Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Aurora - Animation und Echtzeit Rendering

Dominik Sellenthin

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Thomas Ertl
Betreuer/in:	Michael Becher, M. Sc. Tobias Rau, M. Sc. Dr. Guido Reina
Beginn am:	9. April 2018
Beendet am:	24. Oktober 2018

Kurzfassung

Unter den Aspekten Echtzeit und Animation, wird in dieser Arbeit die Simulation und Visualisierung der Aurora Borealis vorgestellt. Dafür müssen sowohl physikalische als auch mathematische Grundlagen geschaffen werden, die anschließend verwendet werden, um die verschiedenen Effekte der Aurora Borealis so realistisch wie möglich anzunähern und gleichzeitig den Rechenaufwand so gering wie möglich zu halten. Eine Fluidsimulation mit Hilfe der Navier-Stokes-Gleichungen ist zum einen im Stande, den notwendigen Realismus der Effekte anzunähern und zum anderen, den Rechenaufwand der Simulation so gering zu halten, das mit Unterstützung von Programmierschnittstellen wie OpenGL moderne Grafikprozessoren in der Lage sind, sowohl 2D- als auch 3D-Animationen der Aurora in Echtzeit zu realisieren. Dabei wurde auf das auf High-Performance ausgelegte Visualisierungsframework MegaMol aufgebaut, das mit seiner bereits vorhandenen Schnittstelle Interaktion während der Laufzeit ermöglicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

1 Einleitung

Die Aurora Borealis, auch als Nord- oder Polarlicht bekannt, ist ein physikalisches Phänomen, das in der Nähe des Nordpols in Form von farbigen, meist grünlich leuchtenden Bändern in Erscheinung tritt. Seit Jahrtausenden sorgt die Aurora für Erstaunen und lässt die Menschen rätseln, was es ist und woher es kommt. Die alten Griechen hielten die Aurora für die Schwester von Helios, dem Sonnengott, und Selene, der Göttin des Mondes. Nicht nur ist die Aurora atemberaubend anzusehen, sie erregt durch ihre komplexen Vorgänge auch auf wissenschaftlicher Ebene Aufsehen.
Die physikalischen Hintergründe zur Entstehung der Aurora sind heute zum Großteil erforscht. Gelangen geladene Partikel, Protonen und Elektronen, die durch Sonnenwinde in Richtung Erde ausgesandt werden, in den Wirkungsbereich des Erdmagnetfelds, werden diese zu den Polen abgelenkt. Die Aurora Borealis ist deshalb nur in der Nähe um den Nordpol zu sehen, dem Aurora-Oval oder der Aurora-Ellipse. Ihr Pendant, die Aurora Australis, ist nur um den Südpol zu sehen. Auf ihrem Weg zur Erde kollidieren die Partikel mit Atomen und Molekülen, woraufhin diese angeregt und ionisiert werden. Folglich senden die ionisierten Moleküle Licht bestimmter Wellenlängen aus. Durch diesen komplexen Vorgang entstehen Effekte, die der Aurora ihre unterschiedlichen Formen verleihen.
Die unterschiedlichen Formen der Aurora ähneln denen von Fluiden und macht die Aurora daher interessant für Simulationen. Aus diesem Grund wurden im Folgenden mit einer Fluidsimulation die komplexen Vorgänge der Aurora simuliert. Dazu wurde zuerst ein CPU-basiertes Programm erstellt, das ein Geschwindigkeits- und ein Dichtefeld, in dem sich ein Fluid befindet, so simuliert, dass das Fluid die Formen einer Aurora wiederspiegelt. Als Grundlage der Fluidsimulation dienen die Navier-Stokes-Gleichungen, die als mathematisches Modell verwendet werden, um Strömungen von viskosen Newtonschen Fluiden darzustellen. Bei der Simulation wurde dabei besonderer Wert auf Echtzeit-Animationen sowohl in 2D als auch in 3D gelegt. Deshalb wurde zusätzlich ein Programm erstellt, das die Simulation auf einem Grafikprozessor ausführt. Für Echtzeit-Animationen wird nicht nur eine effiziente Simulation, sondern auch eine effiziente Visualisierung benötigt. Damit eine 3D-Visualisierung effizient durchgeführt werden kann, wird ein effizienter Volumen-Renderer benötigt. Dafür stellt das Visualisierungsframework MegaMol einen Raycaster zur Verfügung. Anschließend wird MegaMol, ein Prototypen-Framework zur partikelbasierten Visualisierung, vorgestellt. MegaMol wurde für GPU-beschleunigte High-Performance Visualisierung entwickelt und besteht aus Plugins, die Lösungen für unterschiedliche Visualisierungs-Anwendungen beinhalten. Für die Darstellung der 3D-Aurora wurde deshalb ein Plugin in MegaMol erstellt, das sowohl die Simulation als auch die Visualisierung beinhaltet.
Zu Beginn in Kapitel 2 werden die physikalischen Hintergründe der Entstehung einer Aurora und die benötigten Grundlagen einer Fluidsimulation besprochen. Anschließend wird der Ablauf in Abschnitt 3.1 und die Implementierung der Simulation in Abschnitt 3.2 vorgestellt. Auf verschiedene Rendertechniken, die im Laufe dieser Arbeit verwendet werden, wird in Abschnitt 3.4 eingegangen. Im Anschluss an das Rendering und die Implementierung, wird in Abschnitt 3.5

das Visualisierungsframework MegaMol vorgestellt und wie es in Verbindung mit der Simulation genutzt wurde. Zum Schluss werden in Kapitel 4 die Ergebnisse verschiedener Simulationsprobleme und der Aurora vorgestellt.

2 Grundlagen

2.1 Aurora Borealis

Aurora Borealis, auch bekannt als Polar- oder Nordlicht (hier und im Folgenden nur Aurora genannt), wird aus den Worten „borealis“ (lat. für „nördlich“) und „aurora“ (lat. für „Morgenröte “) zusammengesetzt und ist ein physikalisches Phänomen auf der Nordhalbkugel. Sein Gegenüber auf der Südhalbkugel nennt sich Aurora Australis („australis“, lat. für „südlich“).

2.1.1 Entstehung der Aurora Borealis

Seinen Ursprung hat die Aurora in der Sonne. Von der Sonne aus werden Elektronen und Protonen ausgestrahlt. Die ausgestrahlten Elektronen und Protonen kollidieren dabei mit anderen Atomen, geben ihre Energie ab und erzeugen dadurch einen visuellen Effekt. Hier sei angemerkt, das Protonen nicht sonderlich effizient bezüglich der Generierung von Licht sind, das zur Aurora beiträgt und somit einen lediglich geringen Anteil bilden und daher vernachlässigt werden können [Bar+00]. Im Folgenden wird deshalb nur von Elektronen gesprochen, allerdings wissend, dass dies ebenfalls Protonen beinhaltet.

Elektronen, die von der Sonne aus in Richtung Erde wandern und in die Erdatmosphäre eintreten wollen, treten in einer bestimmten „kleinen“ Region um das Zentrum der magnetischen Pole ein. Auf ihrem Weg zur Erde kollidieren diese Elektronen mit anderen Atomen, die sich in der Atmosphäre befinden. Kommt es zu einer Kollision, werden Atome angeregt und in einen Zustand mit hoher Energie versetzt. Über die Zeit wird diese Energie in Form sichtbaren Lichts wieder freigegeben. Die wohl am häufigsten vorkommende Wellenlänge beträgt dabei ca. 558nm. Durch die Freisetzung sichtbaren Lichts bildet sich eine Art „Vorhang“, der als *Aurora Borealis* bekannt ist [Bar+00].

Elektronen und Protonen (atomare Partikel), die durch Sonneneruptionen von der Sonne ausgehen, bilden Solarwinde. Ein geringer Anteil dieser Solarwinde interagieren mit dem Magnetfeld der Erde. Der interagierende Teil des Solarwinds bildet eine *shock front* vor der Erde. Ein paar Partikel wandern allerdings auf die andere Seite der Erde und bilden ein sogenanntes *plasma sheet*. Abbildung 2.1 verdeutlicht diesen Vorgang. Von diesem *plasma sheet* aus werden nun zeitweise Partikel beschleunigt und mit hoher Geschwindigkeit zur Erde gedrängt. Woher diese plötzliche Beschleunigung und Ablenkung zur Erde stammt, ist bislang unklar. Die durch diesen Prozess zur Erde beförderten Partikel wandern nun in einer umkreisenden spiralförmigen Bewegung entlang Magnetfeldlinien zu den bereits oben angesprochenen Regionen um die Pole und bilden die *Aurora Borealis* [Bar+00].

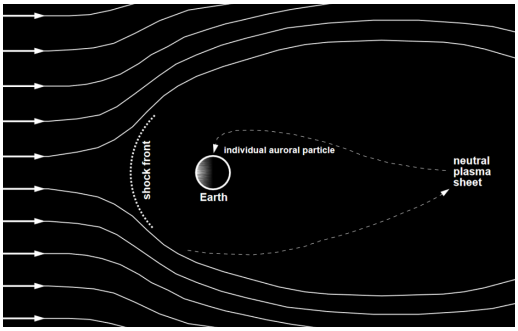

Abbildung 2.1: Solarwind, der eine *shock front* und ein *plasma sheet* bildet [Bar+00].

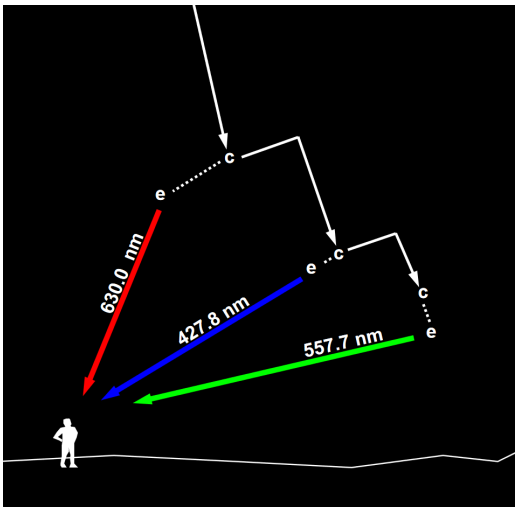

Abbildung 2.2: Wellenlängen in Abhängigkeit von der Höhe [Bar+00].

2.1.2 Lichtemission

Kollisionen von Elektronen mit anderen Atomen in der Atmosphäre sorgen dafür, dass die Elektronen horizontal aus ihrer Bahn geworfen werden. Wie eingangs bereits angesprochen, haben diese Kollisionen zur Folge, dass die kollidierten Atome angeregt werden und mit einer gewissen Wahrscheinlichkeit ein Photon emittieren. Bis zur Freisetzung von Photonen benötigt es in den meisten Fällen allerdings mehrere Kollisionen. Die am häufigsten vorkommende Wellenlänge von Photonen beträgt $557.7nm$. Jedoch hängt die Wellenlänge von mehreren Faktoren ab. Hauptsächlich hängt sie davon ab, mit welchem atmosphärischen Bestandteil das Elektron kollidiert und wie stabil der Zustand des angeregten Atoms ist. Es besteht daher ein Zusammenhang mit der Höhe, in der das Elektron kollidiert, da sich die Verteilung der Bestandteile der Atmosphäre mit der Höhe ändert [Bar+00].

Die Aurora befindet sich allgemein in einer Höhe zwischen etwa 80 km - 500 km über der Erdoberfläche und erstreckt sich dabei von Stratosphäre bis in die Exosphäre, wobei sich der Hauptteil zwischen 100 km und 300 km befindet. Den primären Anteil der Teilchen in dieser Höhe, der zum visuellen Effekt der Aurora führt, bilden vor allem atomarer Sauerstoff (atomic oxygen), atomarer Stickstoff (atomic nitrogen) und molekularer Stickstoff (molecular nitrogen) [Bar+00].

Das Spektrum der Aurora kann in zwei Teile aufgeteilt werden. Zum einen besteht das Spektrum aus spektralen Emissionslinien (spectral emission lines), die entstehen, sobald Elektronen mit Atomen kollidieren und zum anderen aus spektralen Emissionsbändern (spectral emission bands), die durch Kollision von Elektronen mit Molekülen Zustande kommen. Größtenteils erscheint die Aurora grün mit der Wellenlänge $557,7nm$. Da das menschliche Auge bei circa $555,0nm$ am sensibelsten ist, erscheint dieser Teil am hellsten. Dieser grüne Teil entsteht durch Emission von Photonen in einer Höhe von etwa 100 km, die bei Kollisionen zwischen Elektronen und Sauerstoffatomen emittiert werden (atomic oxygen green line). Ebenfalls durch Kollisionen zwischen Sauerstoffatomen und Elektronen entstehen Photonen der Wellenlänge $630,0nm$ (atomic oxygen red line), die sich vor allem in den oberen Regionen der Aurora befinden. Kommt es zu Kollisionen mit ionisiertem Stickstoff (ionized nitrogen), entsteht ein bläuliches Band (ionized nitrogen blue band), das manchmal am unteren Ende der Aurora sichtbar ist [Bar+00].

Nun kann der Zusammenhang zwischen der Wellenlänge und Höhe nochmals aufgegriffen werden. Begleitend hierzu wird auf Abbildung 2.2 verwiesen. Dazu kann die Dauer betrachtet werden, die Atome und Moleküle im angeregten Zustand verbringen. Die geringste Dauer mit bis zu unter $0,001s$ besitzen die angeregten Stickstoffatome, die für das bläuliche Licht zuständig sind. Angeregte Sauerstoffatome, die für das grüne Licht verantwortlich sind, befinden sich lediglich bis zu $0,7s$ in ihrem angeregten Zustand, bevor sie ihr Photon emittieren. Im Gegensatz dazu stehen angeregte Sauerstoffatome, die für rotes Licht verantwortlich sind. Mit bis zu $110s$ befinden sie sich wesentlich länger im angeregten Zustand, bis sie ihr Photon abgeben [Bar+00]. Es kommt nun ein Effekt ins Spiel, der als *quenching* (*dt.* löschend) bekannt ist. Je länger sich ein Atom im angeregten Zustand befindet, desto mehr Möglichkeiten existieren, weitere Kollisionen zu verursachen. Kollisionen mit anderen atmosphärischen Partikeln führen dazu, dass angeregte Atome ihre Fähigkeit, ein Photon zu emittieren, verlieren. Und hier liegt der Grund für die zum Teil höhenabhängige Farbverteilung. Dadurch, dass die Dichte der Atmosphäre mit zunehmender Höhe abnimmt, haben „rote Atome “(also angeregte Sauerstoffatome, die für rotes Licht zuständig sind) mit zunehmender Höhe eine höhere Chance, ihr Photon abzugeben, denn steigt die Dichte, steigt auch die Wahrscheinlichkeit das *quenching* auftritt. „Grüne Atome“hingegen haben durch ihre geringe Dauer im angeregten Zustand auch bei höherer Dichte (und damit niedriger Höhe) eine ausreichende Chance, ihr Photon abzugeben. Deshalb ist der untere Part der Aurora hauptsächlich grün und in den oberen Teilen befindet sich der sichtbare rote Anteil [Bar+00].

