_n) \\ \mathbf{d}_n^{(1)} &= \text{dir}\left(\mathbf{p}_n + \frac{s}{2}\mathbf{d}_n^{(0)}\right) \\ \mathbf{p}_{n+1} &= \mathbf{p}_n + s\mathbf{d}_n^{(1)} \end{aligned} \tag{49}$$

Für sehr turbulente Vektorfelder kann es sogar manchmal von Nutzen sein, das Runge-Kutta Verfahren zu verwenden. Zwar wurde bei keinem Datensatz eine Steigerung der Geschwindigkeit beobachtet, aber die Genauigkeit der Feldlinien steigt bei gleichem Fehler stark an.

$$\begin{aligned} \mathbf{d}_n^{(0)} &= \text{dir}(\mathbf{p}_n) \\ \mathbf{d}_n^{(1)} &= \text{dir}(\mathbf{p}_n + s\mathbf{d}_n^{(0)}) \\ \mathbf{d}_n^{(2)} &= \text{dir}\left(\mathbf{p}_n + \frac{s}{2}\mathbf{d}_n^{(1)}\right) \\ \mathbf{d}_n^{(3)} &= \text{dir}\left(\mathbf{p}_n + \frac{s}{2}\mathbf{d}_n^{(2)}\right) \\ \mathbf{p}_{n+1} &= \mathbf{p}_n + \frac{s}{6}\mathbf{d}_n^{(0)} + \frac{s}{3}\mathbf{d}_n^{(1)} + \frac{s}{3}\mathbf{d}_n^{(2)} + \frac{s}{6}\mathbf{d}_n^{(3)} \end{aligned} \tag{50}$$

Wie bereits erwähnt mußten die Partikel im Datensatz so verteilt werden, daß Bereiche mit geringer Turbulenz durch wenige Partikel repräsentiert werden, während Bereiche mit starker Turbulenz oder mit kritischen Punkten durch viele Partikel visualisiert werden müssen. Zu diesem Zweck benötigt man eine Funktion, die zu jeder Stelle im Datensatz den Grad des Interesses oder das Gewicht berechnen kann. Um dem Benutzer dabei eine gute Kontrolle über diese Funktion zu gestatten, kann das Gewicht jedes Punktes über einige Parameter definiert werden.

Um die Gewichtsfunktion zu definieren kann sie an einige Eigenschaften des Datensatzes gekoppelt werden. Neben einem über den ganzen Datensatz konstantem Gewicht kann man auch direkt einen skalaren Parameter oder die Länge des Vektors verwenden. Um jedoch eine bessere Verteilung zu erhalten kann man auch noch die lokale Divergenz der Vektoren oder die lokale Divergenz der normierten Vektoren, also den Grad der Rotation, verwenden. Faßt man alle diese

Möglichkeiten zusammen, erhält man die Gewichtsfunktion $\omega(\mathbf{x})$.

$$\omega(\mathbf{x}) = \omega_{konst} + \omega_1(\mathbf{x}) + \dots + \omega_n(\mathbf{x}) + \omega(\mathbf{x})_{div} + \omega(\mathbf{x})_{rot}.$$

Um die Partikel nun zu verteilen, und vor allem um sie während der Animation zu verwalten, benötigt man einen zweiten Octree, in dem jede Zelle über ein vergleichbares Gewicht verfügt, also ungefähr die selbe Anzahl an Partikeln aufnehmen kann. Eine gute Wahl ist dabei, jede Zelle zu unterteilen, in die mehr als acht Partikel passen. Außerdem können Zellen zusammengefaßt werden, wenn von den acht zusammenfassenden Zelle nur eine einzige einen Partikel aufnehmen kann. Um nun einen Partikel einzufügen wird auf jeder Ebene des Octree, von der Wurzel an, jeweils die Zelle gewählt, die noch am meisten Partikel aufnehmen kann. Wird so ein Blatt des Octrees erreicht, wird die Zelle dort an einer zufälligen Position platziert und eingetragen.

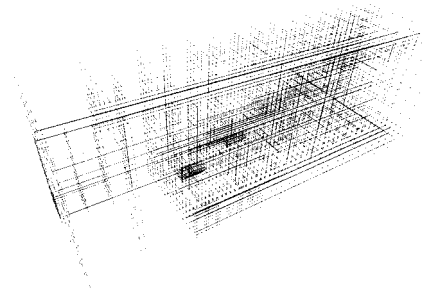


Abbildung 27: Octree zur Verteilung der Partikel im Blunt Fin Datensatz

Während der Animation werden die Partikel innerhalb des Octrees entsprechend verschoben. Um nun eine Häufung der Partikel an Stellen mit niedrigem Gewicht zu vermeiden, werden die Partikel die zu viel in einer Zelle sind gealtert, während Partikel in unterbesetzten Zellen verjüngt werden. Übersteigt ein Partikel das vom Benutzer definierte Alter oder verläßt er den Datensatz wird er entfernt. Ein so frei gewordener Partikel wird nun wieder entsprechend der Gewichtsfunktion in den Octree eingefügt.

Nachdem nun eine Verteilung und eine Animation der Partikel möglich ist, bleibt noch die Wahl der Partikel selbst. Zum einen sollten die Partikel gut sichtbar sein und die Richtung und Geschwindigkeit des Vektorfeldes visualisieren. Auf der anderen Seite sollten sie aber auch einen guten Einblick in das Vektorfeld ermöglichen und einen dreidimensionalen Eindruck vermitteln. Wie man in Abbildung 28 erkennen kann, eignen sich Pfeile unterschiedlicher Länge für die erste Anforderungen sehr gut. Um auch den Einblick zu ermöglichen sind etwas aufwendigere Überlegungen nötig. Partikel ohne Transparenz erzeugen durch die starke Verdeckung zwar einen guten dreidimensionalen Eindruck, aber der Einblick in das Vektorfeld ist nicht möglich. Bei konstanter Transparenz wird zwar der Einblick ins Vektorfeld verbessert, aber der dreidimensionale Eindruck ist fast verloren gegangen. Auch eine Transparenz entsprechend der Dicke des Vektorpfeils löst dieses Problem nicht. Erst eine schmale schwarze Umrandung um den Pfeil mit nicht konstanter Transparenz ermöglicht sowohl einen dreidimensionalen Eindruck, als auch einen guten Einblick ins Vektorfeld.

Die Partikel wurden alle mit einem Raytracer und starker Überabtastung berechnet, um eine gute Qualität zu erreichen. Da die Partikel keine Farbe besitzen und nur Luminanz- und Absortionswerte berechnet wurden, können Farbe und Sättigung zur Visualisierung weiterer Parameter verwendet werden.

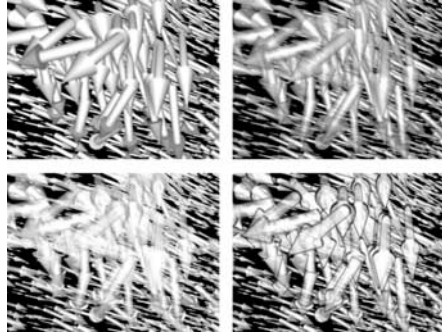


Abbildung 28: Verschiedene Optimierungen der Partikel. Von links oben nach rechts unten: keine Transparenz, konstante Transparenz, Transparenz von der Dicke abhängig und zusätzliche Umrandung

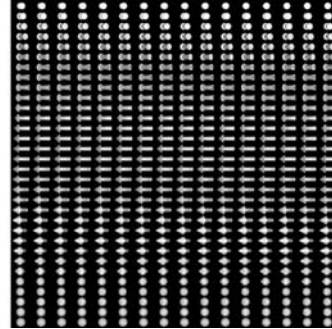


Abbildung 29: Textur mit deren Hilfe die Rotation entlang der y-Achse realisiert wird.

Bei der Darstellung wird ein Verfahren benutzt, daß an das Splatting von Volumendaten angelehnt ist. Hier werden zwar keine Volumen visualisiert, aber auch hier werden vorberechnete Texturen an bestimmten Positionen im Raum dargestellt. Um die Richtung und Länge jedes Partikels an seiner Position zu visualisieren werden insgesamt drei Rotation, zwei Streckungen und eine Translation benötigt. Die erste Rotation um die x-Achse (Abbildung 30) kann man durch rotationssymmetrische Partikel, wie die hier benutzten Pfeile, sparen. Die zweite Rotation um die y-Achse (Abbildung 31) kann man durch die Rotation der Textur erreichen. Für die dritte Rotation entlang der z-Achse (Abbildung 32) wird genau wie beim Texturesplatting von Crawfis [CM93] eine Textur für jede Richtung benötigt. Da es sich bei dieser dritten Rotation nur noch um eine Rotation um maximal 180 Grad handelt, werden diese Winkel in insgesamt 512 verschiedenen Texturen abgelegt.

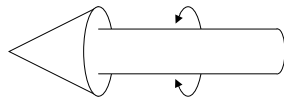


Abbildung 30: Rotation entlang der x-Achse (indifferent).

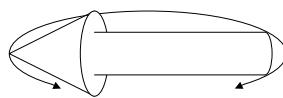


Abbildung 31: Rotation entlang der y-Achse (Textur).

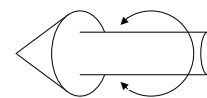


Abbildung 32: Rotation entlang der z-Achse (Rotation).

Die Textur muß jedoch nicht nur an die Rotation um die y-Achse, sondern auch an die Streckung entlang der ursprünglichen x- und z-Achsen angepaßt werden. Wird ein Partikel der Länge l (der Durchmesser beträgt dann $\frac{l}{\sqrt{7}}$) um α rotiert, kann man nicht einfach die Textur zu α verwenden. Um ein Partikel

ohne Verzerrung zu erhalten benötigt man, neben den Streckungsfaktoren xs und ys noch einen Ersatzwinkel für die Textur. Es gilt nun:

$$\frac{xs \cdot \cos\beta}{l \cdot \cos\alpha} = \frac{xs \cdot \sin\beta}{\frac{1}{\sqrt{l}} \cdot \sin\alpha} = 1 \quad (51)$$

Und damit:

$$\begin{aligned} \beta &= \operatorname{atan}\left(\frac{\tan\alpha}{\sqrt{l^3}}\right) \\ xs &= l \cdot \frac{\cos\alpha}{\cos\beta} \\ ys &= \frac{1}{\sqrt{l}} \end{aligned} \quad (52)$$

Durch diese Anpassung können die Partikel gestreckt werden, ohne daß eine Verzerrung auftritt. Da die Partikel mit geringer Perspektive, also nicht parallel projiziert, berechnet wurden, ist der Bereich ohne Verzerrung jedoch eingeschränkt. In dem hier verwendeten Bereich von $l = [0, 5 \dots 5, 0]$ bleibt die Darstellung fast verzerrungsfrei.

Die angepaßte Streckung der Partikel wird nun zusammen mit der zweiten Rotation und der Translation in einer Transformationsmatrix nach Westover [Wes91] zusammengefaßt. Durch die Fähigkeiten von OpenGL ist nun eine weitgehende Entlastung der CPU und damit eine schnelle Darstellung des Partikels möglich, die die Interaktivität dieses Verfahrens garantiert.

Neben den bisher angesprochenen Kontrollparametern hat der Benutzer auch noch einen großen Einfluß auf die Darstellung der Partikel. Neben der Wahl der Anzahl der Partikel kann die Größe und die minimale beziehungsweise maximale Transparenz der Partikel bestimmt werden. Des weiteren kann noch die maximale und die minimale Länge der Partikel gewählt werden, wobei ein Partikel vierfacher Länge nur noch den halben Durchmesser, also das selbe Volumen besitzt. Nicht nur die zusätzlichen Parameter des Datensatzes können für die Farbe und Sättigung verwendet werden, sondern auch eine einfache Beleuchtung der Partikel ist durch Modulation der Helligkeit in Abhängigkeit von der Entfernung zum Betrachter möglich.

Zur Animation kann, wie bereits angesprochen, die Gewichtsfunktion und die minimale Tiefe des Octrees bestimmt werden. Außerdem ist noch die Anpassung des maximalen Alters möglich. Auch der maximale Fehler während der Integration und die Länge der Feldlinie, die für jeden Partikel zur Bewegung berechnet wird, kann vom Benutzer gewählt werden.

4 Implementierung 3D LIC

Die Implementierung der dreidimensionalen Linienintegralfaltung (3D LIC) erfolgte in C++. Der verwendete Pre-Viewer wurde ebenfalls in C++ unter der Verwendung der Qt-Bibliotheken von Troll-Tech und der OpenGL-Bibliotheken geschrieben. Diese Implementierung erzeugt eine PVM-Datei (einen Volumendatensatz) der in einem separaten Viewer angezeigt wird. Die abgebildeten Bilder wurden dabei mit Viewer erzeugt. Die Vectorfelder können entweder als reguläres Gitter oder als CDF Datensatz eingelesen werden.

4.1 Klassenhierarchie

Die Implementierung erfolgt mit fünf Klassen, die zur Implementation unterschiedlicher Verfahren weiter abgeleitet wurden. Basisklassen ohne Funktionalität sind dabei in Abbildung 33 durch eine gestrichelte Umrandung gekennzeichnet.

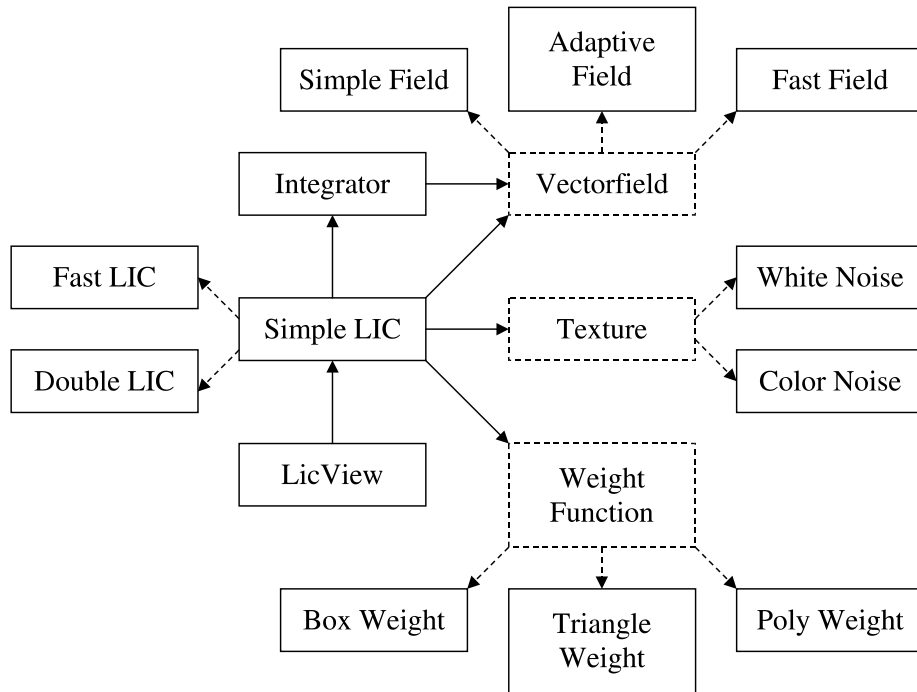


Abbildung 33: Klassenhierarchie der Visualisierung dreidimensionaler Vektorfelder mittels Linienintegralfaltung.

Die zentrale Klasse ist bei dieser Implementierung die LIC-Klasse. Als Basisklasse *SimpleLIC* dient hier das ursprüngliche Verfahren zur Linienintegralfaltung. Davon abgeleitet wurden jeweils *FastLIC*, nach dem Verfahren von Hege und Stalling [HS98] und *DoubleLIC* nach dem selben Verfahren mit Antialias durch doppeltes Filtern. Zur Berechnung der Integrale in der Klasse *Integrator* wird in der Klasse *Vectorfield* eine Funktion aufgerufen, die einen Integrations-schritt mit bestimmter Schrittweite ausführt. In der Klasse *Vectorfield* befindet

sich dabei ein Euler Integrator. Hier lassen sich aber auch beliebig andere Integratoren höherer Ordnung implementieren, da immer nur ein einziger Schritt einer Feldlinie vom Integrator berechnet wird. Der Integrator greift dabei auf die Klasse *Vectorfield* zu, um sich den Vektor an einer beliebigen Stelle zu besorgen. Von dieser Klasse existieren wieder drei unterschiedliche Implementierungen, *SimpleField*, ein reguläres Gitter auf dem trilinear interpoliert wird, *AdaptiveField*, ein Octree mit unterschiedlichen Tiefen und wiederum trilinear Interpolation und *FastField*, ein Octree mit unterschiedlichen Tiefen und ohne Interpolation. Als Gewichtsfunktionen für die so berechneten Feldlinien wurden die B-Splines erster bis dritter Ordnung in den Klassen *BoxWeight*, *TriangleWeight* und *PolyWeight* implementiert, da sie sich auch für das schnelle Verfahren eignen. Als Basisklasse dient hier die Klasse *WeightFunction* die keine Funktionalität, sondern nur ein Interface für die LIC-Klassen bildet. Für die benötigten Eingabebilder wurde die Klasse *Texture* als Interface für die LIC-Klassen implementiert. Die Klasse *WhiteNoise* liefert nur ein weißes Rauschen, während die Klasse *ColorNoise* für Rot, Grün und Blau unterschiedliche Werte liefert. Die Klasse *LicViewer* wird nun als Pre-Viewer für das erzeugte Volumen und zur Speicherung in eine PVM-Datei benutzt, die dann zum Beispiel von einem Raycaster angezeigt werden kann.

4.2 Optimierungen

Wie bereits in der Klassenhierarchie (siehe Abbildung 33) zu sehen ist, läßt sich diese Implementierung recht einfach durch Ableiten bestehender Klassen verbessern. Dabei wurden die meisten der von Hege und Stalling [HS98] Beschleunigungen, beziehungsweise die meisten der von Interrante und Grosch [IG97] Verbesserungen implementiert.

4.2.1 Beschleunigung

Die größte Beschleunigung ließ sich durch die Verwendung der schnellen Berechnung der Faltung aus dem Algorithmus von Hege und Stalling [HS98] erreichen, wobei hier eine etwas abgewandelte Implementation verwendet wird. Im Gegensatz zum ursprünglichen Algorithmus können auch Punkte mehrfach berechnet werden, falls der berechnete Wert sehr weit vom Mittelpunkt des Voxels abweicht. Bei dieser Mehrfachberechnung wird dann ein gewichtetes Mittel gebildet.