2.1.3 Formen der Aurora

Nachdem im vorherigen Teil die Farbgebung der Aurora besprochen wurde, folgt in diesem Teil die Formgebung der Aurora. Grundlegende Formen der Aurora werden sowohl durch die Energie und Dichte der Elektronen als auch durch lokale Schwankungen des elektrischen Felds der Erde bestimmt. Ebenfalls fließen kurze zeitliche Veränderungen mit ein, bedingt durch *magnetic substorms*, die durch Schwankungen des Erdmagnetfelds verursacht werden (und spiegeln sich in einer stärkeren Darstellung der Aurora wieder. Diese Schwankungen entstehen durch erhöhte Aktivität der Sonne. Eine höhere Aktivität der Sonne zieht dementsprechend eine stärkere Darstellung der Aurora nach sich [Bar+00].)

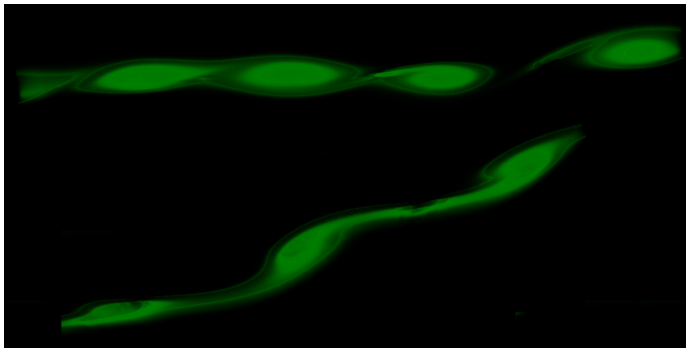

Abbildung 2.3: Oben: Beispiel mehrerer Wirbel (engl. curls). Unten: Beispiel einer Falte (engl. fold).

Eine der herkömmlichen Formen, die die Aurora annimmt, ist der weitestgehend bekannte *Vorhang* (engl. *curtain*). Seine Farbdistribution entspricht der bereits in Abschnitt 2.1.2 besprochenen Verteilung: (falls sichtbar) rötlich an der oberen Grenze, grün im Zentrum und im unteren Bereich (mit gegebenenfalls einem gelb-grünen Übergang an der unteren Grenze).
Durch die ständig neu eintreffenden Elektronen erhält die Aurora eine Dynamik. Diese Dynamik findet sich in vorkommenden „Verzerrungen“wieder und kommen in Form von *Spiralen* (engl. spirals), die in etwa $50km$ voneinander entfernt sind, *Wirbel* (engl. curls), mit einer Entfernung von ungefähr $2 - 10km$ zueinander und *Falten* (engl. folds), mit einem jeweiligen Abstand von ungefähr $20km$ zueinander, zum Ausdruck [Bar+00]. Am häufigsten zu beobachten sind jedoch nur Wirbel und Falten (Abbildung 2.3).

2.1.4 Aurora Morphologie

Ausgehend von der Grundform der Aurora, dem Vorhang, lässt sich die Aurora in weitere Arten von Formen unterteilen. Es wird hierfür zwischen zwei Arten unterschieden: die Aurora weist entweder eine Strahlenstruktur (ray structure) auf oder nicht. Strahlen sind dabei ein Nebenprodukt von Wirbeln. In den Wirbeln entwickeln sich Elektronenstrahlen, die ein dünnes Feld von „Fäden“ bilden, die als Strahlen interpretiert werden können. Ihr Durchmesser liegt bei etwa bei $1km$ und sie können bis zu mehrere hundert Kilometer hoch werden.
Enthält die Struktur der Aurora keine Strahlen, so lassen sich unter anderem drei Formen feststellen: Bögen, Bänder und „verteilte Felder“ (diffuse patches). Bögen und Bänder haben eine ähnliche homogene Struktur und können sich über mehr als $1000km$ erstrecken. Diffuse patches hingegen weisen eine wolkenähnliche Struktur auf und trotz dessen, das sie sich über mehrere hundert Quadratkilometer erstrecken, sind sie nur schwer zu erkennen, da sie keine hohe Lichtintensität aufweisen [Bar+00].

In der Gruppe der Aurorastrukturen, in denen Strahlen enthalten sind, finden sich vor allem: Strahlen, Strahlenbögen, Strahlenbänder, Corona und (drapery). Strahlen können auch einzeln oder in Gruppen vorkommen, in denen sie voneinander getrennt sind. Strahlenböden- und Bänder sind ähnlich wie normale Bögen und Bänder, mit dem Zusatz, das die Bögen und Bänder aus Strahlen bestehen. Die Corona ist eine Aurora, die aus Strahlen besteht (nahe dem magnetischen Zenit) und kronenähnlich erscheint. Sie hat Ähnlichkeiten mit einer Art Triangle-Strip, nur das Punkte

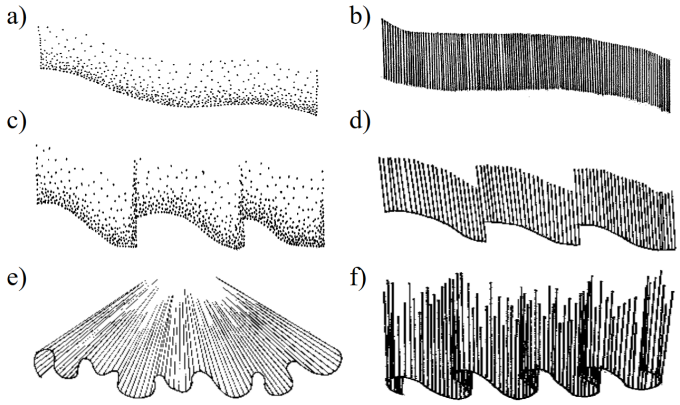

Abbildung 2.4: a) Bogen, b) Strahlenbogen, c) Band, d) Strahlenband, e) Corona, f) drapery [Bar+00].

ausschließlich mit dem Ursprungspunkt verbunden werden. Drapery, bestehend aus einem Band langer Strahlen, sieht einem normalen gefalteten Aurora-Vorhang und Strahlenbändern sehr ähnlich. Abbildung 2.4 verdeutlicht nochmals die beschriebenen Formen der Aurora.

2.1.5 Modellierung der Aurora

Ein typischer Aurora substorm beginnt mit einem einfachen, glatten Vorhang. Die Vorhänge fangen an zu wachsen und sich zu falten, wenn die substorms anfangen. Das resultiert am Ende in vielen sich überlappenden und interagierenden Vorhängen, die immer komplexer und lückenhafter werden, je mehr die substorms sich ihrem Ende nähern [LG11].
Zum Rendern der Aurora genügt es, diese komplexen Vorgänge/Effekte anzunähern. Die Vorgänge sind vom wissenschaftlichen Standpunkt noch nicht genau erforscht und sind zudem auch sehr rechenintensiv. Visuelle Abstraktionen können auf verschiedene Weisen erfolgen. Ein intuitiver Ansatz wäre zum Beispiel eine simple Kurve oder Spline zu nehmen, ihr eine entsprechende Höhe entlang der Kurve und die passende Farbe zu geben. Allerdings reicht dieser naive Ansatz nicht aus, um die komplexen Vorgänge (wie zum Beispiel Verwirbelungen) visuell darzustellen.
Die Autoren von [LG11] fanden heraus, dass sich diese komplexen Vorgänge durch eine geeignete Fluidsimulation gut annähern und darstellen lassen.
Die Simulation findet dabei in einem länglichen Bereich statt, dessen Grenzen periodisch sind, damit ein beliebig langer Vorhang bzw. ein beliebig langer Aurora-Strang erzeugt werden kann. Um die auftretenden Turbulenzen und Effekte der Aurora zu simulieren, wird ein Kelvin-Helmholtz Instabilitätsproblem gelöst, in dessen Zentrum sich die Aurora befindet.
Anfängliche Schritte der Simulation können zur frühen Darstellung der Aurora verwendet werden. Je weiter die Simulation voranschreitet, desto mehr Turbulenzen oder anderweitige Effekte treten auf und können zur Darstellung der späteren Phase einer Aurora dienen.

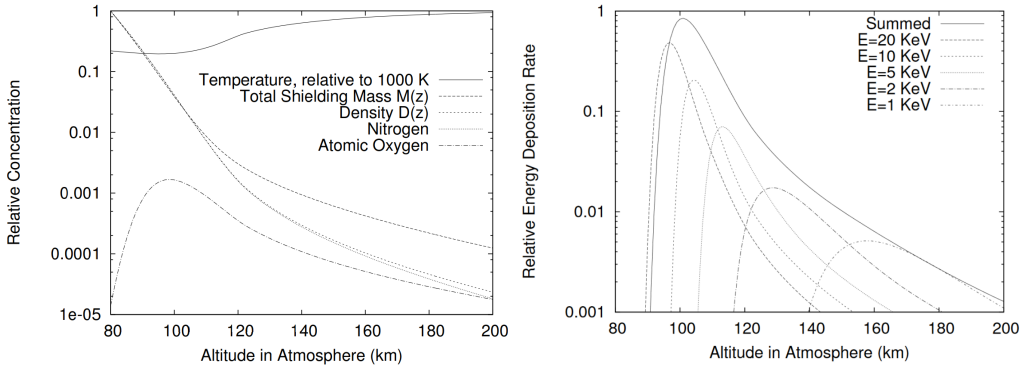

Abbildung 2.5: Links: MSIS-E-90 atmosphere. Rechts: Relative Energie-Absetzungsrate [LG11].

2.1.6 Vertikale Absetzung der Aurora

Verwirbelungen und andere auftretende Effekte der Aurora können in 2D simuliert werden. Um nun der Aurora noch eine Höhe zu verleihen, wird sich einer „vertikalen Absetzungsfunktion“ bedient.

Um die Höhe der Aurora beziehungsweise die Tiefe der Partikel, die die Atmosphäre durchdringen, exakt zu bestimmen, müssen sehr viele Aspekte wie zum Beispiel die Geschwindigkeit der Partikel, den Zustand der Atmosphäre, Energiezufuhr durch Solarstrahlung und noch vieles weitere betrachtet werden. Zwar ist eine genau Berechnung des Ganzen möglich, zum Beispiel mit „NCAR’s thermospheric general circulation model“, allerdings ist der Rechenaufwand viel zu hoch, um irgendeine Form der Echtzeit-Interaktion oder Animation durchzuführen. Es wird deshalb auf ein einfacheres Model zurückgegriffen, dem *MSIS-E-90 athmosphere model*, siehe dazu auch Abbildung 2.5 [LG11].

Die „Aurora Energy deposition rate“ kann nun mit Hilfe des „Lazarev charged particle energy deposition model“ berechnet werden. Als Eingabe für dieses Model wird folgendes benötigt [LG11]:

- Höhe z über der Erdoberfläche
- E Partikelenergie: Initialenergie der eintreffenden Partikel in Kilo-Elektronenvolt keV
- $M_z = \int_z^\infty D_{z'} dz'$: Masse der Atmosphäre über der Höhe z
- D_z: die Dichte in der Höhe z

Mittels dieser Eingabeparameter lassen sich nun weitere Parameter berechnen, die für die Ausgabe relevant sind [LG11]:

- $M_E = 4.6 * 10^{-6} E^{1.65}$: die charakteristische „shielding mass“ für Partikel mit Energie E
- $r = \frac{M_z}{M_E}$: die relative Durchdringungstiefe der Partikel
- $L = 4.2re^{-r^2-r} + 0.48e^{-17.4r^{1.37}}$: Lazarevs Interaktionsrate

Als Ausgabe folgt schließlich die „auroral energy deposition rate A_z“:

$$A_z = LE\frac{D_z}{M_E}$$

Die Aurora befindet sich im Bereich von $1 - 30keV$ für die Partikelenergie E. Abbildung 2.5 zeigt die für verschiedene Anfangswerte für E und dazugehörigen Höhen z entsprechenden Absetzungsraten.

2.2 Stable Fluids

2.2.1 Einführung

Als Basis der Fluidsimulation werden die „Fluidmechaniken“ als mathematische Grundlage verwendet. Die Navier-Stokes-Gleichungen (später mehr) sind dabei ein gutes Model dafür. Es gilt bei der Lösung der Navier-Stokes-Gleichungen prinzipiell zwei Punkte zu beachten: physikalische Korrektheit und das Aussehen. In diesem Fall ist ein gutes Aussehen der physikalischen Korrektheit (viel Rechenaufwand) vorzuziehen (in anderen Anwendungsfällen ist das durchaus anders!).
Verwirbelungen (engl. turbulences) sind Masseerhaltend und zeigen eine Rotationsbewegung.
Die Suche nach einem Löser ist also von großer Bedeutung und eines der Hauptprobleme solcher Löser ist, dass sie für größere Zeitschritte instabil werden. Instabilität führt zu numerischen Simulationen die „explodieren“ und deshalb mit einem kleineren Zeitschritt neu gestartet werden müssen. Die Folge ist: Limitierung der Geschwindigkeit und der Interaktion. Eine „Explosion“ bedeutet in diesem Fall, das geringe numerische Fehler sich durch die Simulation fortsetzen und dabei immer größer werden, bis die Werte gänzlich unbrauchbar sind.

Das Stabilitätsproblem des Lösers wird durch Semi-Lagrange-Schemata und eine implizite Methode adressiert, die die Navier-Stokes-Gleichungen lösen. Durch die gewonnene Stabilität sind nun Echtzeitinteraktionen möglich, da die zeitliche Schrittweite hochgesetzt werden kann.

2.2.2 Stabiler Navier-Stokes

Grundlegende Gleichungen

Ein Fluid (bei fast konstanter Temperatur und Dichte) wird durch ein Geschwindigkeitsfeld **u** und ein Druckfeld p beschrieben. Beide Größen verändern sich in Raum und Zeit und sind von den Grenzen abhängig, die das Fluid umgeben. Im Folgenden die räumliche Koordinate als **x** abgekürzt und steht für $\mathbf{x} = (x, y)$ für 2D, respektive $\mathbf{x} = (x, y, z)$ für 3D [Sta99].
Voraussetzung für die Navier-Stokes-Gleichung ist, dass das Geschwindigkeits- und Druckfeld zu irgendeinem Zeitpunkt t = 0 bekannt sein müssen. Ist das der Fall, kann die Entwicklungen der Größen über die Zeit wie folgt beschrieben werden [Sta99]:

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + f \tag{2.2}$$

„·“ ist das Skalarprodukt, v ist die kinematische Viskosität des Fluids, ρ die Dichte und f ist eine externe Kraft. ∇ ist der Vektor der partiellen Ableitungen $\nabla = (\partial/\partial x, \partial/\partial y)$. Der 3D Fall gilt analog. Die Navier-Stokes-Gleichungen sind sowohl Masse- als auch Impulserhaltend [Sta99]. Die Navier-Stokes-Gleichungen müssen noch um Randbedingungen ergänzt werden. Dafür werden zwei verschiedene Arten von Restriktionen vorgesehen: periodische und feste Grenzen. Im Falle der periodischen Restriktionen ist das Fluid auf einem 2- bzw. 3D Torus definiert, so dass es keine Wände gibt, sondern das Fluid sich herumschlingt [Sta99]. Bei fixen Restriktionen liegt das Fluid in einem begrenzten Bereich D. Die Restriktionen werden durch eine Funktion u_D gegeben, die auf der Grenze ∂D des Bereichs D definiert ist. Egal wie die Restriktionen festlegt werden, es sollte immer gegeben sein, dass Materie nicht durch Wände wandern kann. Die Geschwindigkeit direkt an der Grenze sollte daher 0 sein.
Gleichungen (2.1) und (2.2) können zu einer Gleichung kombiniert werden. Laut der *Helmholtz-Hodge Decomposition* kann jedes Vektorfeld $\mathbf{w}$ eindeutig in folgende Form aufgeteilt werden [Sta99]:

$$\mathbf{w} = \mathbf{u} + \nabla q \tag{2.3}$$

wobei q ein Skalarfeld ist und $\mathbf{u}$ keine Divergenz hat: $\nabla \cdot \mathbf{u} = 0$. Das heißt, jedes Vektorfeld kann als Summe eines masseerhaltenden Feldes und eines Gradientenfeldes dargestellt werden. Ein Operator $\mathbf{P}$ projiziert ein beliebiges Vektorfeld $\mathbf{w}$ auf seinen divergenzfreien Teil $\mathbf{u} = \mathbf{P}\mathbf{w}$ (siehe Abschnitt 2.2.2 für mehr zu Divergenz).$\mathbf{P}$ ist implizit definiert, in dem beide Seiten von Gleichung (2.3) mit ∇ multipliziert werden. Das Ergebnis dieser Operation ist [Sta99]:

$$\nabla \cdot \mathbf{w} = \nabla^2 q \tag{2.4}$$

Gleichung (2.4) ist eine Poisson-Gleichung für das Skalarfeld q mit der Neumann-Randbedingung $\frac{\partial q}{\partial n} = 0$ auf dem Bereich D. Eine Lösung dieser Poisson-Gleichung wird verwendet, um die Projektion $\mathbf{u}$ zu berechnen [Sta99]:

$$\mathbf{u} = \mathbf{P}\mathbf{w} = \mathbf{w} - \nabla q$$

Wird dieser Projektionsoperator auf beide Seiten von Gleichung (2.2) angewandt, ergibt sich eine einzige Gleichung für die Geschwindigkeit [Sta99]:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{P}(-(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + f) \tag{2.5}$$

wobei hier $\mathbf{P}\mathbf{u} = \mathbf{u}$ und $\mathbf{P}\nabla p = 0$ verwendet wurde. Aus dieser Gleichung kann nun ein stabiler Solver für Fluide entwickelt werden.

$$\mathbf{w}_0(\mathbf{x}) \xrightarrow{\text{add force}} \mathbf{w}_1(\mathbf{x}) \xrightarrow{\text{advect}} \mathbf{w}_2(\mathbf{x}) \xrightarrow{\text{diffuse}} \mathbf{w}_3(\mathbf{x}) \xrightarrow{\text{project}} \mathbf{w}_4(\mathbf{x}).$$

Abbildung 2.6: Vier Schritte für einen Simulationsschritt [Sta99].

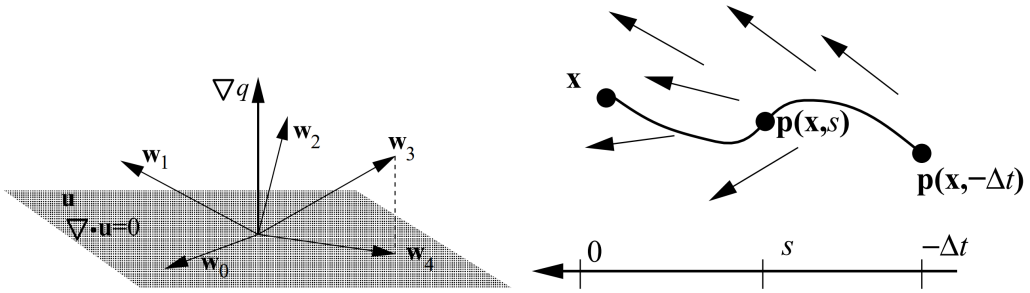

Abbildung 2.7: Links: Illustration der 4 Schritte zum Lösen eines Simulationschrittes. Rechts: Illustration des Backtracing für den Advektion-Schritt [Sta99].

Anmerkungen Im Sinne der Massenerhaltung ist die Forderung nach Divergenzfreiheit wichtig. Divergenz ist ein Skalarfeld, das mittels Differentialoperator aus einem Vektorfeld gewonnen wird und gibt für jeden Punkt des Feldes an, wie sehr dieser Punkt als „Quelle“ (positive Divergenz) oder „Senke“ (negative Divergenz) dient. Übertragen auf einen Punkt mit positiver Divergenz innerhalb eines Dichtefelds, bedeutet das, dass dieser Punkt als Quelle dient und somit zusätzliche Dichte hinzukommt. Für den Fall der negativen Divergenz bedeutet das, dass Dichte das Feld verlässt [**tong2003discrete**]. Das Gleiche gilt analog für ein Geschwindigkeitsfeld. Egal, ob positive oder negative Divergenz vorliegt, das Feld ist nicht mehr Masseerhaltend.

2.2.3 Lösungsmethode

Um nun mit Hilfe von Gleichung (2.5) einen Simulationsschritt zu absolvieren, wird ein Initialzustand $\mathbf{u}_0 = \mathbf{u}(\mathbf{x}, 0)$ benötigt. Von diesem Zustand aus wird nun ein Zeitschritt der Weite Δt berechnet. Das heißt, angenommen das Feld ist zum Zeitpunkt t bekannt und soll nun zum Zeitpunkt $t + \Delta t$ gelöst werden. Gleichung (2.5) wird nun in vier Schritten über den Zeitschritt Δt gelöst [Sta99]. Es wird dafür von einem Initialzustand $\mathbf{w}_0(\mathbf{x}) = \mathbf{u}(\mathbf{x}, t)$ ausgegangen, auf diesen dann sequenziell die einzelnen Terme von Gleichung (2.5) angewandt werden, gefolgt von einem abschließenden Projektionsschritt, um ein divergenzfreies Feld zu erhalten. Abbildung 2.6 zeigt nochmal die benötigten Schritte, um einen Simulationsschritt zu berechnen. Das Ergebnis eines Simulationsschrittes zum Zeitpunkt $t + \Delta t$ ist das letzte Geschwindigkeitsfeld und zwar [Sta99]: $\mathbf{u}(\mathbf{x}, t + \Delta t) = w_4(\mathbf{x})$.

Im Folgenden wird auf die einzelnen Schritte näher eingegangen. Der wohl einfachste und erste Schritt ist die Hinzunahme einer externen Kraft (letzter Term aus Gleichung (2.5)). Diese wird unter Berücksichtigung von Δt auf das aktuelle Feld addiert (dabei wird angenommen, dass die Kraft sich wenig bis gar nicht über den Zeitschritt ändert.) Daraus ergibt sich folgende Annäherung [Sta99]:

$$\mathbf{w}_1(x) = \mathbf{w}_0(x) + \Delta t f(\mathbf{x}, t)$$

Als zweiter Schritt wird der Effekt der Advektion (engl. advection) durchgeführt. Die Advektion wird durch den ersten Term aus Gleichung (2.5) beschrieben, nämlich $-(\mathbf{u} \cdot \nabla)\mathbf{u}$. Advektion beschreibt, wie sich eine „Störung“ im Fluid verhält. Genauer gesagt, beschreibt die Advektion das beispielsweise Moleküle oder Partikel sich mit dem Geschwindigkeitsfeld bewegen. Ein Dichtefeld bewegt sich also mit dem Geschwindigkeitsfeld. Für ein Geschwindigkeitsfeld gilt, dass es sich mit sich selbst bewegt (*self-advection*) [Sta99]. Und hier liegt eines der Probleme der NSG, nämlich, dass dieser Term die NNSG nicht-linear machen und dadurch die Stabilität nicht zwingend gegeben ist. Das Problem, das sich aus der Instabilität ergibt, ist, dass die Simulation nur für Zeitschritte korrekt läuft, sofern diese ausreichend klein sind. Ist Δt zu groß, explodiert die Simulation und die Simulation ist damit nicht mehr brauchbar. Um diese Instabilität in den Griff zu bekommen, verwenden der Autor aus [Sta99] die sogenannte "method of characteristics". Für weitere Details bezüglich dieser Methode wird der Leser auf den Anhang der hier verwendeten Literatur [Sta99] verwiesen. Bei diesem Schritt wird versucht, die Geschwindigkeit an einem Punkt $\mathbf{x}$ zu einem neuen Zeitschritt $t + \Delta t$ zu berechnen. Das Prinzip dahinter ist nun, dass ein Punkt $\mathbf{x}$ über den Zeitschritt Δt durch das vorherige Feld $\mathbf{w}_1$ zurückverfolgt wird. Dieses Konzept ist als *backtracing* bekannt, siehe dazu Abbildung 2.7. Das heißt, die neue Geschwindigkeit bei $\mathbf{x}$ ist gleich der Geschwindigkeit, die das entsprechende Partikel im vorherigen Zeitschritt an seiner alten Position im vorherigen Feld besitzt [Sta99]. Das ist auch der Grund für die Stabilität dieses Schrittes. Die neuen Werte des aktualisierten Geschwindigkeitsfeldes können nicht höher sein als die Werte des alten Feldes. Deswegen kann es bei dieser Methode zu keiner Explosion kommen. Genauer gesagt gilt [Sta99]:

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{p}(\mathbf{x}, -\Delta t))$$

wobei $\mathbf{p}(\mathbf{x}, -\Delta t)$ den Pfad des Partikels durch das Geschwindigkeitsfeld beschreibt.

Nachdem nun der Einfluss einer externen Kraft und die Aktualisierung der Geschwindigkeit besprochen wurde, folgt nun Schritt 3, die Behandlung der Diffusion beziehungsweise des Effekts der Viskosität entsprechend der Gleichung [Sta99]:

$$\frac{\delta \mathbf{w}_2}{\delta t} = \nu \nabla^2 \mathbf{w}_2$$

Um auch hier die Stabilität zu wahren, wird eine implizite Methode verwendet [Sta99]:

$$(\mathbf{I} - \nu \Delta t \nabla^2)\mathbf{w}_3(\mathbf{x}) = \mathbf{w}_2(\mathbf{x})$$

wobei $\mathbf{I}$ der Identitätsoperator ist. Wird nun der Diffusionoperator ∇^2 diskretisiert, führt das zu einem dünnbesetzten linearen Gleichungssystem (sparse linear system) für das unbekannte Feld $\mathbf{w}_3$, das es nun effizient zu lösen gilt (später mehr). Es ist auch möglich den Diffusionsoperator direkt zu diskretisieren, ohne die eben genannte implizite Methode zu verwenden und anschließend einen expliziten Zeitschritt zu berechnen. Allerdings hat das den Nachteil, das für ausreichend große Werte für die Viskosität, die Simulation nicht mehr stabil ist [Sta99].