Die ursprüngliche Codesequenz aus *SimpleLIC*:

```
für alle Voxel V im Ausgabe-Volumen
  Feldlinie = Integrator->berechneFeldlinie(V, Länge)
  Wert      = 0
  für alle n Punkte P der Feldlinie
    Wert = Wert + Texture->Wert(P) *
           WeightFunction->berechneWert(P, Länge/n)
  Wert(V) = Wert
```

Wird nun durch eine modifizierte Codesequenz mit dem Parametern *Mindestgewicht*, also wie sicher der berechnete Wert sein muß und dem Parameter *MindestSamples*, also wie viele Werte maximal für ein Voxel berechnet werden, ersetzt. Wobei ein weiterer Wert für ein Voxel nur dann berechnet wird, falls beide

Parameter unterschritten werden. Diese Implementierung findet sich sowohl in *FastLIC* als auch in *DoubleLIC* wieder.

```
// Initialisierung
für alle Voxel V im Ausgabe-Volumen
    Wert(V) = 0
    Gewicht(V) = 0
    Samples(V) = 0

// Neuen Voxel suchen
für alle Voxel V im Ausgabe-Volumen
    wenn Samples(V) < Mindestsamples
        // Feldlinie und Integrale berechnen
        Feldlinie = Integrator->berechneFeldlinie(V, 4*Länge)
        für alle Punkte P der Feldlinie
            Feldlinie->Wert(P) = Textur->Wert(P)
        für i von 1 bis WeightFunction->Ordnung
            Summe = 0
            für alle Punkte P der Feldlinie
                Summe = Summe + Feldlinie->Wert(P)
            Feldlinie->Wert(P) = Summe

// Faltung für Punkte auf der Feldlinie berechnen
Wert = 0
n=Feldlinie->AnzahlPunkte
für i von n/8 bis 7*n/8
    NV = berechneVoxel(Feldlinie->Punkt(i))
    Gewicht = berechneGewicht(Feldlinie->Punkt(i))
    wenn Gewicht(NV) > Mindestgewicht
        für j von 1 WeightFunction->Sprünge
            Wert = Wert +
                Feldlinie(WeightFunction->Sprung(j)*Länge+i) *
                WeightFunction->Sprungwert(j)
        Wert(V) = Wert(V) + Wert * Gewicht
        Gewicht(V) = Gewicht(V) + Gewicht
        Samples(V) = Samples(V) + 1

// Werte normalisieren
für alle Voxel V im Ausgabe-Volumen
    Wert(V) = Wert(V) / Gewicht(V)
```

Diese Optimierung kann genau wie das Verfahren von Hege und Stalling [HS98] durch eine pseudozufällige Wahl des nächsten Voxels weiter beschleunigt werden.

4.2.2 Verdeckungsproblem

Wie bereits angesprochen ist die Verdeckung ein zentrales Problem bei der dreidimensionalen Linienintegralfaltung. Interrante und Grosch [IG97] haben zwar einige Verfahren, um den dreidimensionalen Eindruck zu verbessern, aber sie erhöhen dabei zum Beispiel durch Halo's den Grad der Verdeckung.

Die einzige Möglichkeit die Verdeckung effektiv zu reduzieren ist eine Einschränkung des Eingabebildes im Bezug auf seine Opazität. In dieser Implementierung wurden sowohl bei *WhiteNoise* als auch bei *ColorNoise* eine direkte Nachbarschaft zwischen zwei opaquen Voxeln verboten. Da höchstens jedes achte Voxel, im Durchschnitt jedes Sechzehnte, opaque sein kann, wird die Menge an opaquen Voxeln und damit die Verdeckung im Ausgabebild deutlich reduziert.

4.3 GUI Einbindung

Die Gui Einbindung erfolgt über die Pre-Viewer Klasse *LicViewer*, der die Qt-Bibliotheken verwendet. Dabei werden die, durch den Benutzer ausgewählten Klassen erzeugt und in der entsprechenden LIC-Klasse eingetragen. Die Einstellungen erfolgen dabei über ein Interface, das in der Klasse *LicOptions* definiert wird.

5 Implementierung Particle Splatting

Die Implementierung der Visualisierung mit Particle Splatting erfolgte in C++. Der verwendete Viewer wurde ebenfalls in C++ unter der Verwendung der Qt-Bibliotheken von Troll-Tech und der OpenGL-Bibliotheken geschrieben. Die Vectorfelder können entweder als reguläres Gitter oder als CDF Datensatz eingelesen werden.

5.1 Klassenhierarchie

Die Implementierung erfolgt mit vier Klassen, die zur Implementation unterschiedlicher Verfahren weiter abgeleitet wurden. Basisklassen ohne Funktionalität sind dabei in Abbildung 34 durch eine gestrichelte Umrandung gekennzeichnet.

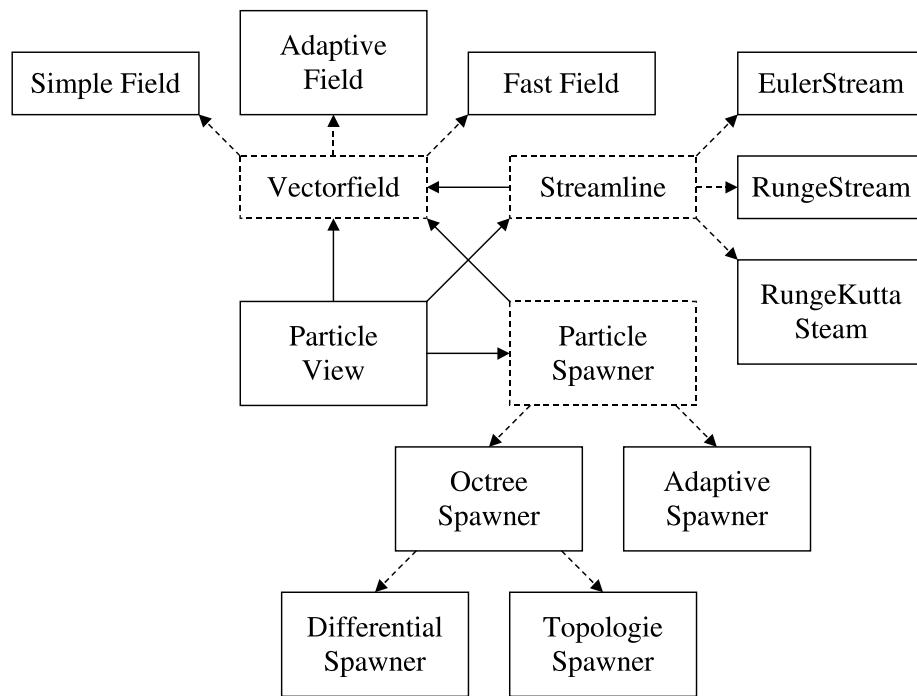


Abbildung 34: Klassenhierarchie der Visualisierung dreidimensionaler Vectorfelder mittels Particle Splatting.

Die zentrale Klasse ist bei dieser Implementierung die *ParticleView* Klasse, die sowohl die Darstellung durch OpenGL, als auch die Verwaltung der einzelnen Splats beinhaltet. Das Vektorfeld befindet sich in der Klasse *Vectorfield*. Davon abgeleitet werden *SimpleField*, ein reguläres Gitter auf dem trilinear interpoliert wird, *AdaptiveField*, ein Octree mit unterschiedlichen Tiefen und wiederum trilinearere Interpolation und *FastField*, ein Octree mit unterschiedlichen Tiefen und ohne Interpolation. Durch die Klasse *ParticleSpawner* wird ein Interface definiert, damit *ParticleView* die Partikel verteilen und auch wieder im *ParticleSpawner* eintragen kann. Von dieser Klasse gibt es zwei unterschiedliche

Arten der Implementierung, der *OctreeSpawner* und die von ihm abgeleiteten Klassen, verteilt die Partikel mit Hilfe eines Octrees gleicher Tiefe. Die Verteilung bei *OctreeSpawner* ist dabei gleichmäßig über das ganze Vektorfeld, die Verteilung bei *DifferentialSpawner* hängt von der lokalen Divergenz des Vektorfeldes ab und die Verteilung bei *TopologieSpawner* hängt mit dem Abstand von kritischen Punkten oder Bereichen zusammen. Der *AdaptiveSpawner* benutzt eine Gewichtsfunktion, die, je nach Wahl der dazugehörigen Parameter, von der Länge des Vektors, der lokalen Divergenz, der lokalen Rotation oder einem weiteren Parameter des Vektorfeldes abhängt. Beim *AdaptiveSpawner* wird dabei, im Gegensatz zum *OctreeSpawner*, ein Octree mit unterschiedlicher Tiefe benutzt, sodaß in jedem Blatt ein bis acht Partikel unterzubringen sind. Die Berechnung einer Feldlinie mittels einer von *Streamline* abgeleiteten Klasse wird erst zur Animation der Partikel und damit wieder nur vom *ParticleViewer* benötigt. Von dieser Klasse wurden drei Implementierungen realisiert. *EulerStream* ist ein einfacher Euler Integrator, während *RungeStream* ein Runge Verfahren zweiter Ordnung und *RungeKuttaStream* ein Runge Kutta Verfahren vierter Ordnung implementieren.

5.2 Optimierungen

Die meisten möglichen Optimierungen führen entweder zu einer anderen Darstellung des Vektorfeldes, oder einer Vorberechnung häufig gebrauchter Werte im Vektorfeld, da die am meisten benutzte Funktion den Vektor an einer beliebigen Stelle des Vektorfeldes liefert. Eine andere Art der Optimierung ist jedoch noch im *AdaptiveSpawner* möglich, da hier immer wieder neuer Speicher reserviert und freigegeben wird.

5.2.1 Beschleunigung

Da die meisten Vektorfelder als curvilineare oder unregelmäßige Gitter vorliegen, müssen sie in eine andere Repräsentation übergeführt werden, um einen schnellen Zugriff auf einen Vektor an einer beliebigen Stelle des Vektorfeldes zu ermöglichen. Eine Repräsentation durch ein reguläres Gitter scheidet aus, da sonst eine viel zu große Auflösung benötigt wird. Daher wird das Vektorfeld in einem Octree gespeichert, der in jeder Zelle die Vektoren der acht Eckpunkte $v_0 \dots v_7$ gespeichert hat.

Eine weitere Beschleunigung ist durch eine andere Darstellung der trilinearen Interpolation möglich. Mit den relativen Koordinaten (x_r, y_r, z_r) eines Vektors (x, y, z) in der Zell (x_0, y_0, z_0) bis (x_1, y_1, z_1) :

$$\begin{aligned} x_r &= (x - x_0)/(x_1 - x_0) \\ y_r &= (y - y_0)/(y_1 - y_0) \\ z_r &= (z - z_0)/(z_1 - z_0) \end{aligned} \tag{53}$$

ergibt sich dann für die trilineare Interpolation:

$$\begin{aligned} a_0 &= v_0 + x_r(v_1 - v_0) \\ a_1 &= v_2 + x_r(v_3 - v_2) \end{aligned} \tag{54}$$

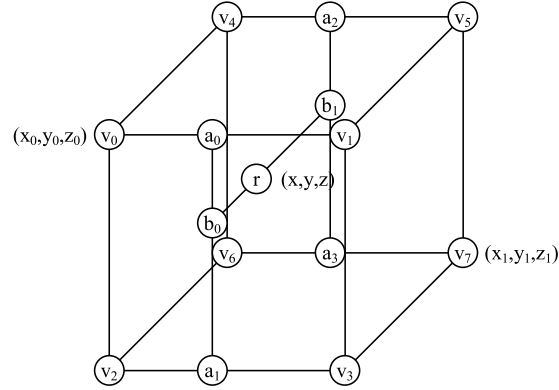


Abbildung 35: Trilineare Interpolation ohne Optimierungen.

$$\begin{aligned}
 a_2 &= v_4 + x_r(v_5 - v_4) \\
 a_3 &= v_6 + x_r(v_7 - v_6) \\
 b_0 &= a_0 + y_r(a_1 - a_0) \\
 b_1 &= a_2 + y_r(a_3 - a_2) \\
 r &= b_0 + z_r(b_1 - b_0)
 \end{aligned}$$

Durch eine Vorausberechnung der Vektoren u_0 bis u_7 wird die Trilineare Interpolation reduziert zu:

$$\begin{aligned}
 c_0 &:= u_0 + xu_1 \\
 c_1 &:= u_2 + xu_3 \\
 c_2 &:= u_4 + xu_5 \\
 c_3 &:= u_6 + xu_7 \\
 d_0 &:= c_0 + yc_1 \\
 d_1 &:= c_2 + yc_3 \\
 r &:= d_0 + zd_1
 \end{aligned} \tag{55}$$

Dadurch entfallen die drei Divisionen und sechs Additionen bei der Berechnung der relativen Position und weitere sieben Additionen bei der Interpolation selbst.

Eine andere Art der Beschleunigung bei der Octree Repräsentation ist die Verwendung nur eines einzigen Vektors für eine Zelle. Die Anzahl der Zellen steigt zwar bei der Beibehaltung des selben Fehlers, aber die Geschwindigkeit wird trotz der größeren Tiefe gesteigert.

Die zweite Optimierung wird für die Animation der Partikel benötigt, da der adaptive Octree seine Tiefe auf Grund der Gewichtsfunktion ändern kann und sich während der Animation auch die Anzahl der Partikel ändert. Der benötigte Speicher wird nicht jedesmal einzeln reserviert, sondern wie von Meyers [Mey92] vorgeschlagen in einer Liste mit freien und reservierten Elementen verwaltet. Wird nun ein neues Element benötigt und es ist kein weiteres verfügbar, wird gleich eine größere Anzahl Elemente reserviert. Wird ein Element frei gegeben, wird es nur in die Liste der freien Elemente eingetragen. Dadurch werden viele

Aufrufe zur Reservierung und Freigabe von Speicher gespart und sowohl die Rechenzeit, als auch der Overhead bei der Speicherreservierung wird reduziert.

5.2.2 Verdeckungsproblem

Die Verdeckung wird durch zum Teil transparente Partikel, die nur an interessanten Stellen platziert werden, möglichst weit reduziert. Die Darstellung durch Splatting ermöglicht dabei eine interaktive Untersuchung des Vektorfeldes. Dadurch und durch die Möglichkeit die Transparenz der Partikel auf die aktuellen Gegebenheiten anzupassen, wird das Problem der Verdeckung fast komplett gelöst. Jedoch bleibt dabei zu beachten, daß die dichte Repräsentation nur durch entsprechend gewählte Parameter der Gewichtsfunktion gewährleistet werden kann.

5.2.3 Animation

Wie bereits erwähnt ist die Animation eine gute Möglichkeit, durch wechselnde Verdeckung, einen tieferen Einblick in das Vektorfeld zu gewinnen. Während der Animation werden die Partikel entlang des Vektorfeldes um den Wert weiterbewegt, der von der *Streamline* Klasse berechnet wird. Die Berechnung der Feldlinie läuft dabei über eine adaptive Schrittweite bei festem Fehler.

```
l = 0
s = Länge
Ziel = Start
so lange l < Länge
  f = berechneSchritt(Ziel,z,s)
  wenn f < Fehler
    l = l + s
    s = s * 2
  wenn l + s > Länge
    s = Länge - s
    Ziel = Ziel + z
  sonst
    s = s / 2
```

Dabei wird mit *berechneSchritt(p, s, l)* von der Position *p* aus ein Schritt *s* mit Länge *l* berechnet. Da bei diesem Verfahren sowohl der Vektor am Startpunkt als auch der Vektor am Endpunkt eines solchen Schrittes mehrmals benötigt werden kann, wird er zwischengespeichert, wodurch die Anzahl der Aufrufe zur Berechnung eines Vektors deutlich reduziert werden. Eine weitere Optimierung während der Animation bezieht sich auf die Bewegung der Partikel im Octree zur Partikelverteilung. Da die Partikel oft nur von einer Zelle in ihre Nachbarzelle wandern, wird bei der Verschiebung der Partikel nicht ganz aus dem Octree entfernt und wieder eingefügt, sondern nur bis zu der Höhe auf der er in einen anderen Teil des Octrees übertragen wird.

5.3 GUI Einbindung

Die Gui Einbindung erfolgt über die Viewer Klasse *ParticleViewer*, der die Qt-Bibliotheken verwendet. Dabei werden die, durch den Benutzer ausgewählten

Klassen erzeugt und im Viewer eingetragen. Die Einstellungen erfolgen dabei über ein Interface, das in der Klasse *ParticleOptions* definiert wird. Die Qt-Bibliotheken sind dabei auch für den Timer zuständig, der die Animation steuert.

Der Timer der Qt-Bibliotheken ist eigentlich kein Echtzeit Timer, da immer nur dann ein Timer Ereignis ausgelöst wird, wenn Qt selbst nicht beschäftigt ist. Dadurch ist es auch nicht möglich, daß das Zeichnen oder gar die Berechnung neuer Partikel unterbrochen wird. Da die erzielten Bildwiederholraten aber meistens im Bereich oberhalb von 20 Bildern pro Sekunde liegen und diese Frequenz bereits zur interaktiven Untersuchung ausreicht, wurde der Timer auf eine Rate von 20 Ereignissen pro Sekunde festgelegt. Dadurch ist eine feste Geschwindigkeit der Animation und damit auch eine Vergleichbarkeit der Geschwindigkeiten an unterschiedlichen Stellen des Vektorfeldes möglich.

Diese hohen Bildwiederholraten konnten jedoch nur durch den Einsatz der Matrixmultiplikationen unter OpenGL ermöglicht werden und so ist dieses Verfahren auch auf die Unterstützung dieser Multiplikationen in Hardware angewiesen. Da OpenGL nur mit einem Z-Buffer arbeitet, müssen die Partikel vor der Darstellung noch mittels einer Hash Tabelle [Sed92], um so wenig wie möglich Zeit in Anspruch zu nehmen, sortiert werden.

6 Ergebnisse

Um die beiden Algorithmen zu vergleichen wurden einige unterschiedliche Datensätze und Repräsentationen ausgewählt. Ein Datensatz der als reguläres Gitter vorliegt: Kern33rel (32^3 Vektoren). Die ursprüngliche Repräsentation dieses Vektorfeldes war eine Formel. Drei CDF Datensätze als curvilineare Gitter: BluntFin ($40 \times 32 \times 32$ Vektoren), OxygenPost ($38 \times 76 \times 38$ Vektoren) und DeltaWing ($56 \times 54 \times 70$ Vektoren). Die Berechnungen wurden auf einem PC mit AMD-K7 Prozessor (650 MHz) durchgeführt und die Darstellung erfolgte unter OpenGL mittels GeForce256 Chipsatzes, da Matrixmultiplikationen von diesem Chipsatz in Hardware unterstützt werden. Dabei wurden sowohl die Qualität der erzeugten Bilder und Animationen als auch die benötigte Rechenzeit verglichen.