In einem letzten Schritt muss das gewonnene Feld noch divergenzfrei gemacht werden. Dafür ist Schritt 4 notwendig, der Projektionsschritt. Wie bereits bei Gleichung (2.4) angesprochen, wird dafür die Lösung einer Poisson-Gleichung benötigt [Sta99].

$$\nabla \cdot \mathbf{w}_3 = \nabla^2 q$$
$$\mathbf{w}_4 = \mathbf{w}_3 - \nabla q$$

2.2.4 Substanzbewegung durch das Fluid

Nachdem im vorherigen Kapitel das Verhalten der Geschwindigkeit besprochen wurde, fehlt noch das Verhalten einer Substanz innerhalb des Fluids. Wird eine Substanz (die nicht mit dem Fluid reagiert) in das Fluid gegeben, wird sie sowohl durch das Fluid bewegt (advected) als auch verteilt (diffused). Sei a nun eine skalare Größe die durch das Fluid bewegt wird (beispielsweise die Dichte von Rauch). Dann kann die Entwicklung dieses skalaren Feldes durch eine Gleichung beschrieben werden, die der eingänglichen NSG Gleichung (2.2) ähnelt [Sta99]:

$$\frac{\delta a}{\delta t} = -\mathbf{u} \cdot \nabla a + \kappa_a \nabla^2 - \alpha_a a + S_a$$

wobei κ_a die Diffusionskonstante, α_a die Zerstreuungsrate (dissipiation rate) und S_a ein Quellterm. Wie auch hier zu sehen ist, ist sie der NSG sehr ähnlich, da sie einen Diffusionsterm, einen Advektionsterm und einen Quellterm besitzt, der der Form des Terms der externen Kraft entspricht. Diese drei Terme können genau wie im vorangegangenen Kapitel gelöst werden. Für den Zerstreuungsterm $-\alpha_a a$ gilt es nun noch folgende Gleichung zu lösen [Sta99]:

$$(1 + \Delta t \alpha_a) a(\mathbf{x}, t + \Delta t) = a(\mathbf{x}, t)$$

2.2.5 Randbedingungen

Randbedingungen spielen eine wichtige Rolle bei der Simulation eines Fluids. Sie bestimmen, wie sich das Fluid innerhalb der Grenzen eines Raumes verhält. Zu den bekanntesten Randbedingungen zählen die Neumann- und die Dirichlet-Randbedingungen und periodische Randbedingungen. Hier sei erwähnt, dass es auch noch die Robin-Randbedingung gibt, die eine Kombination von Neumann und Dirichlet ist, hier aber nicht weiter betrachtet wird. Realisierbar mit Hilfe von Randbedingungen sind Effekte wie zum Beispiel eine konstante Strömung, die an definierten Punkten am Rand zu jedem Zeitpunkt in den Raum fließt (z. B. Wind).

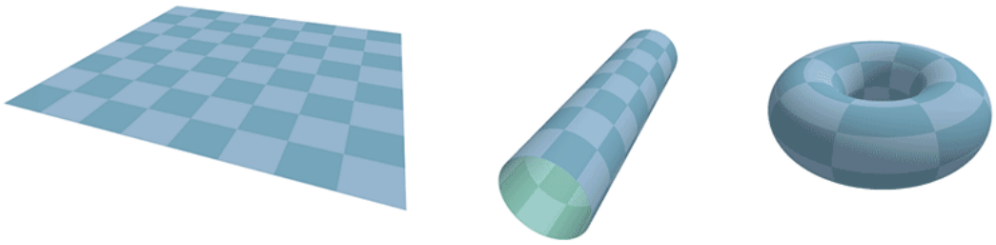

Abbildung 2.8: Illustration periodischer Randbedingungen mittels eines 3D-Torus.

Neumann-Randbedingung

Die Neumann-Randbedingung wurde bereits in einem vorherigen Kapitel erwähnt. Wichtig ist hierbei zu wissen, dass die Neumann-Randbedingungen eine Bedingung für die Ableitung darstellen. Bei Gleichung (2.4) wurde eine solche Bedingung bereits gestellt: $\frac{\partial q}{\partial n} = 0$ auf dem Bereich ∂D. Das heißt, an jedem Punkt auf dem Rand muss die Ableitung 0 sein. Mit dieser Bedingung wird sichergestellt, dass keinerlei Materie den Raum verlässt bzw. über die Grenzen hinauswandert (und damit gegen die Masseerhaltung, die bei Gleichung (2.1) gefordert wird, verstößt).

Dirichlet-Randbedingung

Anders als die Neumann-Randbedingung, gibt die Dirichlet-Randbedingung den direkten Wert am Rand an. Wird beispielsweise ein Geschwindigkeitsfeld mit Dirichlet-Randbedingungen betrachtet, bedeutet das, dass am Rand die Werte der Geschwindigkeiten explizit angegeben werden. Soll zum Beispiel eine Szene dargestellt werden, bei der konstant Wind von einem Rand in die Szene strömt (etwa durch einen Ventilator), kann dafür die Dirichlet-Randbedingung verwendet werden, indem am Rand die dafür benötigten Geschwindigkeiten festgelegt werden.

Periodische Randbedingung

Hinsichtlich der Simulation der Aurora sind periodische Randbedingungen ein wichtiger Part. Im Gegensatz zu Neumann- und Dirichlet-Randbedingungen stellen periodische Randbedingungen keine direkte Bedingung in Form von Ableitung oder Wert an den Rand, sondern „verbinden“ lediglich die Enden (links mit rechts und/oder oben mit unten) miteinander. Sei beispielsweise der linke mit dem rechten (oder andersum) Rand mittels periodischer Randbedingung verbunden, so würde Wind, der über den rechten Rand hinausgeht, wieder am linken Rand erscheinen (gleiches gilt für oben und unten). Das periodische Verhalten im Falle eines Feldes lässt sich gut durch einen 3D-Torus illustrieren, siehe dafür Abbildung 2.8. Zuletzt sei noch angemerkt, das eine Kombination verschiedener Randbedingungen möglich ist. Mittels einer Kombination aus periodischer Randbedingung am linken und rechten Ran und Neumann-Randbedingungen für den oberen und unteren Rand lässt sich ein Art „unendlicher“ Kanal konstruieren.

2.3 Reynoldszahl

Hinsichtlich der Simulation von Strömungen spielt die Reynoldszahl Re eine wichtige Rolle. Die Reynoldszahl beschreibt das Verhältnis zwischen Trägheits- und Zähigkeitskräften und ist definiert durch [**rapp2016microfluidics**]:

$$Re = \frac{\rho \cdot v \cdot d}{\eta} = \frac{v \cdot d}{\nu}$$

Die einzelnen Bestandteile sind die Fluiddichte ρ, die Fluidgeschwindigkeit v, die charakteristische Länge d und die dynamische Viskosität η bzw. kinematische Viskosität ν, wobei $\eta = \nu\rho$ gilt. Die charakteristische Länge d gibt das Ausmaß einer Geometrie an. Bei einer Rohrströmung ist dies beispielsweise der Innendurchmesser des Rohrs. Für ein Quadrat entspricht d der Länge einer Seite. Überschreitet Re einen problemabhängigen kritischen Wert Re_{krit} gehen laminare Strömungen in turbulente Strömungen über. Laminarströmungen haben die Eigenschaft, dass keine Turbulenzen (z. B. Verwirbelungen) in der Grenzschicht zweier unterschiedlicher Strömungsgeschwindigkeit auftreten. Bei Rohrströmungen, wie zum Beispiel der Hagen-Poiseuille-Strömung, liegt der Wert von Re_{krit} etwa zwischen 2000 und 2300. Überschreiten Rohrströmungen diesen Wert, gehen sie in eine Übergangsphase von laminaren zu turbulenten Strömungen über. Die Übergangsphase befindet sich im Intervall $2300 < Re < 4000$. Turbulente Strömungen liegen folglich für $Re \geq 4000$ vor [**rapp2016microfluidics**]. Diese Werte konnten in der Simulation bestätigt werden, siehe dazu Kapitel 4.

3 Implementierung der Fluidsimulation

Aufbauend auf den Grundlagen der vorherigen Kapiteln, wird nun Schritt für Schritt eine geeignete Simulation vorgestellt. Zuerst werden die nötigen Bestandteile und anschließend der generelle Ablauf der Simulation vorgestellt. Details zur exakten Implementierung folgen in Abschnitt 3.2 und Abschnitt 3.3.
Als Setup wurde folgendes gewählt:

- Programmiersprache: C++
- Visual Studio 2017 Enterprise
- OpenGL, GLFW, GLAD

Für die Implementierung werden kompakte Versionen der eingangs beschriebenen Gleichungen gewählt [Sta03]:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{3.1}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S \tag{3.2}$$

Innerhalb der Implementierung wird das in Gleichung (3.1) und Gleichung (3.2) verwendete Geschwindigkeitsfeld $\mathbf{u}$ in ein horizontales und ein vertikales Geschwindigkeitsfeld aufgeteilt.

Um die Simulation zu realisieren, wird folgendes benötigt:

- Variablen für:
  - die Dimensionen x, y, z,
  - den Zeitschritt *dt*,
  - die Viskosität *visc*,
  - und die Diffusion *diff*
- Jeweils zwei geeignete Datenstrukturen für die verschiedenen Felder:
  - vertikale Geschwindigkeit v
  - horizontale Geschwindigkeit u
  - Dichte *dens*

Algorithmus 3.1 Aktualisiere Geschwindigkeitsfelder

```
procedure VELOCITY STEP
    diffuse(u, v)
    project(u, v)
    advect(u, v)
    project(u, v)
end procedure
```

Es werden jeweils zwei Felder benötigt, da sowohl der aktuelle als auch der Stand, der einen Zeitschritt zurückliegt, gespeichert werden müssen (das ist insbesondere für das *backtracing* der Advektion wichtig).

3.1 Allgemeiner Ablauf

Der generelle Ablauf der Simulation kann in drei Schritte, die sich in einer Schleife befinden, unterteilt werden [Sta03]:

1. Initialisiere externe Kraft (f) und Quelle (S)
2. Aktualisiere Geschwindigkeitsfelder u und v (Gleichung (3.1))
3. Aktualisiere Dichte (Gleichung (3.2))

3.1.1 Initialisierung Kraft und Quelle

Der einfachste Schritt ist die Initialisierung der externen Kraft und der Quelle. Da für die Lösung der Navier-Stokes-Gleichung die Felder zu einem Startzeitpunkt vorhanden sein müssen, ist das auch der erste Schritt, der durchgeführt wird. Hier werden die Felder im allerersten Schritt jeweils mit Startwerten gefüllt. Für jede weitere Iteration der Schleife, wird dieser Schritt verwendet, um gegebenenfalls neu eintretende Kräfte (bspw. Wind, der neu in die Szene gelangt) und Quellen (bspw. Rauch) mit einzubeziehen.

3.1.2 Aktualisiere Geschwindigkeitsfelder

Um diesen Schritt zu realisieren, werden nun die einzelnen Schritte aus dem Abschnitt 2.2 aufgegriffen. Dazu wird Gleichung (3.1) von rechts nach links gelöst. Der letzte Term, die Addition einer externen Kraft f, wird bereits im ersten Schritt durchgeführt und kann daher in dieser Routine weggelassen werden. Eine Lösung des zweiten Terms $\nu\nabla^2\mathbf{u}$, um den Effekt der Viskosität einzubringen, kommt der Lösung einer Diffusionsgleichung gleich. Anschließend muss der noch übrig gebliebene Term $-(\mathbf{u}\cdot\nabla)\mathbf{u}$, der für die Advektion verantwortlich ist, berechnet werden. Damit der Masseerhaltung genüge getan wird, folgt zum Schluss noch der Projektionsschritt. Das Geschwindigkeitsfeld wird also zuerst verteilt, anschließend bewegt (self-advection) und von Divergenz befreit. Die Routine sieht entsprechend Algorithmus 3.1 aus [Sta03]:

Algorithmus 3.2 Aktualisiere Dichtefeld

procedure DENSITY STEP
diffuse($dens, dens^{prev}$)
advect($dens, u, v$)
end procedure

Algorithmus 3.3 Fluidsimulation

procedure SIMULATION
while Simuliere **do**
init_source
vel_step($u, v, u^{prev}, v^{prev}, visc, dt$)
dens_step($dens, dens^{prev}, u, v, diff, dt$)
end while
end procedure

Dass der Projektionsschritt zwei Mal vorkommt, ist kein Fehler, sondern ein Bestreben nach höherer Genauigkeit.

3.1.3 Aktualisiere Dichtefeld

Die Routine des letzten Schritts (Gleichung (3.2)) sieht der des zweiten sehr ähnlich und ist tatsächlich recht simpel. Die Addition der Quelle S kann hier ebenfalls weggelassen werden. Der mittlere Term $\kappa\nabla^2\rho$ entspricht der Diffusion und der erste Term $-(\mathbf{u}\cdot\nabla)\rho$ der Advektion. Für diesen Schritt ist ausschließlich ein Diffusions- und ein Advektionsschritt von Nöten. Das Dichtefeld wird hier also zuerst verteilt und anschließend dem Geschwindigkeitsfeld folgend bewegt. Die dazugehörige Routine entspricht Algorithmus 3.2 [Sta03]:

3.2 CPU Implementierung

Als erstes wurde die Simulation auf der CPU realisiert und getestet. Obwohl die Implementierung für eine 3D-Simulation ausgelegt ist, werden aus Performancegründen auf der CPU nur zwei Dimensionen genutzt. z ist zwar vorhanden und kann genutzt werden, wurde hier aber konstant auf 1 gesetzt, so dass nur auf einer Ebene simuliert wird.
Für die benötigten Variablen in C++ wurden einfache *floats* und für die Datenstrukturen werden Vektoren der Standardbibliothek verwendet. Außerdem wird ein Makro $IX(x, y, z) = x + y \cdot xsize + z \cdot xsize \cdot ysize$ verwendet, das den korrekten Index für den Vektorzugriff berechnet, um so den Zugriff zu erleichtern. x, y und z repräsentieren hierbei die aktuelle Position im Feld. $xsize$ und $ysize$ (und $zsize$) stehen für die gesetzten Größen der jeweligen Richtung. Die Implementierung erfolgt nun auf genau dem Wege, der weiter oben beschrieben wurde. Es existiert also eine Hauptschleife *simualte()* Algorithmus 3.3, in der die Initialisierungen und Aktualisierungen berechnet werden.

3.2.1 Funktion „init_source“

Hier wird lediglich den Feldern ihr Initialwert zugewiesen. Dafür werden die Felder verwendet, die die Lösung des vorherigen Zeitschritts beinhalten, u^{prev}, v^{prev} und $dens^{prev}$. Sei nun f eines der Felder und SourceForce der Wert, der an der jeweiligen Stelle festgelegt werden soll, dann reichen einfache Schleifen über jede Richtung aus, in denen $f_{x,y,z} = f_{x,y,z} + dt \cdot SourceForce$ berechnet wird.
Im allerersten Schritt ist f noch leer, das heißt es werden tatsächlich nur Werte festgelegt. Für jeden weiteren Zeitschritt gilt dementsprechend, dass die Werte auf den aktuellen addiert werden (zum Beispiel zusätzliche Dichte, die die Szene betritt).

3.2.2 Funktion „vel_step“

Stehen die Felder zu einem Zeitpunkt t fest, kann die Simulation mit der Aktualisierung der Geschwindigkeitsfelder beginnen. Dazu werden sowohl die Felder, die die aktuelle Lösung beinhalten - u und v - als auch die Felder, die die Lösung des vorherigen Zeitschritts beinhalten - u^{prev} und v^{prev} -, benötigt.

Einschub: Endliche Differenzapproximation, Poisson-Gleichung und Gauss-Seidel

Zur Lösung einzelner Schritte in der Simulation wird die Berechnung partieller Ableitungen benötigt. Eine Ableitung $u_x = \frac{\delta u}{\delta x}$ wird dafür durch die *zentrale Differenzapproximation* berechnet [WB00]: $\frac{\delta u}{\delta x} = \frac{u_{i+1} - u_{i-1}}{2\Delta x}$. Entsprechend wird die zweite Ableitung berechnet: $\frac{\delta u}{\delta x} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2}$.
Die Poisson-Gleichung $u_{xx} + u_{yy} = f(x, y)$ wird für die Modellierung einer vereinfachten viskosen Strömung verwendet (ist $f(x, y) = 0$, wird die Gleichung zur Laplace-Gleichung). Wird die zuvor angesprochene zweite Ableitung verwendet, lässt sich die Poisson-Gleichung einfach zu

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f(x, y)$$

bzw. unter Annahme von $b = \frac{\Delta x}{\Delta y}$ zu:

$$\frac{u_{i+1,j} + u_{i-1,j} + b^2 u_{i,j+1} + b^2 u_{i,j-1}}{2(1 + b^2)} - u_{i,j} = f(x, y)$$

umformen [CM10]. Solche Arten von Gleichungen bzw. lineare Gleichungssysteme können nun mit Hilfe des Gauss-Seidel-Verfahrens näherungsweise gelöst werden. Die Vorgehensweise erfolgt dabei iterativ entweder über eine feste Anzahl k oder bis ein Toleranzkriterium erfüllt ist, beispielsweise $|u^{m+1} - u^m| < toleranz$, wobei u^m die Lösung nach m Iterationen beinhaltet [CM10].

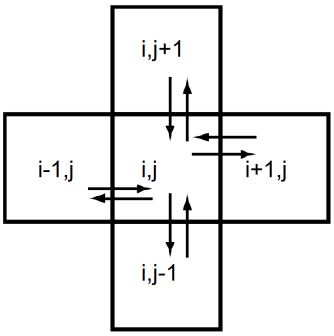

Abbildung 3.1: Interaktion einer Zelle mit seinen Nachbarzellen [Sta03].

Funktion „diffuse“

In diesem Schritt werden die Geschwindigkeitsfelder verteilt. Damit die Felder korrekt verteilt werden, muss die Viskosität mit einbezogen werden. Um das zu erreichen, muss eine zusätzliche neue Variable für die Diffusionsrate a angelegt werden: $a = dt \cdot visc \cdot xsize \cdot ysize$ ($\cdot zsize$ für eine 3D-Simulation). Das Prinzip hinter der Diffusion lässt sich gut anhand von Abbildung 3.4 verdeutlichen. Jede Zelle des Feldes interagiert mit den Zellen, die sie umgeben. In jedem Zeitschritt verliert eine Zelle Geschwindigkeit an die umgebenden Zellen, erhält zugleich aber auch Geschwindigkeit von den umgebenden Zellen. Um die Simulation einfach zu halten, besteht Interaktion nur mit den vier direkt angrenzenden Nachbarzellen. Die resultierende Differenz beträgt [Sta03]:

$$f_{x-1,y,z} + f_{x+1,y,z} + f_{x,y-1,z} + f_{x,y+1,z} - 4f_{x,y,z}$$

An dieser Stelle wird ein ähnliches Prinzip wie bei der Advektion verwendet. Und zwar wird versucht, die Dichte an der aktuellen Position so zu berechnen, dass wenn ein Zeitschritt zurückgegangen wird, der Wert des vorherigen Feldes herauskommt. Als Gleichung sieht das wie folgt aus [Sta03]:

$$f^{prev}_{x,y,z} = f_{x,y,z} - a(f_{x-1,y,z} + f_{x+1,y,z} + f_{x,y-1,z} + f_{x,y+1,z} - 4f_{x,y,z})$$

Mittels eines linearen Gleichungssystem ist es möglich, diese Gleichung zu lösen. In dieser Simulation wird allerdings ein einfacheres iteratives Verfahren verwendet: das Gauss-Seidel-Verfahren (siehe Abschnitt 3.2.2). Obige Gleichung stellt sich dadurch um, zu [Sta03]:

$$f_{x,y,z} = (f^{prev}_{x,y,z} + a(f_{x-1,y,z} + f_{x+1,y,z} + f_{x,y-1,z} + f_{x,y+1,z}))/(1 + 4a)$$

Da in dieser Variante der Simulation z konstant ist, müssen nur die Nachbarzellen in x- und y-Richtung betrachtet werden. Im 3D-Fall müssen noch zusätzlich die Zellen $f_{x,y,z-1}$ und $f_{x,y,z+1}$ berücksichtigt werden und der Nenner ändert sich zu $(1 + 6a)$.
In der Praxis stellt sich heraus, dass 20 Iterationen gute Ergebnisse für die Lösung ergeben. Die Diffusions-Routine setzt sich schlussendlich aus der Berechnung der Diffusionsrate und der Iterationen über die obige Gleichung zusammen. Der Effekt der Diffusion ist in Abbildung 3.2 dargestellt.

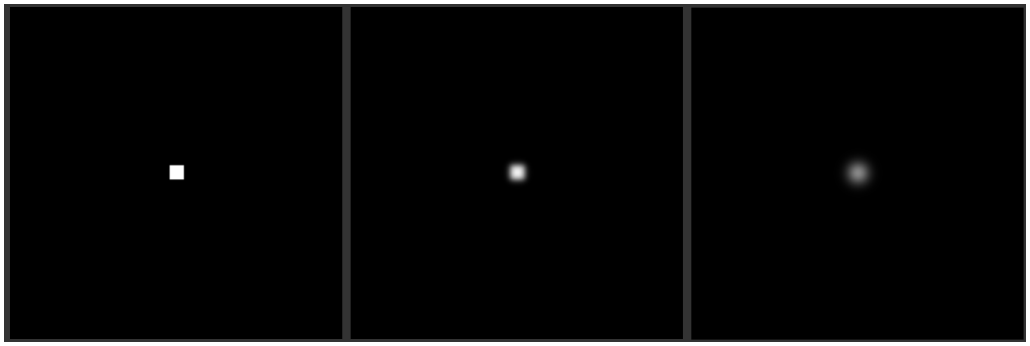

Abbildung 3.2: Effekt der Diffusion mit $diff = 0.00001$. Für $diff = 0$ findet keine Diffusion statt.

Algorithmus 3.4 Diffusion

procedure DIFFUSE
  $a \leftarrow dt * visc * xsize * ysize$
  for k from 1 to 20 **do**
    $f_{x,y,z} \leftarrow (f^{prev}_{x,y,z} + a(f_{x-1,y,z} + f_{x+1,y,z} + f_{x,y-1,z} + f_{x,y+1,z}))/(1 + 4a)$
  end for
end procedure

Funktion „advect“

Der nächste Schritt widmet sich der Advektion. Dazu wird das Prinzip aus Abschnitt 2.2 noch einmal rekapituliert. Angenommen, eine Zelle sei ein Partikel, das sich durch das Geschwindigkeitsfeld bewegt. Ziel ist es nun vorerst, die entsprechende Geschwindigkeit, die das Partikel vor einem Zeitschritt besitzt, zu finden. Das heißt, zuerst muss die Position $\mathbf{p}(\mathbf{x}, -\Delta t)$ des Partikels im vorherigen Feld ermittelt werden.
Die Berechnung der alten Position hängt allerdings von den Randbedingungen ab bzw. ob diese periodisch oder nicht-periodisch sind. Für die Folgende Implementierung gelten nicht-periodische Randbedingungen (für periodische Randbedingungen siehe weiter unten). Exemplarisch anhand der x-Richtung sehen die Schritte dafür wie folgt aus (die y-Richtung folgt analog aus der x-Richtung) [Sta03]:
Berechne zuerst die zurückgelegte Strecke: $traveled = dt \cdot xsize \cdot f_{x,y,z}$ und anschließend die alte Position: $x^{old} = x - traveled$. An diesem Punkt müssen allerdings zwei Sonderfälle beachtet werden, nämlich wenn die Position x_old außerhalb der Grenzen liegt (hier liegt auch der Unterschied zu periodischen Randbedingungen). Zum einen für den Fall $x^{old} < 0.5$ und zum anderen für den Fall $x^{old} > xsize + 0.5$. Auf unter oder über .5 wird deshalb getestet, weil immer die Zellmitten betrachtet werden. Sofern einer der beiden Fälle eintritt, genügt es bei den hier angenommenen (nicht-periodischen) Randbedingungen, x^{old} auf die entsprechenden getesteten Grenzen zu setzen, sprich $x^{old} = 0.5$ im ersten und $x^{old} = xsize + 0.5$ im zweiten Fall.
Wurde die alte Position erfolgreich gefunden, wird mittels linearer Interpolation der vier nächsten Zellen der neue Wert ermittelt und in das aktuelle Feld eingetragen [Sta03]. Der Effekt der Advektion ist in Abbildung 3.3 dargestellt.

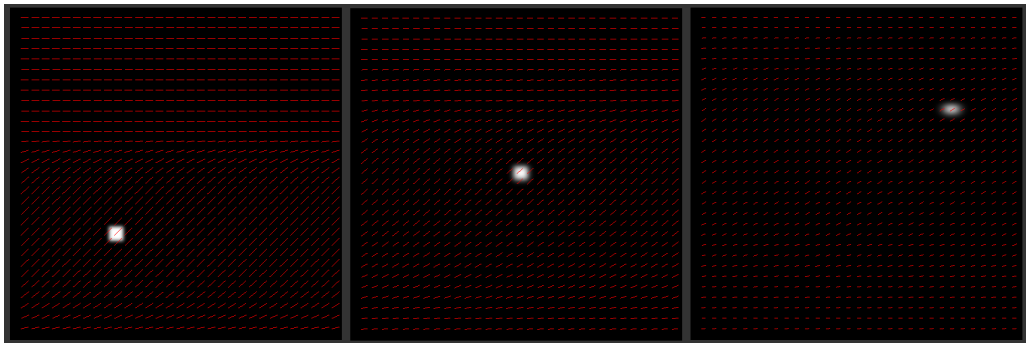

Abbildung 3.3: Effekt der Advektion. Dichte (in weiß) wird entsprechend dem Geschwindigkeitsfeld bewegt. Rote Linien deuten die Richtung an.

Interessant ist noch der Fall der periodischen Randbedingung, der eine Änderung benötigt. Angenommen es gelten periodische Randbedingungen in x-Richtung (y-Richtung gilt analog), existieren links und rechts keine festen Grenzen, sondern ein fortlaufendes Feld. Für ein Partikel bedeutet das also, dass wenn es links über Grenzen hinausgeht, tritt es rechts wieder ein. Die Berechnung von x^{old} erfolgt zunächst auf bereits bekanntem Wege. Befindet sich x^{old} anschließend außerhalb des *sichtbaren* Feldes, müssen so lange „ganze Felder“ übersprungen werden, bis sich x^{old} wieder innerhalb eines sichtbaren Feldes befindet. Eine entsprechende Berechnung muss dieses Verhalten berücksichtigen:

$$x^{old} = (xsize + (x^{old} \mod xsize)) \mod xsize$$

Der Modulo-Operator sorgt für das „überspringen“ und stellt sicher, dass x^{old} im sichtbaren Bereich ist. In Abbildung 3.3

Funktion „project“

Zuletzt müssen die erhaltenen Felder noch von Divergenz befreit werden. Dazu wird noch einmal auf die *Helmholtz-Hodge Decomposition* zurückgegriffen. Jedes Vektorfeld kann aufgeteilt werden in ein divergenzfreies Feld und in ein Gradientenfeld Gleichung (2.3). Auf das Geschwindigkeitsfeld bezogen bedeutet das, dass ein divergenzfreies Feld durch Subtraktion des Gradientenfeldes vom aktuellen Geschwindigkeitsfeld berechnet werden kann. Ziel ist es also, zunächst das entsprechende Gradientenfeld zu berechnen. Das Ganze kommt der Berechnung eines Höhenfelds gleich. Zuerst wird dafür die Divergenz des Geschwindigkeitsfeldes gebildet [Sta03]:

$$div_{x,y} = -\frac{1}{2}\left(\frac{(u_{x+1,y} - u_{x-1,y})}{xsize} + \frac{(v_{x,y+1} - v_{x,y-1})}{ysize}\right)$$

Mit Hilfe eines neuen (Hilfs-)Felds p und des Gauss-Seidel-Verfahrens, kann anschließend die Poisson-Gleichung gelöst und das Höhenfeld berechnet werden [Sta03]:

$$p_{x,y} = (div_{x,y} + p_{x+1,y} + p_{x-1,y} + p_{x,y+1} + p_{x,y-1})/4$$

Ausgehend von diesem Höhenfeld kann nun der Gradient des Höhenfelds vom aktuellen Geschwindigkeitsfeld subtrahiert werden [Sta03]:

$$u_{x,y} = u_{x,y} - \frac{1}{2}(p_{x+1,y} - p_{x-1,y}) \cdot xsize$$
$$v_{x,y} = v_{x,y} - \frac{1}{2}(p_{x,y+1} - p_{x,y-1}) \cdot ysize$$

Als Ergebnis folgt ein divergenzfreies Geschwindigkeitsfeld.