6.1 3D LIC

Da bei der dreidimensionalen Linienintegrale eine feste Auflösung vorgegeben wird, wurden die CDF Datensätze in ein reguläres Gitter mit 128^3 Vektoren umgewandelt. Wie man in Tabelle 2 erkennen kann, wird die Rechenzeit durch

Datensatz	Algorithmus	Filter	Auflösung	Zeit in Sekunden
Kern33rel	SimpleLIC	Box	64^3	155
Kern33rel	FastLIC	Box	64^3	31

Tabelle 2: Vergleich der Rechenzeiten zwischen SimpleLIC und FastLIC am Beispiel des Datensatzes Kern33rel.

die Verwendung des FastLIC Algorithmus deutlich reduziert, ohne daß man in den Abbildungen 36 und 37 einen größeren Unterschied oder eine Zunahme von Artefakten erkennen kann. Dabei wurden für 262.144 Voxel 8.451 Feldlinien von einer Länge von 540 möglichen Sampelpunkten erzeugt. Von diesen 4.563.540 Samplepunkten wurden 2.376.099 Punkte ausgewertet. Die restlichen Samplepunkte wurden verworfen, da sie zu weit vom Zentrum eines Voxels entfernt und damit zu ungenau waren.

Datensatz	Algorithmus	Filter	Auflösung	Zeit in Sekunden
Kern33rel	FastLIC	Triangle	64^3	32
Kern33rel	DoubleLIC	Box	64^3	44
Blunt Fin	FastLIC	Triangle	128^3	72
Blunt Fin	DoubleLIC	Box	128^3	315
Oxygen Post	FastLIC	Triangle	128^3	155
Oxygen Post	DoubleLIC	Box	128^3	1214
Delta Wing	FastLIC	Triangle	128^3	157
Delta Wing	DoubleLIC	Box	128^3	1376

Tabelle 3: Vergleich der Rechenzeiten zwischen FastLIC und DoubleLIC bei jeweils gleichwertigem Filter.

Die starken Aliaseffekte in den Abbildungen 36 und 37 lassen sich entweder durch einen Filter höherer Ordnung, in diesem Fall ein Dreiecksfilter oder durch eine zweifache Linienintegrale mit einem Boxfilter reduzieren. Zwar sind

nur noch wenige Artefakte in den Abbildungen 38, 40, 42 und 44 zu erkennen aber durch die doppelte Faltung können sie, wie in den Abbildungen 39, 41, 43 und 45 zu sehen, noch weiter reduziert werden. Die zusätzliche benötigte Rechenzeit (siehe Tabelle 3) steigt dabei jedoch nur kaum, es sein denn ein Teil des ersten gefalteten Bildes muß ausgelagert werden, wie bei den Datensätzen Blunt Fin, Oxygen Post und Delta Wing.

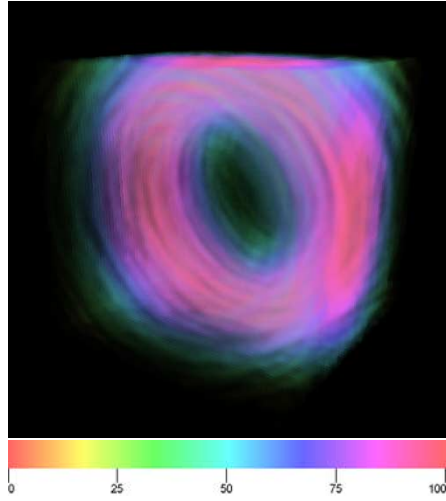


Abbildung 36: SimpleLic mit BoxFilter bei Kern33rel (155 Sekunden).

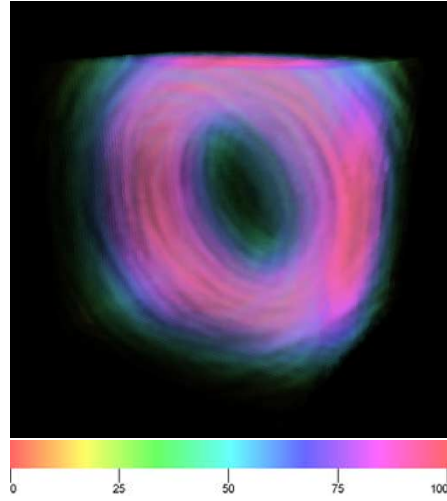


Abbildung 37: FastLic mit BoxFilter bei Kern33rel (31 Sekunden).

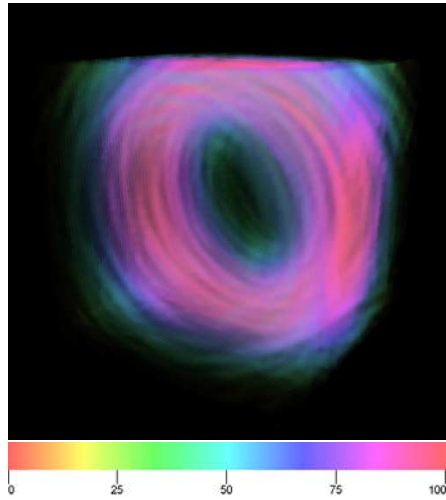


Abbildung 38: FastLic mit Triangle-Filter bei Kern33rel (32 Sekunden).

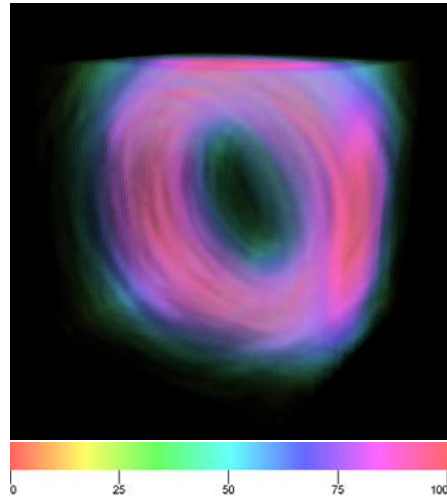


Abbildung 39: DoubleLic mit BoxFilter bei Kern33rel (44 Sekunden).

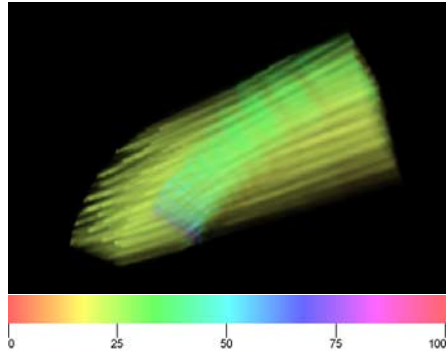


Abbildung 40: FastLic mit Triangle-Filter bei Blunt Fin (72 Sekunden).

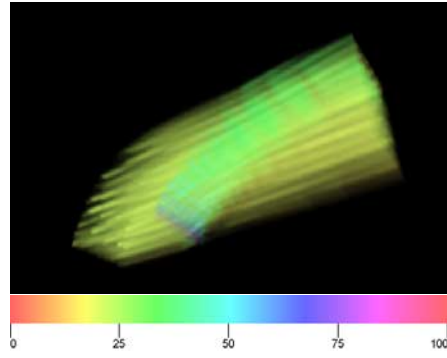


Abbildung 41: DoubleLic mit BoxFilter bei Blunt Fin (315 Sekunden).

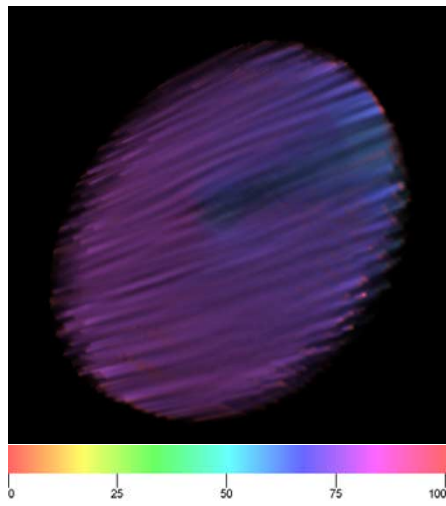


Abbildung 42: FastLic mit Triangle-Filter bei Oxygen Post (155 Sekunden).

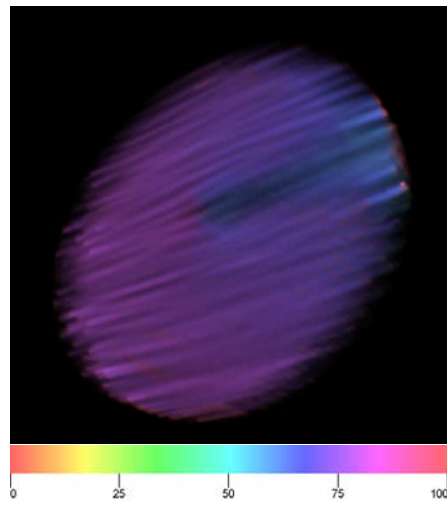


Abbildung 43: DoubleLic mit BoxFilter bei Oxygen Post (1214 Sekunden).

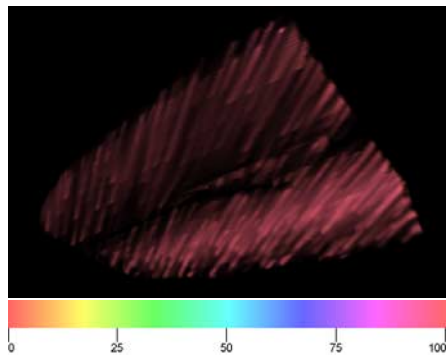


Abbildung 44: FastLic mit Triangle-Filter bei Delta Wing (157 Sekunden).