Funktion „set_bnd“

Da die Felder in jedem Schritt aktualisiert werden, müssen auch ihre Ränder nach jedem Schritt aktualisiert werden. Dazu wird am Ende jeder Routine *set_bnd* aufgerufen. Das gilt insbesondere für den Projektionsschritt; dort muss *set_bnd* nicht nur am Ende, sondern nach jedem der drei einzelnen Schritte aufgerufen werden. Für eine erste Implementierung sei das Feld mit festen Grenzen umgeben. Die Grenzen liegen auf den durch Verbinden vertikaler und horizontaler Geraden der Grenzpunkte $(0,0),(0,ysize-1),(xsize-1,0) und (xsize-1,ysize-1)$. Für das Geschwindigkeitsfeld müssen zwei Bedingungen gelten. Zum einen muss die vertikale Geschwindigkeit an den horizontalen Grenzen reflektiert werden und zum anderen muss die horizontale Geschwindigkeit an den vertikalen Grenzen reflektiert werden. Das kann leicht erreicht werden [Sta03]:

$$u_{0,y} = -u_{1,y} \text{ respektive } u_{xsize-1,y} = -u_{xsize-2,y}$$
$$v_{x,0} = -v_{x,1} \text{ respektive } v_{x,ysize-1} = -v_{x,ysize-2}$$

Für die verbleibenden Grenzen (horizontale Grenzen für u und vertikale Grenzen für v) werden lediglich die Werte angenommen, die sich vor den Grenzen befinden, z. B. $u_{x,0} = u_{x,1}$, die anderen Fälle sind analog. Die Ecken werden anschließend durch das arithmetische Mittel der anliegenden Randzellen bestimmt [Sta03]. Im späteren Verlauf hinsichtlich der Aurora werden periodische Randbedingungen verwendet. Um das zu erreichen, werden lediglich die Werte der gegenüberliegenden Seite übernommen:

$$u_{0,y} = u_{xsize-1,y} \text{ respektive } u_{xsize-1,y} = u_{1,y}$$
$$v_{x,0} = v_{x,ysize-2} \text{ respektive } v_{x,ysize-1} = v_{x,1}$$

Hier wird nochmal darauf hingewiesen, dass, falls periodische Randbedingungen gelten, die advect-Routine ebenfalls angepasst werden muss. Wie das geschieht, ist im entsprechenden Kapitel vermerkt. Ebenfalls denkbar und einfach zu realisieren sind Kombinationen beider Randbedingungen. Beispielsweise kann durch periodische Randbedingungen in x-Richtung und feste Grenzen oben und unten ein Kanal modelliert werden.

3.2.3 Funktion „dens_step“

Als letztes steht noch die Behandlung des Dichtefelds aus. Aus Algorithmus 3.4 geht bereits hervor, das lediglich ein Advektions- und ein Diffusionsschritt benötigt werden. Da zur Lösung von Gleichung (3.2) die gleichen Vorgehensweisen benötigt werden, können die advect- und diffuse-Routinen einfach wiederverwendet werden. Der diffuse-Routine muss anstatt $visc$ die Diffusionskonstante $diff$ übergeben werden. Die Bedeutungen der Schritte bleiben gleich. In der diffuse-Routine wird die Dichte verteilt und in der advect-Routine wird die Dichte entsprechend dem Geschwindigkeitsfeld bewegt. Die Behandlung der Ränder gilt analog zum *vel_step*.
Mit der Fertigstellung dieses Schritts, ist die Simulation fertig und einsatzbereit.

3.3 GPU Implementierung

Auch wenn die CPU-Implementierung der Simulation funktionsfähig ist, ist sie ohne zusätzlichen Aufwand nicht praktikabel für Echtzeit-Animation. Deswegen wurde die Simulation zusätzlich für die GPU-Nutzung implementiert. Das erlaubt der Simulation die einzelnen Punkte der Felder parallel zu berechnen. Da sich an der Berechnung der Simulation grundlegend nichts ändert, ist eine Übertragung auf die GPU relativ einfach möglich.
Als API wurde dafür OpenGL gewählt. Anstatt std::vector für die Felder zu nutzen, werden die Felder nun mittels *Texturen* repräsentiert, in denen einfache *floats* gespeichert werden. Die Größe der Texturen entspricht der Größe des Feldes.
Um nun die Berechnungen auf der GPU auszuführen, werden für jeden Schritt einzelne *Compute Shader* angelegt und nacheinander ausgeführt. Über passende *uniforms* können benötigte Konstanten (z. B. die Diffusionskonstante) an die jeweiligen Shader übergeben werden.
Problematisch bei der Übertragung auf die GPU ist die Umsetzung des Gauss-Seidel-Verfahrens. Wohingegen auf der CPU jeder Punkt des Feldes sequentiell abgearbeitet wird und damit für jeden nächsten Punkt teilweise die bereits aktualisierten Werte vorliegen, werden die Punkte auf der GPU parallel verarbeitet. Das hat zur Folge, das für die vier Nachbarzellen ausschließlich die Werte des alten Feldes vorliegen und keinerlei Aktualisierungen während einer einzelnen Iteration mit einbezogen werden. Damit auch während einer Iteration bereits neu berechnete Werte verwendet werden, wird sich einer Methode bedient, das als *Schwarz-Weiß-Verfahren* (oder auch *Schwarz-Rot-Verfahren*) bekannt ist. Abbildung 3.8 illustriert das Vorgehen des Verfahrens. Das Prinzip ist, das Feld in zwei Teile aufzuteilen und zuerst über diejenigen Punkte zu iterieren, deren Index gerade ist und im Anschluss über die ungeraden Indizes. Jede Iteration des Gauß-Seidel-Verfahrens besteht also zuerst aus der Aktualisierung der geraden und schließlich der ungeraden Indizes.

3.3.1 Erweiterung auf 3D

Für eine 3D-Simulation der Aurora wird zwar keine komplette 3D-Fluidsimulation benötigt, jedoch soll hier trotzdem kurz besprochen werden, wie die aktuelle Simulation auf eine dritte Dimension erweitert werden kann.
Die Erweiterung der allgemeinen Fluidsimulation auf 3D stellt kein Problem dar, da für eine entsprechende 3D-Implementierung das Gründgerüst bereits vorliegt. Hier sei daran erinnert, das bei der CPU-Implementierung z ein konstanter Wert zugewiesen wurde und dass der Vektorzugriff

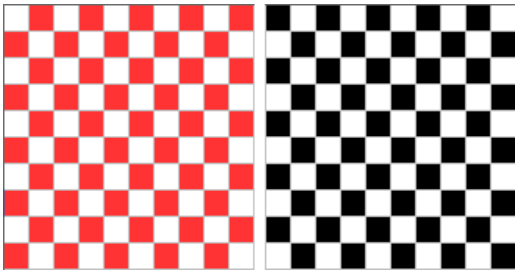

Abbildung 3.4: Rot-Schwarz-Verfahren.

über ein Makro $IX(x, y, z)$ geschieht, das bereits Indizes für den 3D-Fall ausgibt. Zu beachten gilt es also nur noch eine zusätzliche Schleife für z einzurichten, die Behandlung der Ränder für eine zusätzliche Dimension zu erweitern - analog zu x und y - und die an den entsprechenden Stellen in Abschnitt 3.2 erwähnten Erweiterungen der Berechnungen einzubauen. Im Falle der Aurora wird dafür eine 3D-Textur verwendet, die zusammen mit einer 2D-Textur, in der sich das aktuelle Simulationsergebnis befindet, an einen zusätzlichen Shader geschickt wird. Dort wird der Inhalt der 2D-Textur in die unterste Ebene der 3D-Textur übertragen und anschließend wird noch eine Höhe hinzgefügt.
Eine komplette 3D-Simulation der Aurora besteht also aus einer 2D-Fluidsimulation für die Grundform und einer Funktion, die jedem Punkt der Aurora eine „Höhe“ zuweist. Wie genau das geschieht, wird im nachfolgenden Kapitel erläutert.

3.3.2 Vertikale Absetzung

In Abschnitt 2.1.6 wurde bereits mit Hilfe zweier Modelle erklärt, wie der Aurora Höhe verliehen wird. Wie im genannten Kapitel zu erkennen ist, werden dafür einige Parameter benötigt. An dieser Stelle wird explizit auf [1] verwiesen, mit Hilfe derer die Daten für das MSIS-E-90-Model bestimmt wurden. Aus dem MSIS-E-90-Model kann nun die Dichte der Atmosphäre für eine Höhe zwischen 80 km und 380 km bestimmt werden. Dafür wurde ein Ort im südlichen Schweden gewählt, der sich bei 60° nördlicher Breite und 15° östlicher Länge befindet. Aus der Dichte wird nun die *atmosphere shielding mass* M_z berechnet. Liegen alle relevanten Eingabegrößen vor, kann mit Hilfe des Models von Lazarev die *aurora energy deposition rates* für Partikelenergien im Bereich von 1-30keV berechnet und in einer zusätzlichen Textur gespeichert werden.
Liegt nun bereits eine 2D-Simulation vor, die in einer 3D-Texturebene gespeichert ist, kann diese nun leicht in die Höhe gezogen werden. Der Abstand zweier aufeinanderfolgender Indizes entspricht dabei einem Höhenunterschied von $2km$.

3.4 Rendern der Simulation

Nachdem die Aurora nun vollständig simuliert werden kann, fehlt noch der visuelle Aspekt. Die Visualisierung erfolgt zu Anfang ebenfalls mit OpenGL.

[1] https://ccmc.gsfc.nasa.gov/modelweb/models/msis_vitmo.php
[2] Entstammt aus der Vorlesung Scientific Visualization der Universität Stuttgart.

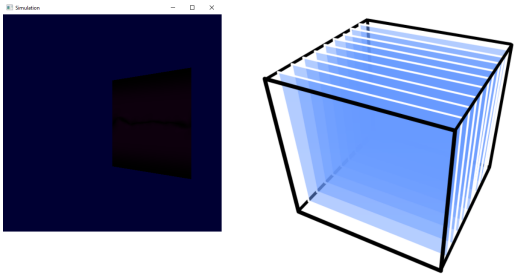

Abbildung 3.5: Links: OpenGL Fensterkontext mit transformiertem gerenderten Quad. Rechts: Texturebasiertes Volumen-Rendering [2].

Um mit OpenGL zu rendern, wird ein simples Standard-Setup verwendet. Das Setup besteht aus einem Kontext für das Fenster, in dem die Ausgabe auf dem Bildschirm angezeigt wird (ein Beispiel ist in Abbildung 3.5 zu sehen), einem Vertex- und einem Fragment-Shader. Dem Vertex-Shader werden die Informationen für die Geometrie, die gerendet werden soll und die Texturkoordinaten, die für den Zugriff auf die Textur notwendig sind, und die dazugehörigen Transformationsmatrix übermittelt. Anschließend werden die Texturkoordinaten (indirekt) vom Vertex-Shader an den Fragment-Shader weitergeleitet. Dem Fragment-Shader wird außerdem direkt die Textur übermittelt, die angezeigt werden soll. Im Fragment-Shader wird nun jedem Punkt der zu rendernden Geometrie ein Farbwert zugewiesen. Die Farbwerte können direkt mit Hilfe der Texturkoordinaten aus der Textur ausgelesen und den entsprechenden Punkten zugewiesen werden. Die Textur wird im Prinzip über die Geometrie gelegt.
Für die 2D-Simulation wird dafür ein Quadrat (engl. quad) verwendet. Ein Quad besteht aus zwei Dreiecken, die wiederum aus jeweils drei Vertizes bestehen. Daraus kann eine Liste von sechs Vertizes erstellt und an den Vertex-Shader geschickt werden, die ebenfalls als Texturkoordinaten verwendet werden können (es kann aber auch eine separate Liste mit Texturkoordinaten an den Shader übermittelt werden). Im Anschluss werden sowohl die Texturkoordinaten als auch die Textur, die auf den Quad gerendert werden soll - wie zum Beispiel das Dichtefeld - an den Fragment-Shader gesendet und dieser „erzeugt“ die endgültige Ausgabe. Bei den Texturkoordinaten gilt es zu beachten, das die Koordinaten auch den passenden Punkten zugewiesen wird. Das heißt, der linken unteren Ecke (0,0) des Quads muss auch der Texturwert der linken unteren Ecke der Textur zugewiesen werden. Prinzipiell ist es aber auch möglich, jedem Punkt des Quads einen beliebigen Wert der Textur zuzuweisen.
Für eine 3D-Simulation und damit das Rendern einer 3D-Textur kann das gleiche Vorgehen verwendet werden, nur muss dieses noch etwas erweitert werden. Sei die zu rendernde Geometrie nun kein Quadrat mehr, sondern ein Würfel (oder ein Quader). Ein Würfel kann nun unterteilt werden in einzelne Ebenen (oder auch *slice* genannt), wobei jede Ebene ebenfalls als Quad dargestellt wird, siehe Abbildung 3.5 rechtes Bild. Sei die verwendete Geometrie bspw. ein 4x4x4 Würfel, so kann der Würfel in vier separate Ebenen unterteilt werden. Um also einen Würfel zu rendern, genügt es seine einzelnen Ebenen zu rendern. Es bedarf deshalb lediglich einer Schleife über die Höhe, in dem bei jedem Schleifendurchlauf die Ebene der entsprechenden Höhe gerendert wird. Die Unterteilung der 3D-Textur in einzelne Ebenen erfolgt gänzlich analog. Jeder Ebene des Würfels wird dementsprechend die Ebene der 3D-Textur der selben Höhe zugewiesen. Dieses Verfahren ist auch als texturbasiertes Volumen-Rendering oder Slice-Rendering bekannt. Hinsichtlich der Echtzeit-Animation der Aurora ist dieses Verfahren allerdings ungeeignet, da es lange dauert, die einzelnen Ebenen zu rendern.

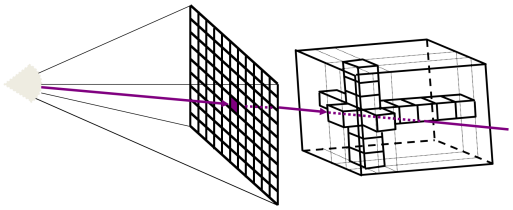

Abbildung 3.6: Prinzip des Raycasting[3].