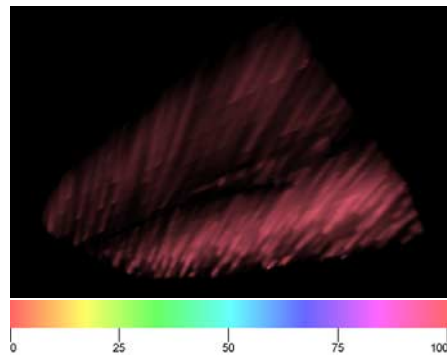


Abbildung 45: DoubleLic mit BoxFilter bei Delta Wing (1376 Sekunden).

6.2 Particle Splatting

Im Gegensatz zur festen Auflösung bei der dreidimensionalen Linienintegralfaltung wird beim Partiklesplatting ein möglichst schneller Zugriff auf einen Vektor an einer beliebigen Stelle des Vektorfeldes benötigt. Um eine möglichst genau Repräsentation bei geringer Datenmenge und damit eine große Anzahl an Zugriffen auf einen kleinen Speicherbereich zu bekommen, werden die bereits erklärten Octrees benutzt und mit der regulären Repräsentation verglichen.

Datensatz	Repräsentation	Integrator	Bilder pro Sekunde
Kern33rel	Simple	Euler (1%)	5,419
Kern33rel	Adaptive	Euler (1%)	7,081
Kern33rel	Fast	Euler (1%)	7,266
Blunt Fin	Adaptive	Euler (1%)	6,433
Blunt Fin	Fast	Euler (1%)	5,525

Tabelle 4: Vergleich der Bildwiederwohraten mit 8000 Partikeln bei verschiedenen Repräsentationen.

Am Beispiel des Kern33rel Datensatzes kann man, wie in Tabelle 4 zu erkennen ist, den Geschwindigkeitszuwachs besonders gut erkennen. Während die Zahl der Bilder pro Sekunde von der regulären Repräsentation zur Repräsentation mittels des Octrees mit trilinearere Interpolation wegen der reduzierten Datenmenge und der optimierten Interpolation deutlich zunimmt, ist nur noch eine geringe Leistungssteigerung beim Octree ohne Interpolation zu erkennen, da die Datenmenge zugenommen hat. Beim Blunt Fin Datensatz ist diese Zunahme dabei so groß, daß, trotz der gesparten trilinearem Interpolation, sogar weniger Bilder pro Sekunde dargestellt werden können. Dabei ist zu beachten, daß der Integrator an der Grenze zweier Zellen beim Octree ohne Interpolation wegen der Unstetigkeit eine größere Anzahl an Schritten benötigt, als beim Octree mit trilinearere Interpolation.

Datensatz	Repräsentation	Integrator	Bilder pro Sekunde
Blunt Fin	Adaptive	Euler (1%)	6,433
Blunt Fin	Adaptive	Runge (1%)	5,358
Blunt Fin	Adaptive	Runge-Kutta (1%)	4,756

Tabelle 5: Vergleich der Bildwiederwohraten mit 8000 Partikeln bei verschiedenen Integratoren.

Obwohl die Integratoren höherer Ordnung für einen Schritt deutlich mehr Vektoren ermitteln müssen um so einen Fehler höherer Ordnung zu erreichen, steigt die benötigte Rechenzeit bei gleichem Fehler wesentlich geringer. So werden beim Runge Verfahren doppelt so viele Vektoren benötigt wie beim Euler Verfahren aber die Rechenzeit steigt nur um 20%. Beim Wechsel vom Runge zum Runge-Kutta Verfahren, wobei wieder die doppelte Anzahl an Vektoren benötigt wird, steigt die Rechenzeit nur um weitere 12,7%. Dabei ist zu bemerken, daß die Fehler zwar vom Absolutwert gleich sind, aber von höherer Ordnung.

Auf den Abbildungen 46 bis 53 sind die vier Datensätze Kern33rel, Blunt Fin, Oxygen Post und Delta Wing als Übersicht über das Vektorfeld, bezie-

hungsweise als Vergrößerung eines interessanten Bereichs (siehe Abbildung 49, 51 und 53), mittels Particle Splatting dargestellt. Die Verteilung der Partikel erfolgte dabei über den adaptiven Octree mit der Gewichtung 90% Divergenz und 10% Konstant für die Abbildungen 46 und 47, beziehungsweise 90% Rotation und 10% Konstant für die Abbildungen 47 bis 53.

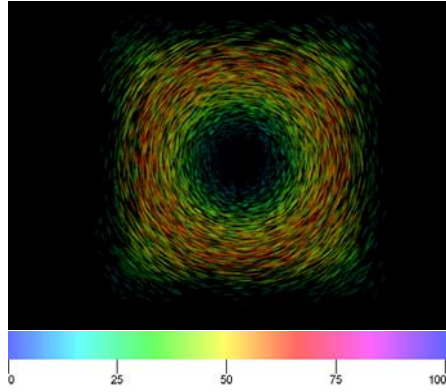


Abbildung 46: Kern33rel, 8000 Partikel Gesamtansicht.

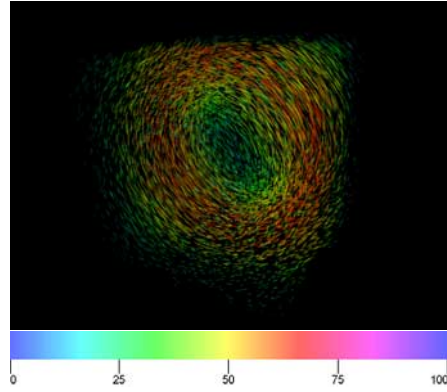


Abbildung 47: Kern33rel, 8000 Partikel Gesamtansicht.

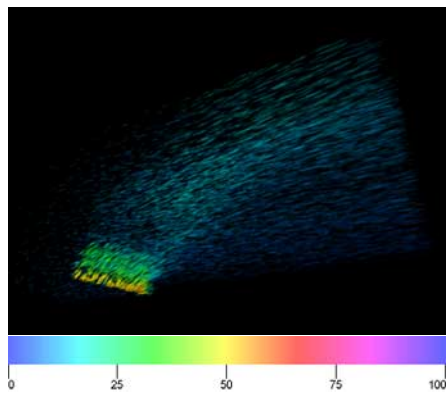


Abbildung 48: Blunt Fin, 8000 Partikel Gesamtansicht.

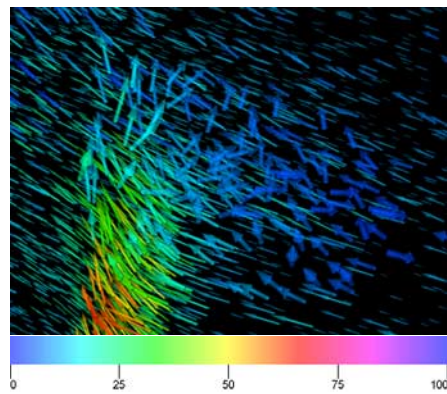


Abbildung 49: Blunt Fin, 8000 Partikel Vergrößerung eines Wirbels.

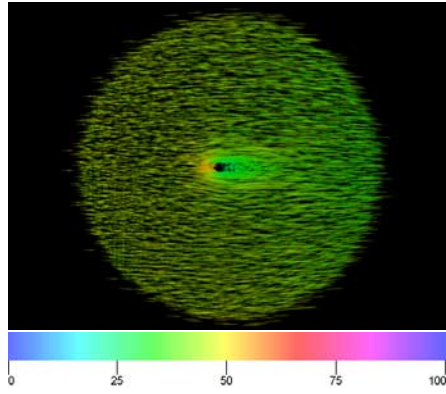


Abbildung 50: Oxygen Post, 8000 Partikel Gesamtansicht.

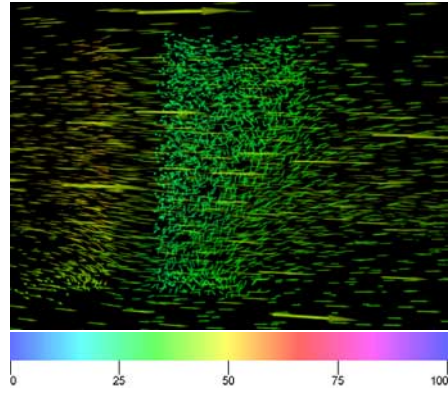


Abbildung 51: Oxygen Post, 8000 Partikel Vergrößerung der Verwirbelung.

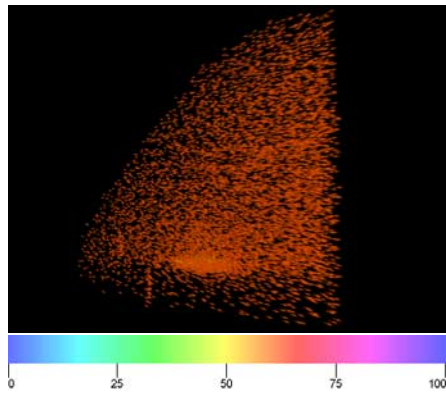


Abbildung 52: Delta Wing, 8000 Partikel Gesamtansicht.

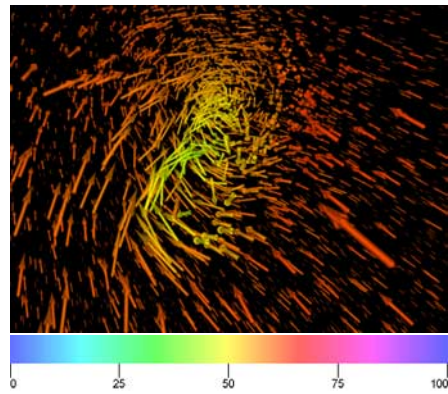


Abbildung 53: Delta Wing, 8000 Partikel Vergrößerung der Verwirbelung.

Literatur

- [Ban94] David C. Banks. Illumination in diverse codimensions. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 327–334. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.
- [Bat67] G. K. Batchelor. An introduction to fluid dynamics. University Press, Cambridge, 1967.
- [BF88] Paul Bratley and Bennett L. Fox. Algorithm 659: Implementing Sobol’s quasirandom sequence generator. *ACM Transactions on Mathematical Software*, 14(1):88–100, March 1988.
- [Blo90] J. Bloomenthal. Calculation of reference frames along a space curve. In A. Glassner, editor, *Graphics Gems*. Academic Press, San Diego, 1990.
- [CL93] Brian Cabral and Leith (Casey) Leedom. Imaging vector fields using line integral convolution. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 263–272, August 1993.
- [CM93] R. A. Crawfis and N. Max. Texture splats for 3D scalar and vector field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 261–267, San Jose, CA, October 1993. IEEE Computer Society Press.
- [DC90] Debra Dooley and Michael F. Cohen. Automatic illustration of 3D geometric models: Lines. In *Proceedings of the Symposium on Interactive 3D Graphics*, volume 24_2 of *Computer Graphics*, pages 77–82, New York, NY, USA, March 1990. ACM Press.
- [dLvW93] W. C. de Leeuw and J. J. van Wijk. A probe for local flow field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 39–45, San Jose, CA, October 1993. IEEE Computer Society Press.
- [Frü96] Thomas Frühauf. Raycasting vector fields. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the Conference on Visualization*, pages 115–120, Los Alamitos, October 27–November 1 1996. IEEE.
- [Gay83] David Gay. Algorithm 611. *Collected algorithms of the ACM, also in ACM Transaction on Mathematical Software*, 9(4):503–524, 1983.
- [HS98] Hans-Christian Hege and Detlev Stalling. Fast LIC with piecewise polynomial filter kernels. In H.-C. Hege and K. Polthier, editors, *Mathematical Visualization - Algorithms and Applications*, pages 295–314. Springer, 1998.
- [Hul92] J. P. M. Hultquist. Constructing stream surfaces in steady 3d vector fields. *Proceedings Visualization '92, Boston*, pages 171–178, 1992.

- [IG97] Victoria Interrante and Chester Grosch. Strategies for effectively visualizing 3D flow with volume LIC (color plate S. 568). In Roni Yagel and Hans Hagen, editors, *Proceedings of the 8th Annual IEEE Conference on Visualization (VISU-97)*, pages 421–424, Los Alamitos, October 19–24 1997. IEEE Computer Society Press.
- [Leo79] B. P. Leonard. A stable and accurate convective modelling procedure based on quadratic upstream interpolation. *Computer Methods in Applied Mechanics and Engineering*, 19:59–98, 1979.
- [Mey92] Scott (Scott Douglas) Meyers. *Effective C++: 50 specific ways to improve your programs and designs*. Addison-Wesley, Reading, MA, USA, 1992.
- [Pho75] Bui-Tuong Phong. Illumination for computer generated pictures. *CACM June 1975*, 18(6):311–317, 1975.
- [Sab88] Paolo Sabella. A rendering algorithm for visualizing 3D scalar fields. volume 22, pages 51–58, August 1988.
- [Sed92] Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, Reading, MA, USA, 1992.
- [SH95] Detlev Stalling and Hans-Christian Hege. Fast and resolution independent line integral convolution. In Robert Cook, editor, *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pages 249–256. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.
- [Sob67] I. M. Sobol'. The distribution of points in a cube and the approximate evaluation of integrals. *USSR Comp. Math. Math. Phys.*, 7(4):86–112, 1967.
- [vW93a] J. J. van Wijk. Implicit stream surfaces. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 245–252, San Jose, CA, October 1993. IEEE Computer Society Press.
- [vW93b] Jarke J. van Wijk. Flow visualization with surface particles. *IEEE Computer Graphics and Applications*, 13(4):18–24, July 1993.
- [Wes91] Lee Alan Westover. SPLATTING: A parallel, feed-forward volume rendering algorithm. Technical Report TR91-029, University of North Carolina, Chapel Hill, July 1, 1991.
- [ZSH96] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive visualization of 3D-vector fields using illuminated streamlines. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the Conference on Visualization*, pages 107–114, Los Alamitos, October 27–November 1 1996. IEEE.

A Methoden der 3D LIC Klassen

Da die Klassen innerhalb der dreidimensionalen Linienintegralfaltung funktional stark von einander getrennt sind, wurden sie in unterschiedliche Bibliotheken aufgeteilt.

Vectorfield	Integrator	Texture	WeightFunction	LIC
Vectorfield	Integrator	Texture	WeightFunction	SimpleLIC
SimpleField		WhiteNoise	BoxWeight	FastLIC
AdaptiveField		ColorNoise	TriangleWeight	DoubleLIC
FastField			PolyWeight	

Tabelle 6: LIC Bibliothek und darin enthaltene Klassen.

Die Attribute und Methoden dieser Klassen werden in den nachfolgendem Abschnitten genauer erläutert. Da die in den einzelnen Bibliotheken enthaltenen Klassen immer von einer einzigen Basisklasse abgeleitet sind, brauchen nur die Attribute und Methoden der Basisklasse erläutert werden.

A.1 Vectorfield

<i>Vectorfield</i> Klasse zur Repräsentation eines Vektorfeldes	
Attribute	Methoden
<i>minStepSize</i>	<i>loadField</i>
<i>secondFunction</i>	<i>putVector</i>
	<i>setValueFunction</i>
	<i>setValueFunction2</i>
	<i>getValue</i>
	<i>getValue2</i>
	<i>setFunction2</i>
	<i>getMinValue</i>
	<i>getMinValue2</i>
	<i>getMaxValue</i>
	<i>getMaxValue2</i>
	<i>setMinStepSize</i>
	<i>getMinStepSize</i>
	<i>getMaxStepSize</i>
	<i>putStep</i>
	<i>putBack</i>

Tabelle 7: Attribute und Methoden von Vectorfield

Vectorfield ist die Basisklasse für alle Repräsentationen eines Datensatzes mit den dazugehörigen Funktionen. Der Datensatz wird mit der Funktion *loadField* geladen und in die jeweilige Art der Repräsentation gebracht. Die Funktionen *putVector*, *getValue* und *getValue2* liefern den Vektor und die beiden weiteren möglichen Parameter an einem bestimmten Punkt im Datensatz. Mit dem Wert von *getValue*, der über die Funktion *setValueFunction* gewählt werden kann, wird dabei die Opazität und mit dem Wert von *getValue2*, der über

die Funktion *setValueFunction* gewählt werden kann, der Farbton moduliert. Außerdem kann der zweite Parameter durch *setFunction2* ein, beziehungsweise aus geschaltet werden. Die zur Normierung benötigten minimalen und maximalen Werte der beiden Parameter lassen sich durch die Funktionen *getMinValue*, *getMinValue2*, *getMaxValue* und *getMaxValue2* abfragen. Mit der Funktion *setMinStepsize* kann man eine untere Grenze für die Schrittweiten bei der Integration angeben. Mit den Funktionen *getMinStepSize* und *getMaxStepSize* kann man nun diese untere, beziehungsweise obere Grenze der Schrittweite abfragen. Die letzten beiden Funktionen *putStep* und *putBack* berechnen einen Integrationsschritt in Richtung, beziehungsweise gegen die Richtung des Vektorfeldes mit einem angegebenen maximalen Fehler.

A.2 Integrator

<i>Integrator</i>	
Klasse zur Berechnung einer Feldlinie	
Attribute	Methoden
<i>flowLine</i>	<i>setVectorField</i>
<i>flowLength</i>	<i>getVectorField</i>
<i>oldLength</i>	<i>computeFlowLine</i>
<i>vectorField</i>	<i>putPosition</i>

Tabelle 8: Attribute und Methoden von Integrator

Mit den Funktionen *setVectorField* und *getVectorField* wird dem Integrator ein neues Vektorfeld zugewiesen, beziehungsweise das aktuelle Vektorfeld ermittelt. Nach der Berechnung einer Feldlinie der Länge *flowLength* mit der Funktion *computeFlowLine*, die dann in der Variable *flowLine* abgelegt wird, kann die Position an einer beliebigen Stelle dieser Feldlinie mittels der Funktion *putPosition* abgefragt werden. Die Variable *oldLength* ist dabei die Länge der längsten verwendeten Feldlinie und damit der momentan verfügbare Speicherbereich für eine Feldlinie.

A.3 Texture

<i>Texture</i>	
Klasse zur Repräsentation einer Eingabetextur	
Attribute	Methoden
	<i>putTexture</i>
	<i>putIntegratedTexture</i>
	<i>getMinStep</i>

Tabelle 9: Attribute und Methoden von Texture

Die Funktion *putTexture* liefert den Texturwert an einer beliebigen Stelle der Textur, während die Funktion *putIntegratedTexture* den durchschnittlichen Texturwert entlang einer Linie in der Textur angibt. Die Funktion *getMinStep* liefert außerdem noch die minimale Schrittweite, die für diese Textur benötigt wird.

A.4 WeightFunction

<i>WeightFunction</i>	
Klasse zur Berechnung des Filterkerns	
Attribute	Methoden
<i>directionBoth</i>	<i>getWeight</i> <i>getIntegratedWeight</i> <i>setDirection</i> <i>setNextLevel</i> <i>putPeak</i>

Tabelle 10: Attribute und Methoden von WeightFunction

Mit Hilfe der Funktionen *getWeight* und *getIntegratedWeight* kann man den Wert an einer bestimmten Stelle, beziehungsweise das Integral über einen Bereich des Filterkerns abfragen. Mit der Funktion *setDirection* kann man zwischen einem symmetrischen Filterkern und einem berichteten Filterkern wählen. Dadurch kann man bestimmen, ob man auch die Richtung der Vektoren sehen kann. Die letzten beiden Funktionen *setNextLevel* und *putPeak* sind zur Optimierung durch das Fast LIC Verfahren. Dabei wählt *setNextLevel* die nächste Ableitung aus und *putPeak* liefert alle Spitzen in dieser Stufe.

A.5 SimpleLIC

<i>SimpleLIC</i>	
Klasse zur Berechnung der Linienintegralfaltung	
Attribute	Methoden
<i>vectorField</i>	<i>setVectorField</i>
<i>weightFunction</i>	<i>getVectorField</i>
<i>texture</i>	<i>setWeightFunction</i>
<i>integrator</i>	<i>getWeightFunction</i>
<i>quality</i>	<i>setTexture</i>
<i>pixelCount</i>	<i>getTexture</i>
<i>pixelWeight</i>	<i>setIntegrator</i>
<i>pixelColor</i>	<i>getIntegrator</i>
<i>flowColor</i>	<i>setQuality</i>
<i>flowPixel</i>	<i>getQuality</i>
<i>flowPoint</i>	<i>setOptions</i>
<i>flowWeight</i>	<i>getAbs</i>
<i>size</i>	<i>getRel</i>
<i>valueLenght</i>	<i>putAbsPoint</i>
<i>valueContrast</i>	<i>putRelPoint</i>
	<i>computeImage</i>
	<i>normalizeColors</i>

Tabelle 11: Attribute und Methoden von SimpleLIC

Mit den Funktionen *setVectorField*, *getVectorField*, *setWeightFunction*, *getWeightFunction*, *setTexture*, *getTexture*, *setIntegrator* und *getIntegrator* werden

Vektorfeld, Filterkern, Eingabetextur und Integrator in die LIC Klasse eingetragen beziehungsweise ausgelesen. Mit der Funktion *setQuality* wird der maximale Fehler eingetragen und kann mit *getQuality* abgefragt werden. Die Funktionen *getAbs*, *getRel*, *putAbsPoint* und *putRelPoint* dienen zur Konvertierung der normalisierten Koordinaten auf die Voxelkoordinaten und umgekehrt. Die Funktion *computeImage* berechnet nun das ganze LIC Volumen, das danach mit der Funktion *normalizeColors* auf den ganzen Farbbereich skaliert wird.

B Methoden der Particle Splatting Klassen

Da die Klassen innerhalb des Particle Splatting funktional stark von einander getrennt sind, wurden sie in unterschiedliche Bibliotheken aufgeteilt.

Vectorfield	Streamline	ParticleSpawner	Viewer
Vectorfield	Streamline	ParticleSpawner	ParticleView
SimpleField	EulerStream	OctreeSpawner	
AdaptiveField	RungeStream	DifferentialSpawner	
FastField	RungeKuttaStream	TopologieSpawner	
		AdaptiveSpawner	

Tabelle 12: Particle Splatting Bibliothek und darin enthaltene Klassen.

Die Attribute und Methoden dieser Klassen werden in den nachfolgendem Abschnitten genauer erläutert. Da die in den einzelnen Bibliotheken enthaltenen Klassen immer von einer einzigen Basisklasse abgeleitet sind, brauchen nur die Attribute und Methoden der Basisklasse erläutert werden.

B.1 Vectorfield

<i>Vectorfield</i>	
Klasse zur Repräsentation eines Vektorfeldes	
Attribute	Methoden
<i>minD</i>	<i>loadField</i>
<i>maxD</i>	<i>getDir</i>
<i>minE</i>	<i>getParams</i>
<i>maxE</i>	<i>getDif</i>
<i>minL</i>	<i>containsPoint</i>
<i>maxL</i>	<i>isLegal</i>
<i>attractingPoint</i>	
<i>repellingPoints</i>	
<i>zeroPoints</i>	

Tabelle 13: Attribute und Methoden von Vectorfield

Vectorfield ist die Basisklasse für alle Repräsentationen eines Datensatzes mit den dazugehörigen Funktionen. Der Datensatz wird mit der Funktion *loadField* geladen und in die jeweilige Art der Repräsentation gebracht. Die Funktionen *getDir* und *getParam* liefern den Vektor, den Vektor und die beiden weiteren Parameter oder nur die beiden Parameter an einem bestimmten Punkt im Datensatz. Die Vektoren und Parameter können dabei durch die Attribute *minL*, *maxL*, *minD*, *maxD*, *minE* und *maxE* normiert werden. Außerdem besteht die Möglichkeit die Ableitung des Vektorfeldes nach einer Achse über die Funktion *getDif* zu berechnen. Sowohl die Ableitung, als auch die kritischen Punkte, die nach der Berechnung mit *containsPoint* in *attractingPoints*, *repellingPoints* und *zeroPoints* abgelegt wurden, werden für die Verteilung der Partikel benötigt. Als letztes bleibt noch die Funktion *isLegal*, die zu einem Punkt angibt, ob er im Datensatz liegt oder nicht.

B.2 Streamline

<i>Klasse zur Berechnung der Feldlinien</i>	
Attribute	Methoden
<i>vectorField</i>	<i>computeStreamline</i>

Tabelle 14: Attribute und Methoden von Streamline

Mit der Funktion *computeStreamline* wird eine Feldlinie mit einer bestimmten Länge von einem gegebenen Punkt aus, für den in *vectorField* gespeicherten Datensatz, berechnet. Das Attribut *vectorField* wird im Konstruktor gesetzt.

B.3 ParticleSpawner

<i>Klasse zur Verteilung der Partikel im Datensatz</i>	
Attribute	Methoden
<i>vectorField</i>	<i>setMaxAge</i>
<i>particles</i>	<i>setSeed</i>
<i>numParticles</i>	<i>setDepth</i>
<i>maxAge</i>	<i>setNumParticles</i>
<i>dWeight</i>	<i>setWeights</i>
<i>eWeight</i>	<i>setBounds</i>
<i>divWeight</i>	<i>getBounds</i>
<i>rotWeight</i>	<i>insertParticle</i>
<i>addWeight</i>	<i>addParticle</i>
	<i>moveParticle</i>
	<i>removeParticle</i>
	<i>flushParticles</i>
	<i>getFirstParticle</i>
	<i>getNextParticle</i>

Tabelle 15: Attribute und Methoden von ParticleSpawner

Die Klasse ParticleSpawner ist die Basisklasse für alle Verteilungen von Partikeln und implementiert nur ein Interface für die ParticleView Klasse. Das Vektorfeld wird im Konstruktor gewählt und in *vectorField* gespeichert. Mit der Funktion *setMaxAge* wird das maximale Alter der Partikel bestimmt. Mit *setSeed* kann eine bestimmte Zufallsfolge gewählt werden. Die Funktion *setDepth* legt die minimale Tiefe des Octrees fest, der die mit *setNumParticles* bestimmte Anzahl an Partikeln aufnehmen kann. Die Gewichtsfunktion wird mit *setWeights* bestimmt, während die Ausmaße des Octrees mit *setBound* bestimmt, beziehungsweise mit *getBounds* abgefragt werden. Nach diesen Funktionen zur Beeinflussung des Octrees liefert *insertParticle* eine Position für einen neuen Partikel, der dann mit *addParticle* in den Octree eingefügt wird. Während der Animation wird er dann mit *moveParticle* innerhalb des Octrees bewegt und mit *removeParticle* entfernt. Die Funktion *flushParticles* übernimmt nun das Altern

der Partikel, während die Funktionen *getFirstParticle* und *getNextParticle* benutzt werden, um eine Reihenfolge der Partikel zu erzeugen, so daß Partikel innerhalb einer Zelle unmittelbar aufeinander folgen.

B.4 Viewer

<i>Klasse zur Verwaltung und zum Anzeigen der Partikel</i>	
Attribute	Methoden
<i>vectorField</i>	<i>loadTexture</i>
<i>streamLine</i>	<i>newField</i>
<i>particleSpawner</i>	<i>draw</i>
<i>particles</i>	<i>calcParticle</i>
<i>minBound</i>	<i>drawParticle</i>
<i>maxBound</i>	<i>setBounds</i>
	<i>doAnimation</i>
	<i>addParticle</i>
	<i>removeParticle</i>
	<i>removeParticles</i>

Tabelle 16: Attribute und Methoden von ParticleView

ParticleView wird zur Verwaltung und Darstellung der Partikel verwendet. mit *loadTexture* kann eine alternative Textur zur Darstellung der Partikel geladen werden. Mit *newField* wird ein neuer Datensatz geladen und das entsprechende Objekt vom Typ Vektorfeld erzeugt. Die Darstellung der Partikel erfolgt nun mit der Funktion *draw*, die die beiden Funktion *calcParticle* und *drawParticle* verwendet. *calcParticle* wird dabei zum Berechnen der Matrizen verwendet, während die Darstellung nach der Sortierung mit *drawParticle* geschieht. Die Animation wird in *doAnimation* berechnet, während dabei mit *addParticle* ein neuer Partikel erzeugt und in die Verwaltungsstrukturen eingefügt wird. Wird ein Partikel überflüssig, wird er mit *removeParticle* entfernt. Bei einem neuen Datensatz und bei Beendung des Programms werden alle Partikel mit *removeParticles* entfernt.