Es existieren jedoch raffiniertere 3D-Rendering-Techniken als das simple wiederholte Rendern eines Quads, auf das eine Textur gelegt wird. Eine für die Aurora verwendete Technik nennt sich *Raycasting* und ist vor allem im Kontext der wissenschaftlichen Volumenvisualisierung geläufig. Raycasting findet auch häufig in der Medizin Anwendung, wenn es beispielsweise darum geht Daten eines MRT-Scans zu visualisieren. Allgemein eignet sich Raycasting auch um gasähnliche Effekte zu rendern und ist damit gut für die Visualisierung eines semi-transparenten Mediums wie der Aurora geeignet. Es wird im Folgenden lediglich das Grundprinzip des Raycasting erläutert. Weitere Information zum Thema Volume-Rendering im Allgemeinen und Raycasting im speziellen, kann in passender Fachliteratur gefunden werden, die dieses Thema detailliert behandeln. Sei nun wieder der Würfel die Geometrie, die das Volumen repräsentiert. Anstatt den Würfel in einzelne Ebenen zu unterteilen, wie es beim vorherigen Vorgehen der Fall war, wird der Würfel als gesamtes Volumen betrachtet. Um nun ein Bild aus den Volumendaten (repräsentiert durch eine 3D-Textur) zu rendern, benötigt es noch einen Betrachter bzw. eine Kamera, die vor der Bildwand positioniert ist und in Richtung des Volumens schaut, das sich hinter der Bildwand befindet. Abbildung 3.6 illustriert die gesamte Konstellation. Vom Betrachter aus werden nun Strahlen durch die Mitte jedes einzelnen Pixels der Bildwand geschickt. Trifft ein Strahl das Volumen, wird der Strahl durch das gesamte Volumen verfolgt. Entlang des Strahls werden nun die Werte jedes getroffenen Volumenelements (Voxel) betrachtet und auf bestimmte Weise miteinander verrechnet. Es gilt dabei eine Volumen-Rendering-Gleichung anzunähern, die unter anderem Emission und Absorption des Lichts berücksichtigt. Auf Details zur Rendering-Gleichung und genauen Berechnung wird hier nicht weiter eingegangen. Der hier verwendete Raycaster wird allerdings nicht als eigenständiger Renderer innerhalb eines einzelnen Programms mit der Simulation genutzt, sondern ist Teil des Frameworks MegaMol, mit dem sich das nächste Kapitel auseinandersetzt.

3.5 MegaMol

Ein Teil dieser Arbeit beschäftigt sich mit dem Visualisierungs-Framework MegaMol [Gro+15], das in Zusammenarbeit des Visualisierungsinstituts der Universität Stuttgart und dem Institut für Computergrafik und Visualisierung der Technischen Universität Dresden entstanden ist. MegaMol wurde entwickelt, um der Nachfrage nach High-Performance, GPU-beschleunigtem Rendering von Partikeldaten nachzukommen und darüber hinaus verschiedene Visualisierungs-Anwendungen umzusetzen. Partikelbasierte Datenmengen bestehen größtenteils aus dem gleichen Typ von Daten: Atome, Moleküle oder makroskopische Partikel. Je nach Anwendungsfall, Partikelmenge und den

[3] Entstammt aus der Vorlesung Scientific Visualization der Universität Stuttgart.

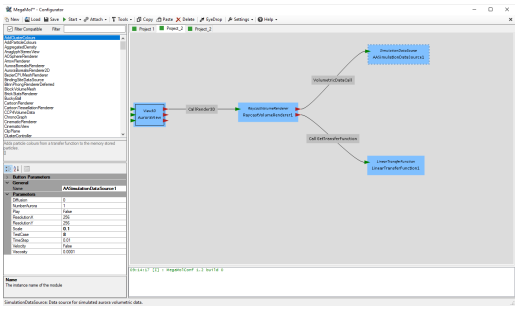

Abbildung 3.7: MegaMol-Konfigurator mit dem Modulgraph für die Simulation und Visualisierung der Aurora.

zu untersuchenden Aspekten werden zur Visualisierungsanalyse allerdings verschiedene visuelle Modelle und Analysefunktionen benötigt. MegaMol soll hierfür als eine generelle Anlaufstelle dienen, die flexibel unterschiedliche Visualisierungsmöglichkeiten bietet und gleichzeitig die hohe Rechenleistung der GPU mit unter anderem der OpenGL-API ausnutzt. Ein weiterer Fokus des Frameworks liegt bei der Wiederverwendbarkeit bereits existierender Methoden [Gro+15].
Der Aufbau von MegaMol besteht aus zwei Teilen, einer oberen Ebene bestehend aus einer Schnittstelle zwischen den einzelnen Plugins und ihren ausführbaren Dateien (executables) und DLLs (dynamic linked libraries) und einer unteren Ebene, die die Anwendungslogik einzelner Komponenten wie Modulen und Aufrufen beinhaltet. In den verschiedenen Plugins befindet sich die eigentliche Implementierung einzelner Anwendungen und können über die dazugehörigen Module aufgerufen werden. Module können einzeln ausgeführt werden, zum Beispiel lässt sich die Simulation und Visualisierung der Aurora mit einem einzelnen Plugin umsetzen. Module können aber auch über Aufrufe mit anderen Modulen verknüpft werden. Ein Modul könnte zum Beispiel die Simulation und ein weiteres Modul könnte die Visualisierung der Aurora beinhalten.
Abbildung 3.7 zeigt die Benutzerschnittstelle von MegaMol mit dem Modulgraphen für die Simulation und Visualisierung der Aurora. Das Modul View3D dient als Einstiegspunkt und ist für die Kameraansicht zuständig. Der Aufruf CallRender3D verknüpft das Modul View3D mit dem Renderer und teilt dem Programm mit, das ein Ausgabebild erzeugt werden soll. Das Modul SimulationDataSource ist ein für diese Arbeit neu erstelltes Modul und beinhaltet die Simulation der Aurora und übermittelt mit dem Aufruf VolumetricDataCall das Handle für die 3D-Textur an den Raycaster, der anschließend das Ausgabebild rendert. Ein Vorteil von MegaMol ist die bereitgestellte Schnittstelle für Parameter. Unten links in Abbildung 3.7 ist die Auswahl verschiedener Parameter zu sehen, die während der Simulation geändert werden können und somit eine direkte Interaktion mit der Simulation erlauben.

Es ist auch möglich Plugininterne Renderer zu verwenden, zum Beispiel solche wie sie in Abschnitt 3.4 vorgestellt wurden. Dafür bedarf es lediglich die Erstellung eines weiteren Moduls, das sowohl die Simulation als auch die Visualisierung enthält und mittels eines CallRender3D Aufruf an das View3D-Modul gekoppelt ist. Ein entsprechender Modulgraph ist in Abbildung 3.8 zu sehen.

Abbildung 3.8: Das Modul AuroraBorealisRenderer beinhaltet sowohl die Simulation als auch das Rendering.

4 Ergebnisse

Um die Fluidsimulation nun auch zu testen, wurden verschiedene Szenarien implementiert und simuliert. Als erstes wurde das Strömungsverhalten für verschiedene Reynoldzahlen getestet. Wie bereits im Abschnitt 2.3 erwähnt, gehen laminare Strömungen für $Re > 2300$ in turbulente Strömungen über. Um dieses Verhalten zu überprüfen, wurde eine Szene mit einer konstanten Strömung von links nach rechts erstellt. Abbildung 4.1 zeigt die Ergebnisse für unterschiedliche Reynoldzahlen.

Als nächstes wurde ein Testszenario für die Poiseuille-Strömung oder auch Hagen-Poiseuille-Strömung erstellt. Die Poiseuille-Strömung trifft eine Aussage über das Verhalten einer Laminarströmung eines inkompressiblen Newtonschen Fluids innerhalb eines Rohres. Laut dem Gesetz von Hagen-Poiseuille sind die Strömungsgeschwindigkeiten eines Fluid mit den genannten Eigenschaften in der Mitte des Rohres am schnellsten und werden nach außen immer geringer. Dafür wurde ein Geschwindigkeitsfeld simuliert, das periodisch in y-Richtung ist und eine feste linke und rechte Grenze hat. Jedem Punkt des Felds wurde dafür am Anfang der gleiche Geschwindigkeitswert zugewiesen und anschließend durchlief die Simulation 2000 Iterationen. Das Ergebnis in Abbildung 4.2 entspricht genau dem erwarteten Verhalten.

Als nächstes folgte die Simulation mit einem konstanten Strom in der Mitte der linken Seite, der auf ein festes Objekt trifft, ähnlich wie bei der Kármánschen Wirbelstraße. Bei der Kármánschen Wirbelstraße entstehen hinter dem Objekt gegenläufige Wirbel. Um die Wirbelstraße zu simulieren, wird eine weitere Technik benötigt, die unter anderem als *Wirbelvektor-Einschränkung* bekannt ist, die in dieser Arbeit zwar implementiert wurde, aber nicht ausreichend getestet wurde und deshalb auch nicht detailliert besprochen wird. Das Ziel bei der Wirbelvektor-Einschränkung ist, Wirbel länger am Leben zu halten, in dem externe Kräfte orthogonal zu den Vektoren addiert werden, die die Wirbel bilden.

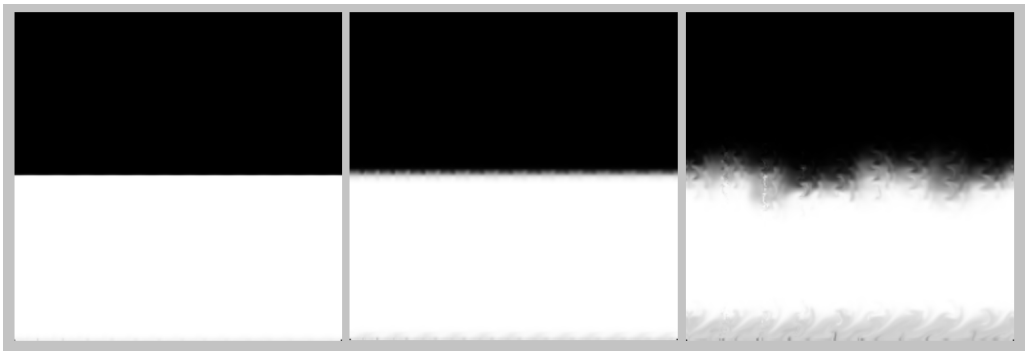

Abbildung 4.1: Steigende Reynoldzahl liefert Übergang von laminarer zu turbulenter Strömung. Links: $Re = 2170$, laminarer Strom. Mitte: $Re = 2500$, leichte Turbulenzen. Rechts: $Re = 6670$, starke Turbulenzen.

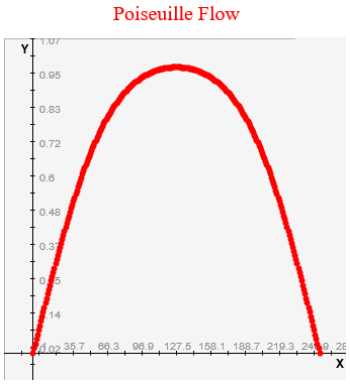

Abbildung 4.2: Querschnitt in der Mitte des Geschwindigkeitsfelds nach 2000 Iterationen, mit relativer Geschwindigkeit für die y-Achse.

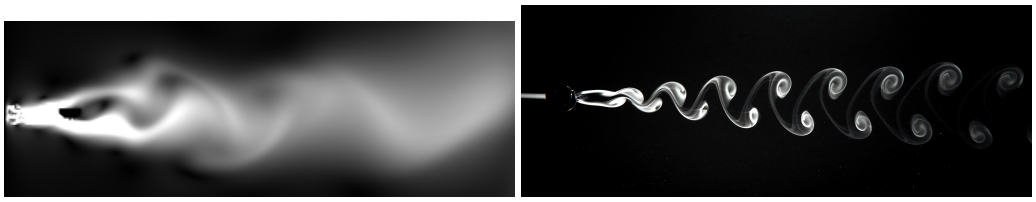

Abbildung 4.3: Links: Konstanter Strom, der auf ein Objekt trifft. Rechts: Kármánsche Wirbelstraße.

Als letztes wurden die Störungen der Kelvin-Helmholtz-Instabilität simuliert. Grenzen zwei Fluide mit entgegengerichteten Geschwindigkeiten aneinander, entstehen in der Grenzschicht, auch Scherschicht genannt, Verwirbelungen. Dieses Verhalten konnte mit der verwendeten Fluidsimulation sehr gut nachgestellt werden. Abbildung 4.4 zeigt die entstandenen Verwirbelungen der Scherschicht. Der obere Teil des Feldes zeigt in die positive x-Richtung, der untere Teil läuft in entgegengesetzter Richtung.

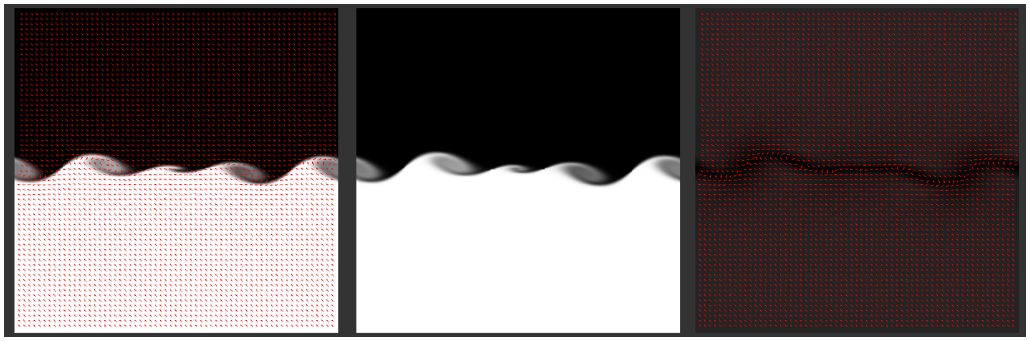

Abbildung 4.4: Verwirbelungen der Kelvin-Helmholtz-Instabilität mit (links) und ohne Richtungsvektoren (mitte) des Geschwindigkeitsfeldes. Rechts: Verwirbelungen durch Richtungsvektoren des Geschwindigkeitsfeldes dargestellt.

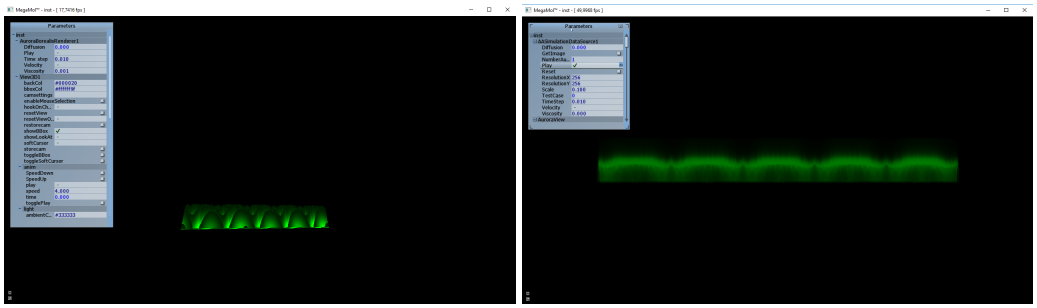

Abbildung 4.5: Gegenüberstellung des Slice-Renderers (links) und des Raycasters (rechts).

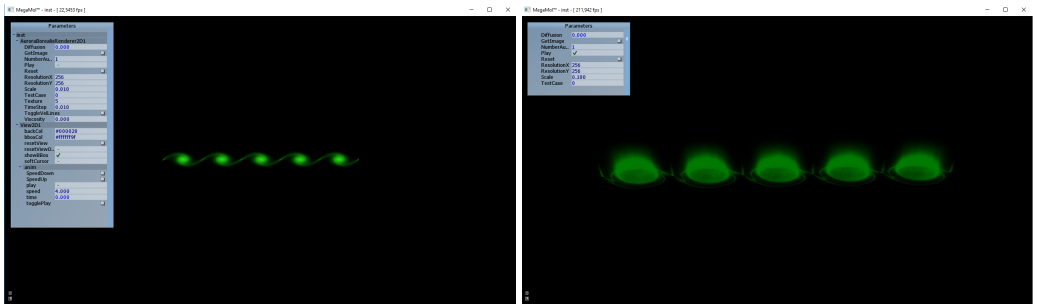

Abbildung 4.6: 2D-Aurora mit Verwirbelungen und die dazugehörige 3D-Visualisierung.

4.1 Simulation und Rendering der Aurora

Mit der erfolgreichen Simulation der Kelvin-Helmholtz-Verwirbelungen kann eine 3D-Aurora simuliert werden. Das Standard-Setup sei dafür wie folgt: für das Geschwindigkeitsfeld **u** sind die Geschwindigkeiten in der oberen Hälfte in positive und in der unteren Hälfte in negative x-Richtung gerichtet. Das Geschwindigkeitsfeld **v** setzt sich aus einer nach unten gerichteten Geschwindigkeit für die obere und eine nach oben gerichtete Geschwindigkeit in der unteren Hälfte. In das Dichtefeld wurde entlang der Mitte eine sinusartige Kurve gelegt. Für den Fall von mehreren Auroren in einer Simulation, sind die Richtungen in **u** in den Zwischenräumen der Auroren jeweils entgegengesetzt. Für die anfänglichen 3D-Simulationen wurde ein Slice-Renderer verwendet, der später dann mit der Verwendung von MegaMol durch einen Raycaster ersetzt wurde, Abbildung 4.5 zeigt die deutlichen Unterschiede in der Darstellung beider Renderer.

Die ersten Tests wurden in 2- und 3D ausgeführt, um die Grundform zu bestimmen und Effekte wie Wirbel und Faltungen zu erzeugen. Die sinusartige Kurve entlang der Mitte des Dichtefeldes nimmt dabei wirbelartige Formen an, siehe Abbildung 4.6. Die Verwirbelungen der Aurora sind sehr deutlich zu sehen. Allerdings ist auch zu erkennen, dass sich die einzelnen Verwirbelungen voneinander trennen. Diese Eigenschaft ist im Allgemeinen nicht wünschenswert, da so kein typisches Aurora-Band erzeugt werden kann.

Eine weitere Eigenschaft, die die Simulation aufweist, sind große Wirbel. Die Aurora beginnt mit kleinen Verwirbelungen, die sich dann zusammenschließen und zu einem großen Wirbel werden. Abbildung 4.7 zeigt die einzelnen Phasen dieses Verhalten.

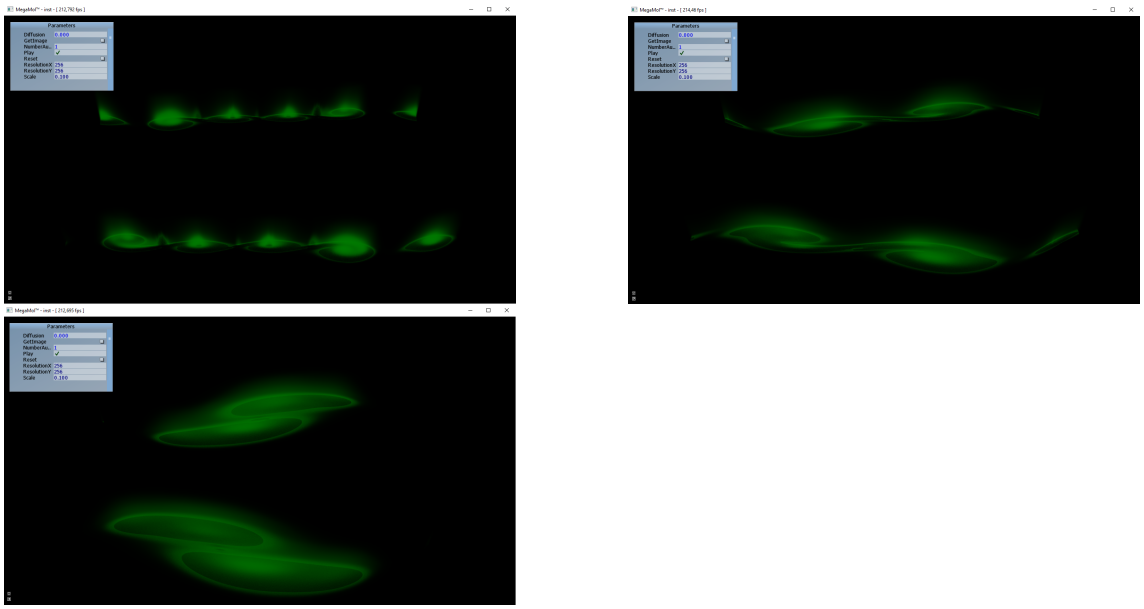

Abbildung 4.7: Hier wurden zwei Aurora-Bänder gleichzeitig simuliert. Bei beiden Bändern sind die immer größer werdenden Verwirbelungen deutlich zu sehen (von links-oben nach rechts-unten).

Das beste Ergebnis liefert eine Simulation, bei der fünf Auroren gleichzeitig simuliert wurden. Abbildung 4.8 zeigt die Entwicklung dieser Simulation über eine Dauer von circa 37s. In diesem Szenario sind sowohl Verwirbelungen als auch Faltungen gut zu erkennen. In der letzten Phase sind allerdings wieder die großen Verwirbelungen und das Auseinandergehen der Auroren zu sehen.

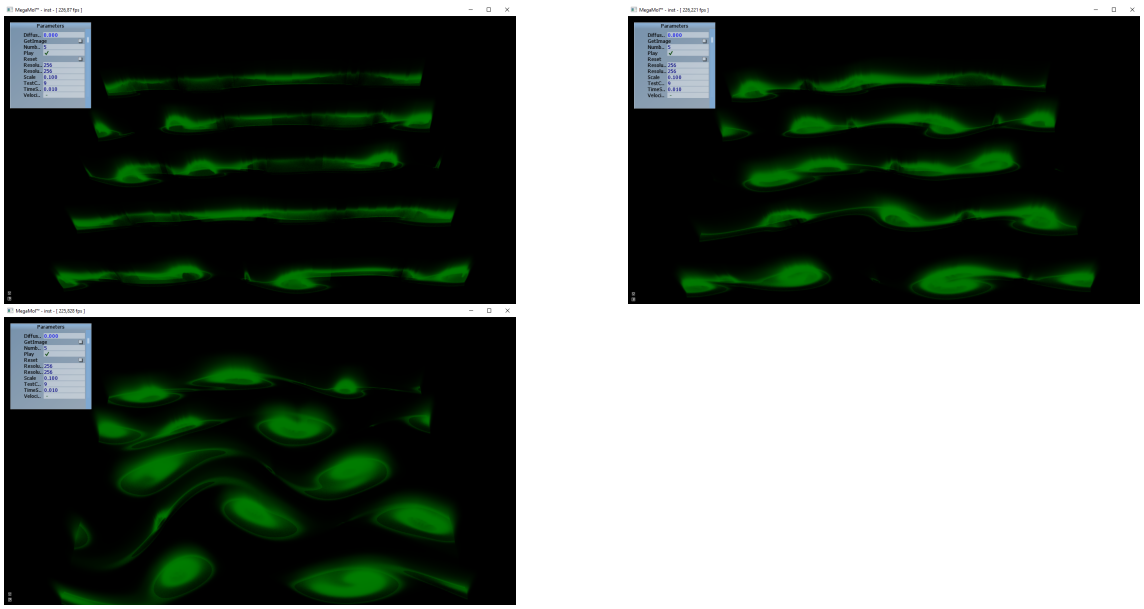

Abbildung 4.8: Simulation von fünf Auroren gleichzeitig (von links-oben nach rechts-unten). Verwirbelungen, Faltungen und das Auseinandergehen sind deutlich zu erkennen.

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden zu Beginn die Grundlagen auf Basis der Navier-Stokes-Gleichungen für eine Fluidsimulation geschaffen, mit Hilfe derer gängige Probleme der Fluidmechanik, wie Verwirbelungen im Sinne der Kelvin-Helmholtz-Instabilität, die einen Teil zur Form der Aurora beitragen, effizient und visuell ansprechend simuliert werden können. Anschließend wurde für das Lösungsverfahren ein Programm erstellt, das die Simulation auf der CPU realisiert. Wegen der schlechten Performance auf der CPU, wurde eine Implementierung für die GPU vorgenommen. Durch die hohe Parallelisierbarkeit der Berechnungen auf Grafikprozessoren, kann die Simulation effizient auf der GPU ausgeführt werden und öffnet den Zugang zu Echtzeit-3D-Animationen der Aurora. Im Anschluss an die Implementierung der Simulation, wurde zuerst Slice-Rendering vorgestellt, das wegen seiner schlechten Performance ungeeignet ist für Echtzeit-Animation. Daraufhin wurde Volumen-Rendering mittels Raycaster vorgestellt. Der Raycaster bietet eine sehr gute Möglichkeit, die 3D-Aurora zu rendern und visuell ansprechend darzustellen. Eigenschaften wie Transparenz und strahlenartige Effekte der Aurora konnten damit gut umgesetzt werden. Des Weiteren wurde MegaMol vorgestellt, ein effizientes Framework, mit dem sich die Simulation und ein effizientes Volumen-Rendering einfach zu einem Plugin vereinen lassen.
Ein Problem der Animation ist, die Aurora stabil in einem Band zu halten. Bei der bisherigen Animation treten zwar sehr gute Verwirbelungseffekte auf, allerdings „teilt“ sich die Aurora während der Simulation und bleibt nicht zusammenhängend. Um dieses Problem zu lösen, ist es denkbar, ein Kraftfeld anzulegen, so dass in jedem Simulationsschritt eine Kraft auf das Geschwindigkeitsfeld wirkt. Ebenfalls nur schwer zu realisieren, sind Faltungen, die der Aurora ihre typische Form verleihen. Die Ergebnisse der Simulation und die Laufzeit der Implementierung haben gezeigt, dass es durchaus möglich ist, 3D-Echtzeit-Animationen einer Aurora zu simulieren, die zudem einen angemessenen Grad an Realismus zeigen.
Hinsichtlich der korrekten Darstellung komplexer Effekte der Aurora kann noch viel verbessert werden. Die Aurora stabil in einem Band zu halten und gleichzeitig Faltungen zu erreichen, gehört zu den Herausforderungen, die es noch zu lösen gilt. Im Bezug auf längere und konstantere Verwirbelungen können ebenfalls noch Verbesserungen erreicht werden, in dem die Wirbelvektor-Einschränkung mit in die Simulation aufgenommen wird. Zukünftige Arbeiten könnten Fluidsimulationen außerdem mit der Lattice-Boltzmann-Methode, einer numerischen Strömungssimulation, untersuchen. Für dieses Verfahren existieren bereits Bibliotheken und Frameworks, wie waLBerla oder OpenLB. Des Weiteren könnte die Visualisierung so erweitert werden, dass die Aurora in einem Terrain, umgeben von anderer Geometrie, gerendert werden kann.

Literaturverzeichnis

[Bar+00] G. V. Baranoski, J. G. Rokne, P. Shirley, T. Trondsen, R. Bastos. „Simulating the aurora borealis“. In: *Computer Graphics and Applications, 2000. Proceedings. The Eighth Pacific Conference on*. IEEE. 2000, S. 2–432 (zitiert auf S. 13–17).

[CM10] D. Causon, C. Mingham. *Introductory finite difference methods for PDEs*. Bookboon, 2010 (zitiert auf S. 30).

[Gro+15] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl. „MegaMol—a prototyping framework for particle-based visualization“. In: *IEEE transactions on visualization and computer graphics* 21.2 (2015), S. 201–214 (zitiert auf S. 38, 39).

[LG11] O. S. Lawlor, J. Genetti. „Interactive volume rendering aurora on the GPU“. In: (2011) (zitiert auf S. 17, 18).

[Sta03] J. Stam. „Real-time fluid dynamics for games“. In: *Proceedings of the game developer conference*. Bd. 18. 2003, S. 25 (zitiert auf S. 27–29, 31–34).

[Sta99] J. Stam. „Stable fluids“. In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1999, S. 121–128 (zitiert auf S. 19–23).

[WB00] A. Wiegmann, K. P. Bube. „The explicit-jump immersed interface method: finite difference methods for PDEs with piecewise smooth solutions“. In: *SIAM Journal on Numerical Analysis* 37.3 (2000), S. 827–862 (zitiert auf S. 30).

Alle URLs wurden zuletzt am 24. 10. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift