Diplomarbeit

Zweiphasige Navier-Stokes Fluidsimulationen in Maya: Konfiguration, Visualisierung und Animation

vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät der
Rheinischen Friedrich-Wilhelms-Universität Bonn

angefertigt am
Institut für Numerische Simulation

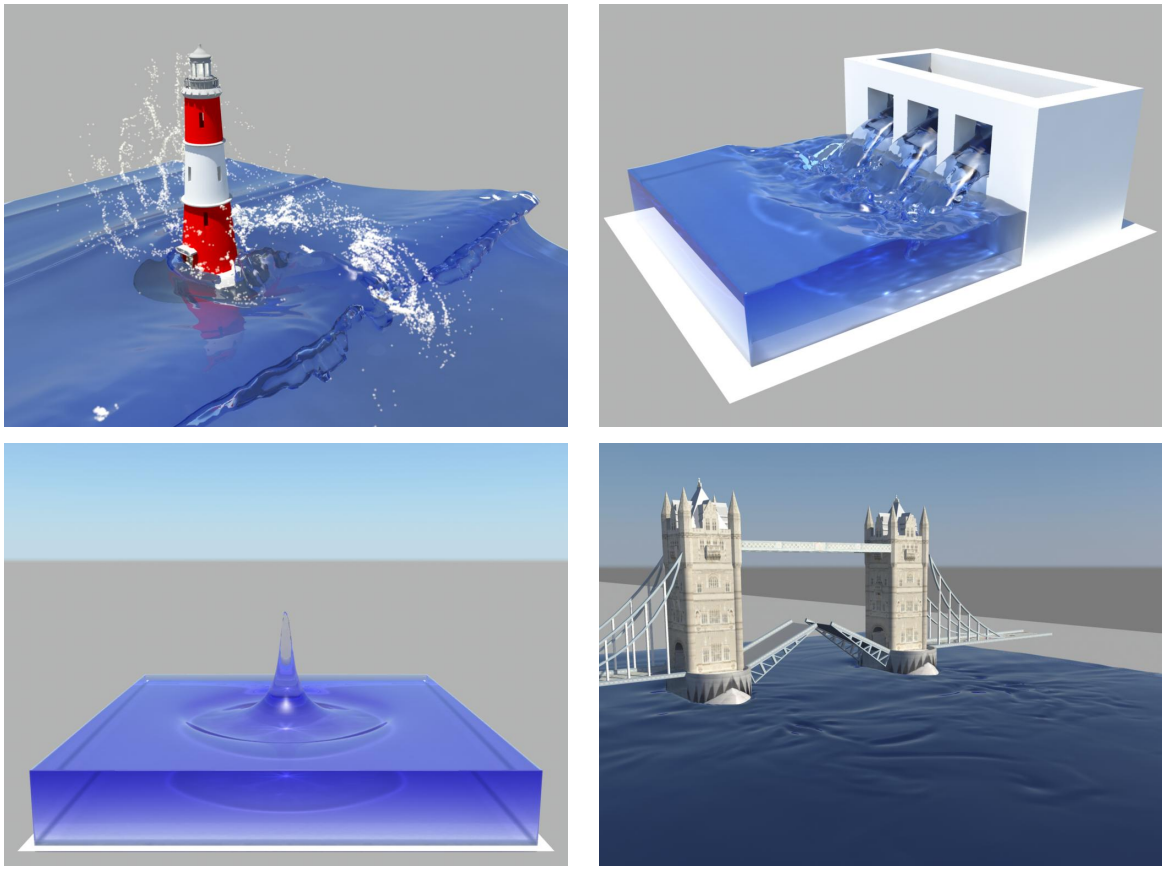
April 2009

von
Peter Zaspel

aus Siegburg

Diplomarbeit

Zweiphasige Navier-Stokes Fluidsimulationen in Maya: Konfiguration, Visualisierung und Animation



von

Peter Zaspel

aus Siegburg

Erklärung

Mit der Abgabe der Diplomarbeit versichere ich gemäß §20 Absatz 6 der Diplomprüfungsordnung vom 14. März 2003, dass ich die Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Bonn, den 27. April 2009

Inhaltsverzeichnis

1 Einleitung

Die numerische Simulation von Fluiden, beispielsweise Luft und Wasser, ist in den letzten Jahren mehr und mehr zu einem Standardwerkzeug in den Natur- und Ingenieurswissenschaften geworden. Beflügelt wird diese Entwicklung durch die stetige Leistungsverbesserung moderner Rechnersysteme. Immer genauere numerische Approximationen lassen sich berechnen, so dass immer realistischere Simulationsergebnisse erzielt werden können.

Schon längst ist der Einsatz von Simulationstechnik als ein *virtuelles Labor* keine reine Fiktion mehr. Bei vielen Anwendungen ist ein Versuchsaufbau zu kostspielig oder gar zu gefährlich. Aber auch beim Einsatz von Laborversuchen kommt es zunehmend dazu, dass zusätzliche Simulationen die gemessenen Ergebnisse validieren.

Motivation

Üblicherweise lässt sich das Vorgehen bei einer Simulation in drei Hauptschritte unterteilen: Der erste Schritt besteht in der Modellierung beziehungsweise *Konfiguration* der Simulationsaufgabe. Im Kontext des virtuellen Labors entspricht dies dem Aufbau des Versuchs. Es muss also festgelegt werden, mit welchen Fluiden gearbeitet wird, wie das Versuchsgebiet aussieht, aus welcher Richtung und an welcher Stelle ein Fluid einströmt und wohin es sich bewegt.

Im nächsten Schritt wird die Simulation durchgeführt. Fluidsimulationen beruhen entweder auf Kontinuumsmodellen oder auf diskreten Modellen. Häufig arbeiten diskrete Modelle mit Partikelsystemen. Kontinuumsmodelle beschreiben die Physik des gesamten Simulationsbereiches mittels partieller Differentialgleichungen. Diese werden diskretisiert, also nur noch an endlich vielen Punkten im Raum betrachtet, und anschließend numerisch gelöst. Die *numerische Approximation* ist die eigentliche Simulation und findet zumeist auf großen Rechnersystemen statt.

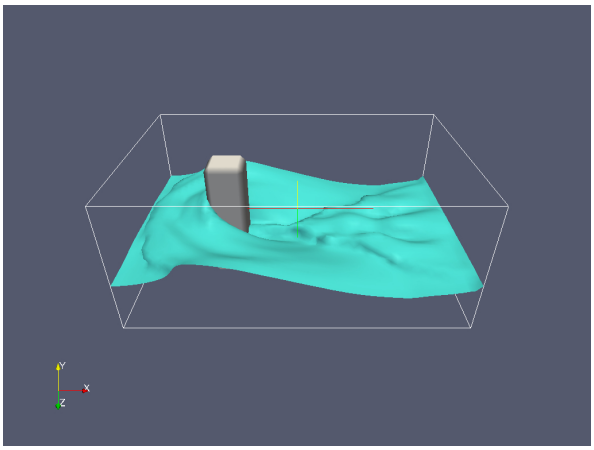
Bei Fluidsimulationen entstehen dreidimensionale, zeitabhängige Daten. Diese erfordern eine erhebliche Menge an Speicherplatz und müssen geeignet verwaltet werden. Eine große Herausforderung stellt die Extraktion und Darstellung der für den Wissenschaftler wesentlichen Daten dar. Hier muss eine Balance zwischen dem Detailgrad der ausgegebenen Informationen und der Einfachheit ihrer Darstellung gefunden werden. Die Extraktion und Darstellung der Simulationsdaten stellt den dritten und letzten Arbeitsschritt im virtuellen Labor dar. Zusammengefasst wird dieser Schritt unter dem Begriff *Visualisierung*.

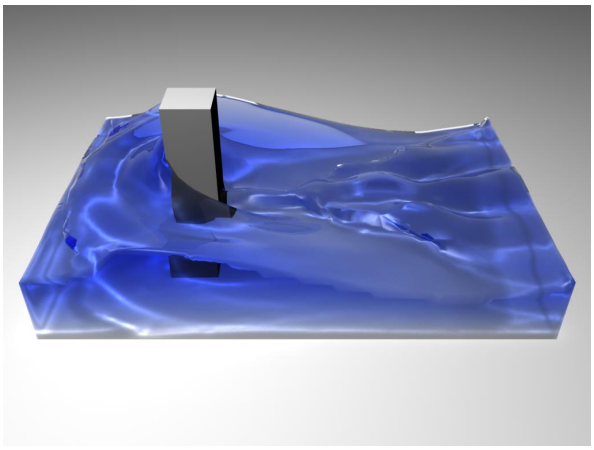
Die vorliegende Arbeit setzt sowohl beim ersten als auch beim dritten Hauptschritt an. Während mathematische Modelle und geeignete numerische Methoden für die Simulation von Fluiden schon lange zum Standardkanon in der Lehre und Forschung zählen, wird der Zusammenstellung der Simulationsinformationen, also der *Konfiguration* der Simulationen nur wenig Beachtung geschenkt. Doch tatsächlich steht und fällt der Einsatz von Simulationswerkzeugen mit ihrer Bedienbarkeit. Solange entsprechende Programme einen zu großen Einarbeitungsaufwand erfordern, wird die Akzeptanz moderner Simulationstechniken in der Anwendung gering bleiben.

Auch das Thema *Visualisierung* wurde lange Zeit vernachlässigt. Erst in den letzten Jahren ist, verbunden mit dem steten Anwachsen der erzeugten Datenmengen, zunehmend dieser Themenkomplex in den Fokus des Interesses gelangt. Dabei muss man zwischen zwei Formen der Visualisierung unterscheiden. Die Ursprungsform, die *wissenschaftliche Visualisierung*, konzentriert sich darauf aus großen Datenmengen für Wissenschaftler leicht zugängliche aber ebenso genaue graphische Darstellungen für die berechneten Ergebnisse zu gewinnen. Häufig muss dabei zwischen der Allgemeinverständlichkeit der Graphiken und dem wissenschaftlichen Anspruch ein Kompromiss eingegangen werden. Das führt dazu, dass Visualisierungen im wissenschaftlichen Kontext meist nur einem Fachpublikum verständlich sind.

In letzter Zeit nimmt das Interesse an *allgemeinverständlichen Darstellungsformen* zu. Ganz offensichtlich müssen neue Wege gefunden werden, einem breiten Publikum wissenschaftliche Arbeit näher zu bringen. Ein deutliches Indiz hierfür sind die Sonderprogramme unter den Titeln *Jahr der Informatik* und *Jahr der Mathematik*, die in den Jahren 2007 und 2008 aufgelegt wurden. Diese hatten zum Ziel, die jeweiligen Themengebiete in den Mittelpunkt des allgemeinen Interesses zu stellen.

Als folgerichtige Konsequenz soll in dieser Arbeit verstärkt der Fokus auf eine allgemeinverständlichere Form der Visualisierung gelegt werden. Eine Unterscheidung zwischen der wissenschaftlichen Visualisierung und der hier betrachteten Form besteht vor allem in dem abweichenden Umgang mit Photorealismus. Während im Kontext der Wissenschaft photorealistische Graphiken eher als ungeeignetes Mittel zur Visualisierung betrachtet werden, dienen sie vor dem Hintergrund der Allgemeinverständlichkeit gerade zur besseren Nachvollziehbarkeit. Festmachen lässt sich dies daran, dass bei Publikationen für die breite Öffentlichkeit zumeist visuell ansprechende Graphiken den Vorzug vor wissenschaftlich nutzbaren Visualisierungen erhalten. Die zwei nachfolgenden Graphiken zeigen sehr deutlich, dass der Einsatz von Photorealismus (rechts) im Gegensatz zum klassischen Ansatz der wissenschaftlichen Visualisierung (links) für die Allgemeinverständlichkeit zu bevorzugen ist.





Bei Strömungssimulationen sollte stets die Qualität der Ergebnisse sehr genau analysiert werden. So gibt es Quellen für Ungenauigkeiten in den Bereichen Modellierung, Diskretisierung und Visualisierung:

Mathematische Modelle machen Annahmen an bestimmte physikalische Gesetzmäßigkeiten. Dies führt zwangsläufig zu einem *Modellfehler*, da die Realität nicht in ihrer Gesamtheit abge-

bildet werden kann. So approximieren die eingangs erwähnten Kontinuumsmodelle makroskopische Fluid-Effekte genauer als die meisten diskreten Partikelmodelle und erreichen damit oft einen geringeren Modellfehler.

Bei der Diskretisierung mathematischer Modelle wird ebenfalls ein Fehler gemacht, da im Allgemeinen endlich viele Datenpunkte nicht den gesamten Informationsgehalt von kontinuierlichen Daten wiedergeben können. Eine Kontrolle des *Diskretisierungsfehlers* ist zum einen durch den Einsatz von Verfahren mit besseren Konvergenzraten und zum anderen durch die Erhöhung der Anzahl der Datenpunkte möglich.

Häufig wird angenommen, dass die Modell- und Diskretisierungsfehler die einzigen beiden Fehlerquellen bei der Simulation einer Strömung sind. Tatsächlich können ebenfalls bei der Visualisierung Ungenauigkeiten entstehen. Die zuvor diskutierte Abwägung zwischen dem Detailgrad der dargestellten Simulationsdaten und ihrer Verständlichkeit führt immer dazu, dass nicht alle Simulationsdaten dargestellt werden können und somit ein *Visualisierungsfehler* gemacht wird.

Zu dieser Arbeit

In der vorliegenden Arbeit wird der dreidimensionale zweiphasige Navier-Stokes–Löser *NaSt-3DGPF* [17], [15] des Instituts für Numerische Simulation der Universität Bonn in das kommerzielle dreidimensionale Graphik- und Animationswerkzeug *Autodesk Maya* [53],[21] eingebettet. Erstmals wird damit ein zweiphasiger, auf dem Kontinuumsmodell der inkompressiblen Navier-Stokes–Gleichungen beruhender Strömungslöser derart mit Maya gekoppelt, dass sich alle zu einer Simulationsaufgabe gehörenden Arbeitsschritte in dem Programm durchführen lassen. Insbesondere kann man in der dreidimensionalen Oberfläche des Graphik-Werkzeuges wie in einem virtuellen Labor beliebige Strömungssimulationen aufbauen. Gleichzeitig finden die ausgezeichneten Rendering-Fähigkeiten von Autodesk Maya Anwendung bei der optisch hochqualitativen Visualisierung der Simulationsresultate.

Der Strömungssimulator NaSt3DGPF ist ein Löser für die inkompressiblen Navier-Stokes–Gleichungen in drei Dimensionen. Mit dem Softwarepaket kann die Interaktion von zwei verschiedenen Fluiden simuliert werden. Zur Beschreibung der Trennfläche zwischen den Fluiden kommt die von Osher und Sethian [57] entwickelte Level-Set–Technik zum Einsatz. Die Navier-Stokes–Gleichungen, das Standardmodell für Fluidsimulationen, sind ein mathematisches Kontinuumsmodell und besitzen einen sehr geringen Modellfehler. Im Simulationspaket NaSt3DGPF werden die Gleichungen über die Chorin'sche Projektionsmethode gelöst. Ein impliziter Poissonlöser dient der genauen Approximation des Druckes im Simulationsgebiet. Mit der verwendeten Level-Set–Technik lassen sich die Krümmung und die Oberflächennormale der freien Oberfläche zwischen den beiden simulierten Fluiden sehr zuverlässig bestimmen. Dies nutzt der Löser NaSt3DGPF, um ebenfalls Oberflächenspannungseffekte mit zu berechnen. Unter Einbeziehung der Oberflächenspannung verringert sich nochmals der Modellfehler. Eine Parallelisierung mit dem *Message Passing Interface* (MPI) [56] auf Basis der von Schwarz vorgeschlagenen Gebietszerlegungsmethode [63] dient der Beschleunigung der Simulation. Der Strömungssimulator ist ein Universalwerkzeug im Bereich der Fluiddynamik und wurde schon in mehreren Anwendungen wie Schleusenbau [15], Tropfendynamik [18] oder porösen Medien [16], [48] sehr erfolgreich eingesetzt. Im Folgenden wird dieser Löser der einfachen Notation halber als *physikalisch korrekt* bezeichnet.

Autodesk Maya ist der Industriestandard im Bereich der Gestaltung und Animation von dreidimensionalen Umgebungen. Das Softwarepaket bietet eine umfangreiche Werkzeugpalette für alle bei einer Animation anfallenden Arbeiten. Maya kann über ein Programmierinterface beliebig erweitert werden. Damit liefert es eine gute Basis für die Implementierung eines virtuellen Labors. Das 3D-Animationsprogramm kam beispielsweise in Filmen wie Shrek, Findet Nemo, Herr der Ringe oder Star Wars zum Einsatz.

Eine vollständige Integration des Strömungslösers NaSt3DGPF in das Animationspaket Maya erleichtert nicht nur die Konfiguration und Visualisierung von Strömungssimulationen. Durch die Kopplung eröffnet sich für den Löser auch eine ganz neue Anwendung, die *Animation* von Fluiden. So wird auf einfachste Art und Weise eine physikalisch hochgenaue zweiphasige Fluidsimulation ein Teil einer in Maya erzeugten Szene. Der Lernaufwand für Anwender aus der Animationsbranche ist minimal, da die Erweiterung auf einer ihnen bekannten Software fußt.

Beim Einsatz der Level-Set–Technik für die Animation von Fluiden ergibt sich jedoch ein Problem. So kann die Level-Set–Methode zu unerwünschten Diskretisierungsartefakten führen, da sie an die Diskretisierung der Navier-Stokes–Gleichungen gebunden ist: Fallen bei einer numerischen Simulation bestimmte Fluidfeatures wie Tropfen oder Gischt unter die Diskretisierungsgenauigkeit, verschwinden diese, da sie nicht mehr aufgelöst werden können. Dies hat keinen gravierenden Einfluss auf die mathematische Qualität des Ergebnisses. Aus visueller Hinsicht sind aber solche Verluste fatal, da dem Betrachter der Animation verschwindende Tropfen direkt auffallen.

Eine Möglichkeit dem Problem zu begegnen ist eine Verfeinerung der Diskretisierung, so dass einzelne Tropfen wieder darstellbar werden. Dieser Ansatz ist möglich, solange nur kleinskalige Fluidsimulationen durchgeführt werden. Für diese liefern aktuelle Hochleistungsrechnersysteme noch ausreichende Kapazitäten.

Sehr viel schwieriger sind aber Simulationen von großskaligen Animationsszenen, wie die Überflutung einer Stadt. Hier stoßen selbst modernste Hochleistungsrechnersysteme an ihre Grenzen. Um ein realistisches Simulationsergebnis zu erhalten, müsste sowohl die großskalige Dynamik des Wassers als auch die sehr kleinskalige Dynamik von Spritzern und Gischt durch eine geeignete Diskretisierung aufgelöst werden. Das übersteigt alle aktuellen Möglichkeiten.

An dieser Stelle setzt eine in dieser Arbeit entwickelte Erweiterung des Lösers NaSt3DGPF an. Die *Zwei-Wege-Kopplung zwischen der Level-Set–Methode und einem Partikelsystem* dient der Auflösung von Fluidfeatures wie Tropfen oder Gischt, die normalerweise verloren gehen würden. Ein vollautomatischer Mechanismus erzeugt stets dann Partikel in einem Partikelsystem, wenn normalerweise Tropfen und Gischt entstehen würden, aber durch die zu grobe Diskretisierung wegfallen. Die Partikel werden geeignet visualisiert, so dass sie dem Betrachter als Tropfen erscheinen. Durch einen physikalisch motivierten Ansatz werden sie ähnlich wie echte Tropfen transportiert und können mit dem Restfluid interagieren. Die in dieser Arbeit entwickelte Rückkopplung zwischen dem Partikelsystem und der Level-Set–Funktion integriert die Partikel in das numerische Framework der Navier-Stokes–Gleichungen. Aus visueller Hinsicht wird mit der Rückkopplung der Tropfeneinschlag in die Wasseroberfläche nachgebildet.

Stand der Forschung

Das in der vorliegenden Arbeit vorgestellte graphische System zur Konfiguration, Simulation und Visualisierung von physikalisch korrekten Strömungssimulationen zählt zu einer sehr kleinen Gruppe derartiger Projekte aus dem universitären Umfeld. Ein Beispiel für ein ähnliches Projekt ist *DeViSor - Design and Visualization Software Ressource* [5] der Technischen Universität Dortmund. Im Rahmen des Projektes wurde eine graphische Benutzerschnittstelle zu dem frei verfügbaren Finite-Elemente–Löser FEATFLOW [24] für die einphasigen inkompressiblen Navier-Stokes–Gleichungen in drei Dimensionen umgesetzt.

Generell ist eine graphische Benutzeroberfläche für einen Strömungslöser vor allem ein Alleinstellungsmerkmal der großen kommerziellen Strömungspakete, wie STAR-CCM+ [65], Comsol [12], Fluent [30] oder ANSYS CFX [3]. Diese bieten dem Benutzer eine gemeinsame Oberfläche für den gesamten Arbeitsablauf einer auf ingenieurswissenschaftliche Anwendungen ausgerichteten Fluidsimulation.

Im Bereich der *Computer-Animation* gibt es ein sehr breit gefächertes Feld an Ansätzen zur Simulation und Animation von Fluiden, auf das hier nur auszugsweise eingegangen werden kann.

So betrachten beispielsweise verschiedene Arbeiten von Thürey und Rüde [43], [62], [67], [61] die Simulation und Animation von einphasigen Fluiden mit einer freien Oberfläche unter Anwendung der Lattice-Boltzmann-Methode. Diese Methode beruht auf einem diskreten mathematischen Modell für das Fluidverhalten. Der Strömungslöser wird mit dem freien 3D-Animationsprogramm *Blender* gekoppelt [67], [6]. Außerdem kommt in einer der Publikationen [61] ein Partikelsystem zur Animation von Wassertropfen und Spritzern zum Einsatz. Teile hiervon werden in der vorliegenden Arbeit weiterentwickelt. Insgesamt sind die von Thürey und Rüde eingesetzten Verfahren sehr schnell, werden aber im wissenschaftlichen Umfeld nur für Spezialanwendungen verwendet.

Die Arbeitsgruppe um Ron Fedkiw hat in verschiedenen Publikationen [25], [28], [27] Fluidanimationstechniken auf Basis der *Partikel-Level-Set*–Methode vorgestellt. Die Methode dient der Beschreibung der Trennfläche zwischen zwei Fluidphasen und nutzt eine um Partikel erweiterte Level-Set–Funktion für eine verbesserte Masseerhaltung. Durch den Einsatz eines Kontinuumsmodelles als Basis für die Simulation, wird zum Teil ein sehr geringer Modellfehler erreicht. Mit der in [28] vorgestellten Kopplung zwischen der Partikel-Level-Set–Methode und dem numerischen Framework der *Smoothed Particle Hydrodynamics* (SPH) [51], [33], sollen ähnlich wie in der vorliegenden Arbeit Fluidfeatures wie Tropfen und Gischt, die unterhalb der Diskretisierungsgenauigkeit liegen, nachgebildet werden. Je nach Anwendung relaxiert Fedkiw Anforderungen an den Diskretisierungsfehler um eine verbesserte Laufzeit zur erreichen.

Neben vielen weiteren aktuellen Arbeiten aus dem universitären Umfeld gibt es auch für die Animation von Fluiden einige kommerzielle Pakete. Das mit der *Unlimited*-Version von Autodesk Maya ausgelieferte System zur dreidimensionalen Fluidanimation *Maya Fluid Effects* beruht auf einer einphasigen Partikel-Simulation. Wegen der fehlenden zweiten Fluidphase lassen sich aber viele Effekte, wie zum Beispiel das Aufsteigen einer Luftblase im Wasser, nicht darstellen.

In Maya direkt lässt sich auch das Fluidanimations-Werkzeug *glu3D* [35] einbinden. Dieses sowie das eigenständige *Realflow* [59] beruhen auf Partikelverfahren. Solche Verfahren werden

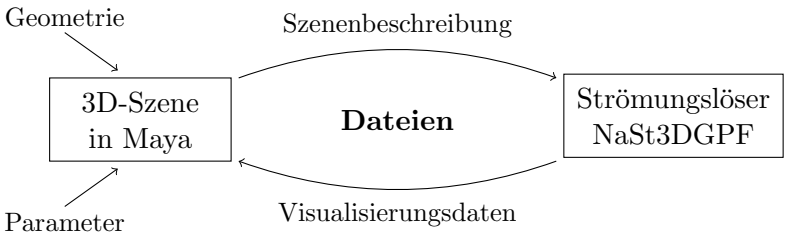
in der Animation häufig eingesetzt, da sie sehr schnell sind. Außerdem vermitteln diese Verfahren durch eine geeignete Nachbearbeitung trotz eines höheren Modellfehlers eine aus visueller Hinsicht akzeptable Approximation des Fluidverhaltens.

Fasst man den aktuellen Stand der Forschung zusammen, so kann man erkennen, dass sich zwei voneinander getrennte Ausrichtungen bei der Simulation von Fluiden gebildet haben. Auf der einen Seite gibt es Strömungslöser mit entsprechenden Benutzerschnittstellen, die sich wie der NaSt3DGPF-Löser auf äußerst geringe Modell- und Diskretisierungsfehler spezialisiert haben. Auf der anderen Seite wird im Bereich der Animation mehr auf den optischen Eindruck und die Geschwindigkeit von Fluidsimulationen Wert gelegt.

Mit der vorliegenden Arbeit wird nun erstmals eine erfolgreiche Verknüpfung der beiden Bereiche durchgeführt.

Eigener Beitrag

Für die Einbettung des Strömungslösers NaSt3DGPF in Autodesk Maya muss eine Vielzahl von Teilaufgaben bewältigt werden. Man betrachte dazu zunächst die folgende Abbildung, die die einzelnen Aspekte einer Kopplung der beiden Programme skizziert:



Die erste Aufgabe besteht in der Entwicklung einer Erweiterung der Oberfläche von Autodesk Maya um geeignete Funktionen zum Zusammenstellen einer Strömungssimulation. In dieser Arbeit wird die Erweiterung über Plugins in C++ sowie über eine Skriptsprache realisiert. Mit den neuen Komponenten kann ein Anwender in einer 3D-Szene interaktiv eine Simulationsumgebung, Hindernisobjekte, Bereiche mit Flüssigkeiten oder Gasen und Ein- sowie Ausströmregionen festlegen. Weitere Eigenschaften für die Simulation werden über entsprechende Eingabefelder vorgegeben. Diese Eigenschaften reichen von der Art der betrachteten Fluide über Angaben bezüglich der Auflösung bis hin zu Details der verwendeten numerischen Löser. Alle Teilaspekte der Konfiguration werden dem Benutzer über für Maya typische interaktive Komponenten zur Verfügung gestellt, so dass eine langwierige Einarbeitung entfällt.

Als zweite Teilaufgabe muss aus den in Maya festgelegten Details zur Simulation eine für den Strömungslöser NaSt3DGPF zu verarbeitende Szenenbeschreibung erzeugt werden. Die Beschreibung umfasst zum einen alle gesetzten Parameter und zum anderen eine diskretisierte Form der festgelegten Hindernis- und Fluid-Geometrien. Eine automatisierte Erzeugung der Szenenbeschreibung erfolgt in der vorliegenden Arbeit ebenfalls über ein Plugin für Maya. Die darin umgesetzten Verfahren verwenden für die Generierung der diskreten Geometriedarstellung *Voxelisierungs*-Algorithmen. Mit dem neu entwickelten Plugin werden außerdem Menü-Befehle zur Verfügung gestellt um den Löser auf Knopfdruck zu starten.

Es ist zu beachten, dass bereits die gerade skizzierten Plugins zur Konfiguration einer Strömungssimulation in Maya einen erheblichen Zeit- und Funktionsgewinn gegenüber dem vor

dieser Arbeit verwendeten Prinzip zum Zusammenstellen einer Simulation bedeuten. So muss in der ursprünglichen Form des Strömungslösers die gesamte Konfiguration der zu simulierenden Szene über eine ASCII-Datei geschehen. Nicht nur die unzähligen Parameter sondern gerade auch die Vorgabe von Hindernis- und Fluid-Geometrien machen dies sehr arbeitsaufwendig. Eine Angabe entsprechender Formen verläuft über geometrische Primitive wie Quader, Sphären oder Polyeder, die durch boolesche Operationen miteinander verknüpft werden. Dabei muss jede Größen- und Koordinatenangabe von Hand erfolgen.

Bei den in dieser Arbeit betrachteten Strömungssimulationen entstehen dreidimensionale, zeitabhängige und zudem sehr hochaufgelöste Simulationsdatensätze. Es ist weithin bekannt, dass derartige Datensätze extrem schwer zu handhaben sind, da sie mit Datenmengen im Giga- bis Terabyte-Bereich selbst modernste Hardwarekomponenten an ihre Leistungsgrenzen bringen.

In der bisherigen Form des Strömungslösers NaSt3DGPF wird für die zeitabhängige Visualisierung stets der komplette Simulationsdatensatz in ein Format für die Visualisierungsprogramme *Tecplot* [66] oder *Paraview* [58] umgeformt. Dieser Prozess ist nicht effizient und erfordert zudem viel Speicherkapazität. Soll die *freie Oberfläche*, die Trennschicht zwischen den beiden simulierten Fluidphasen, photorealistisch dargestellt und animiert werden, wird aus dem bereits umgeformten Datensatz die zeitabhängige Oberfläche als polygonales Netz extrahiert. Dies erfordert nochmals zusätzliche Zeit- und Speicherressourcen.

Eine in der vorliegenden Arbeit realisierte Erweiterung für den Strömungslöser NaSt3DGPF dient der effizienten und vollautomatischen Extraktion der freien Oberfläche als polygonales Netz beziehungsweise *Mesh*. Der Extraktionsprozess findet zeitgleich mit der Simulation statt, so dass für eine alleinige Visualisierung der Fluidoberfläche die speicherintensiven, zeitabhängigen Simulationsdaten nicht mehr abgelegt werden müssen. Der daraus entstehende Speichergewinn ist erheblich. Für die Oberflächenextraktion kommt der bekannte *Marching-Cubes*–Algorithmus von Lorensen und Cline [9] in einer korrigierten Version von Montani, Scateni und Scopigno [55] zum Einsatz. Dieser wird parallelisiert und somit beschleunigt.

Die bei der Extraktion der freien Oberfläche entstehenden zeitabhängigen polygonalen Netze sollen in Maya visualisiert oder animiert werden. Sie liegen für jeden Zeitschritt als einzelne Dateien in einem Standardformat für polygonale Netze vor und müssen in Maya eingelesen werden. Die große Anzahl macht ein manuelles Laden in Maya allerdings unmöglich. Deshalb wird in dieser Arbeit, soweit dem Autor bekannt, erstmalig ein Plugin entwickelt und implementiert, mit dem zeitabhängige polygonale Netze vollautomatisch in Maya integriert werden können. Um die Verarbeitung der Oberflächendaten besonders effizient zu machen, wird stets nur das polygonale Netz eingelesen, das für den gerade in Maya gewählten Animationszeitpunkt benötigt wird. Da die Erweiterung derart konzipiert ist, dass sich das zeitabhängige Mesh aus der Sicht des Benutzers wie ein statisches polygonales Netz verhält, können darauf sämtliche Standard-Werkzeuge für Polygone angewendet werden.

Durch die Extraktion der Fluid-Oberfläche während der Simulation sowie den automatisierten Mechanismus zum Einlesen der polygonalen Netze kann der Fortschritt der Simulation jederzeit überwacht werden. Hierdurch ist es möglich zu grob aufgelöste Berechnungen frühzeitig zu erkennen und zu korrigieren.

Insgesamt erhält ein Anwender von Autodesk Maya durch die in dieser Arbeit implementierten Zusatzkomponenten ein äußerst intuitives und effizientes vollständiges System zum Konfigurieren, Simulieren und Visualisieren von zweiphasigen Strömungen.

Im Bereich der *Animation* wird für die Zwei-Wege-Kopplung zwischen der Level-Set–Funktion und Partikeln im Programm NaSt3DGPF ein Partikelsystem für *massebehaftete* Partikel implementiert und parallelisiert. Es basiert auf einem Partikelsystem für *masselose* Partikel. Masselose Partikel dienen der Visualisierung von Geschwindigkeitsfeldern. Ihre Größe wird als infinitisimal klein betrachtet. So werden sie über die Zeit hinweg nur auf Basis der Geschwindigkeit der simulierten Fluide fortbewegt. Demgegenüber wird in dieser Arbeit ein sphärenförmiges Fluidtröpfchen mit einer realen Ausdehnung als massebehafteter Partikel bezeichnet. Durch die Ausdehnung und eine vorgegebene Dichte erhält der Tropfen ein Gewicht. Folglich wirkt auf ihn die Schwerkraft. Außerdem tritt bei dem Transport des Tropfens in der Luft auf Grund seiner Größe eine Luftwiderstands-Kraft auf. Das Zusammenspiel beider Kräfte und der daraus resultierende physikalisch motivierte Transport der Tropfen wird über eine Differentialgleichung dargestellt. Diese wird aus der Arbeit von Thürey und Rüde [61] übernommen und an das Geschwindigkeitsfeld des Strömungslösers angekoppelt. Aus der Diskretisierung der Gleichung ergibt sich ein Algorithmus zum Partikeltransport.

Für die Zwei-Wege-Kopplung wird eine Heuristik benötigt, die stets dann massebehaftete Partikel in das Partikelsystem emittiert, wenn dies in der Natur geschehen würde und die Diskretisierung der Level-Set–Funktion es aber nicht auflösen kann. In der vorliegenden Arbeit kommt als Entscheidungsmerkmal für die Partikelemission der Wert der Krümmung der freien Oberfläche sowie der Druck im Raum zum Einsatz. Die Startposition der Tropfen wird über eine dem Marching-Cubes–Algorithmus verwandte und sehr effiziente Technik bestimmt.

Bei der Rückkopplung soll die Physik eines Tropfeneinschlages in die Wasseroberfläche nachgebildet werden. Es ist bekannt, dass die physikalischen Vorgänge bei Tropfeneinschlägen extrem komplex sind [32]. Eine genauere Modellierung dieser Vorgänge wäre über die Navier-Stokes–Gleichungen möglich. Jedoch sollen gerade die massebehafteten Partikel Tropfen und Spritzer mit einer Größe unterhalb der Diskretisierungsgenauigkeit des Strömungssimulators darstellen. Somit ist eine Modellierung des Tropfeneinschlags mit den üblichen Mitteln des Lösers nicht sinnvoll.

In der vorliegenden Arbeit wird deshalb ein neuer und auf Grund seiner Einfachheit effizienter Ansatz zur Modellierung von Tropfeneinschlägen entwickelt. Es wird ein grobes Modell für die Kraft hergeleitet, die der einschlagende Tropfen auf die Wasseroberfläche ausübt. Die so berechnete Kraft wird als lokal an der Einschlagsstelle wirkende Volumenkraft umformuliert und so in die Navier-Stokes–Gleichungen eingekoppelt. Als optisches Ergebnis wird die Fluidoberfläche im Moment des Tropfeneinschlags deformiert. Anschließend entsteht die aus der Natur bekannte Wellenbildung um den Auftreffpunkt.

Für eine gemeinsame Visualisierung der massebehafteten Partikel und der freien Oberfläche müssen die vom Strömungslöser in Dateien geschriebenen, zeitabhängigen Partikeldaten in Maya zur Verfügung stehten. Auch hierfür wird in der vorliegenden Arbeit ein neues Plugin entwickelt. Es lädt die Daten vollautomatisch in ein Partikelsystem in Maya. Das Partikelsystem wird dann ausschließlich für die Visualisierung verwendet. Alle für den Partikelmechanismus benötigten Einstellungsmöglichkeiten werden über das eingangs erwähnte Konfigurationsplugin realisiert.

Zusammengefasst werden folgende Beiträge durch diese Arbeit geleistet:

- Konzeption und Implementierung eines Maya-Plugins zur benutzerfreundlichen interaktiven Konfiguration einer Strömungssimulation
- Entwicklung und Implementierung von Algorithmen zur automatisierten Erzeugung einer Szenenbeschreibung für den Strömungslöser NaSt3DGPF unter Einsatz verschiedener Voxelisierungsverfahren
- Integration des hocheffizienten parallelen Marching-Cubes–Algorithmus in den parallelen Strömungslöser
- Konzeption und Umsetzung eines Maya-Plugins zur vollautomatischen Integration von zeitabhängigen polygonalen Netzen in eine 3D-Szene
- Erweiterung eines parallelen Partikelsystems um ein physikalisch motiviertes Modell für den Transport von sphärenförmigen Tropfen beziehungsweise massebehafteten Partikeln
- Entwicklung und Implementierung einer Zwei-Wege-Kopplung zwischen einem Partikelsystem und der Level-Set–Methode im Navier-Stokes–Löser:
  - Konzeption und Implementierung einer vollautomatischen Methode für die Emission von nicht durch die Diskretisierung der Navier-Stokes–Gleichungen aufgelösten Tropfen
  - Herleitung eines einfachen und effizienten Modells für die auftretende Kraft eines Tropfeneinschlages
  - Umsetzung der Tropfeneinschlagskraft als lokale Volumenkraft in den diskretisierten Navier-Stokes–Gleichungen mit dem Ziel der Deformation der freien Oberfläche nach einem Tropfeneinschlag
  - Parallelisierung aller Teilkomponenten in einem mittels MPI realisierten *Distributed Memory* Modell
- Konzeption und Implementierung eines Maya-Plugins zur vollautomatischen Visualisierung von zeitabhängigen Partikeldaten

Übersicht

Die vorliegende Arbeit gliedert sich in folgende Kapitel:
Kapitel 2 stellt grundlegende Informationen zum Thema Strömungssimulation im Allgemeinen, zum Navier-Stokes–Löser NaSt3DGPF im Speziellen, sowie zum 3D-Animationsprogramm Autodesk Maya bereit.
Kapitel 3 beginnt mit der Konzeption einer Maya-Erweiterungen zur interaktiven Konfiguration von Fluidsimulationen. Alle wesentlichen Forderungen an entsprechende Plugins werden formuliert. Zudem erfolgt eine Vorstellung der implementierten Komponenten zum Speichern und Verwalten von Parametern und Geometrien einer Strömungssimulation.
Kapitel 4 geht auf die automatisierte Erzeugung einer Szenenbeschreibung für den Löser NaSt3DGPF aus den in Maya verwalteten Informationen ein. Notwendige Algorithmen und

Verfahren werden bereitgestellt. Insbesondere werden Voxelisierungsverfahren zur Festlegung von Anfangs- und Randbedingungen in einer Strömungssimulation eingeführt und die Implementierung vorgestellt.
Kapitel 5 bespricht die Erweiterung des NaSt3DGPF-Lösers um einen parallelen *Marching-Cubes*–Algorithmus. Nach einer Einführung in den Algorithmus folgen Details zur Implementierung und Parallelisierung im Löser.
Kapitel 6 dient der Konzeption des Maya-Plugins zum Einlesen von zeitabhängigen polygonalen Netzen. Die Implementierung wird skizziert.
Kapitel 7 thematisiert die Kopplung der freien Oberfläche an ein Partikelsystem. So wird zunächst die Umsetzung des Partikelsystems für massebehaftete Partikel im Strömungslöser beschrieben. Darauf folgt die Entwicklung und Implementierung der Verfahren für die Tropfen-Emission und den Tropfen-Einschlag. Abgeschlossen wird das Kapitel durch die Erörterung von Erweiterungen zur Integration der Partikel in Maya.
Kapitel 8 dient der Präsentation der Ergebnisse von simulierten Testproblemen und Performance-Analysen. Es werden gerenderte Resultate von verschiedenen Strömungssimulationen gezeigt. Neben den reinen Testproblemen für die einzelnen implementierten Komponenten werden auch optisch und technisch anspruchsvolle Fluidanimationssequenzen vorgestellt.
Kapitel 9 fasst die vorliegende Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Weiterentwicklungen.

2 Grundlagen

Im Rahmen dieses Kapitels sollen wichtige Grundlagen zur Fluid-Simulation und zum 3D-Animationsprogramm Autodesk Maya vorgestellt werden. Als mathematisches Modell für Strömungen werden die Navier-Stokes–Gleichungen eingeführt. Diese zu lösen ist die Hauptaufgabe in der Fluid-Simulation. Deshalb werden einige Grundideen zu numerischen Lösungsverfahren umrissen. Insbesondere wird der Navier-Stokes–Löser NaSt3DGPF [17],[15], der am Institut für Numerische Simulation der Universität Bonn entwickelt wird, beschrieben. Da dieser die Basis für die in dieser Arbeit erstellten Erweiterungen darstellt, müssen seine Eigenschaften, Funktionen und Techniken genauer betrachtet werden. Abgeschlossen wird dieses Kapitel durch einen Überblick über Autodesk Maya und seine Erweiterungsmöglichkeiten.

2.1 Die Navier-Stokes–Gleichungen

Um das Verhalten von Fluiden, also Gasen oder Flüssigkeiten simulieren zu können, muss ein mathematisches Modell eingeführt werden. Ein Modell ist die abstrakte, in diesem Fall mathematische Beschreibung für einen natürlichen Vorgang. Insbesondere müssen Größen wie die Geschwindigkeit, der Druck oder die Dichte eines Fluids in Zusammenhang gebracht werden. Dies geschieht bei mathematischen Modellen für physikalische Vorgänge meist mittels Differentialgleichungen.

Bei den Navier-Stokes–Gleichungen handelt es sich um das mathematische Modell für viskose[1] Fluide. Insbesondere werden im Folgenden die Navier-Stokes–Gleichungen für inkompressible, einphasige Fluide in drei Dimensionen vorgestellt. Inkompressibel bedeutet, dass eine Flüssigkeit beziehungsweise ein Gas sich nicht zusammenpressen lässt. Man spricht von mehreren Phasen, wenn die Interaktion mehrerer verschiedener Fluidarten, also Wasser und Öl oder Wasser und Luft beschrieben werden soll. Um die Einführung einfacher zu gestalten, sollen hier nur die Navier-Stokes–Gleichungen für genau eine Fluidart beschrieben werden. Die Navier-Stokes–Gleichungen sind ein System von partiellen Differentialgleichungen

$$\frac{\partial}{\partial t}\vec{u} + (\vec{u}\cdot\nabla)\vec{u} + \frac{1}{\varrho_\infty}\nabla p = \frac{\mu}{\varrho_\infty}\Delta\vec{u} + \vec{g}\,, \tag{2.1}$$

$$\nabla\cdot\vec{u} = 0\,. \tag{2.2}$$

Sie sind auf einem Gebiet $\Omega \subset \mathbb{R}^3$ und für die Zeit $t \in [0, t_{end}]$ definiert. In den Differentialgleichungen kommen das Geschwindigkeitsfeld $\vec{u}$ und der Druck p als orts- und zeitabhängige Größen vor. Die Dichte ϱ_∞ wird wegen der eben erwähnten Annahme der Inkompressibilität als konstant in Ort und Zeit angenommen. Als weitere Größe wird die dynamische Viskosität μ eingeführt. Mit ihr wird festgelegt, wie zähflüssig das Fluid ist. So hat Wasser bei Raumtemperatur eine Viskosität von $1.002 \times 10^{-3}\frac{kg}{ms}$. Luft hat dann eine Viskosität von $1.81 \times 10^{-5}\frac{kg}{ms}$.

[1] *Viskose* Fluide besitzen eine innere Reibung.

Wert	Bereich	Einheit	Beschreibung
$\vec{x}$	$\Omega \subset \mathbb{R}^3$	$[m]$	Ort
t	$[0, t_{end}]$	$[s]$	Zeit
$\vec{u}$	$\Omega \times [0, t_{end}] \to \mathbb{R}^3$	$[\frac{m}{s}]$	Geschwindigkeit
p	$\Omega \times [0, t_{end}] \to \mathbb{R}$	$[Pa]$	Druck
ϱ_∞	$\mathbb{R}$	$[\frac{kg}{m^3}]$	dynamische Viskosität
$\vec{g}$	$\mathbb{R}^3$	$[\frac{m}{s^2}]$	Volumenkraft

Tabelle 2.1: Übersicht über die in den Navier-Stokes–Gleichungen verwendeten Größen.

Schließlich steht die vektorielle Größe $\vec{g}$ für die Volumenkraft, die auf das Fluid wirkt. Die Volumenkraft ist meistens die Schwerkraft. In Tabelle 2.1 sind noch einmal alle gerade eingeführten Größen mit ihren Bereichen und Einheiten zusammengefasst.

Gleichung (2.1) wird in der Literatur als *Impulsgleichung* bezeichnet. Sie wird über das physikalische Grundgesetz der Impulserhaltung hergeleitet. Insbesondere werden in dieser Gleichung Kräfte, die im Fluid und außerhalb des Fluids wirken, mit der Geschwindigkeit des Fluids in Zusammenhang gebracht. Die zweite Gleichung (2.2) nennt sich *Kontinuitätsgleichung*. Diese wird aus dem Gesetz der Massenerhaltung hergeleitet. Grob gesagt beschreibt sie die Forderung nach Inkompressibilität. Im Rahmen dieser Einführung soll nicht genauer auf die Herleitung der Navier-Stokes–Gleichungen eingegangen werden. Hierfür sei auf [22],[13] und [7] verwiesen, die auch als Quelle für diese Einführung fungieren.

Auch wenn die Notation in den Gleichungen (2.1) und (2.2) sehr häufig wegen ihrer Kürze verwendet wird, sollen an dieser Stelle auch noch einmal die Gleichungen (nach Einsetzen der Gleichung (2.2) in Gleichung (2.1)) in ihrer komponentenweisen Darstellung[2]

$$\begin{aligned}
\frac{\partial u}{\partial t} + \frac{1}{\varrho_\infty}\frac{\partial p}{\partial x} &= \frac{\mu}{\varrho_\infty}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) - \frac{\partial(u^2)}{\partial x} - \frac{\partial(uv)}{\partial y} - \frac{\partial(uw)}{\partial z} + g_x\,, \\
\frac{\partial v}{\partial t} + \frac{1}{\varrho_\infty}\frac{\partial p}{\partial y} &= \frac{\mu}{\varrho_\infty}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} - \frac{\partial(vw)}{\partial z} + g_y\,, \qquad (2.3) \\
\frac{\partial w}{\partial t} + \frac{1}{\varrho_\infty}\frac{\partial p}{\partial z} &= \frac{\mu}{\varrho_\infty}\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) - \frac{\partial(uw)}{\partial x} - \frac{\partial(vw)}{\partial y} - \frac{\partial(w^2)}{\partial z} + g_z \text{ und}
\end{aligned}$$

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \qquad (2.4)$$

angegeben werden. Diese Darstellung zeigt noch einmal um so deutlicher, dass hier ein System von partiellen Differentialgleichungen zweiter Ordnung vorliegt. Außerdem hilft sie dem im Nabla-Kalkül ungeübten Leser diese Gleichung besser zu lesen.

2.1.1 Anfangs- und Randbedingungen

Die im letzten Abschnitt beschriebenen Differentialgleichungen beschreiben das Verhalten eines Fluids im Raum. Sie sagen aber noch nichts über das konkrete Problem aus, für das man eine

[2] Beachte: $\vec{x} = (x\ y\ z)^t$, $\vec{u} = (u\ v\ w)^t$, $\vec{g} = (g_x\ g_y\ g_z)^t$.

Simulation durchführen möchte. Hier kommen die Anfangs- und Randbedingungen ins Spiel.

Anfangsbedingungen

Unter Anfangsbedingungen versteht man die initialen Festlegungen der Größen in einer Differentialgleichung. So können für die Navier-Stokes–Gleichungen der Druck p und die Geschwindigkeit $\vec{u}$ für den Zeitpunkt $t = 0$ vorgegeben werden.

Randbedingungen

Randbedingungen sind Vorgaben für das Verhalten von Differentialgleichungen auf den Rändern $\Gamma = \delta\Omega$ des Grundgebiets Ω. Im Gegensatz zu Anfangsbedingungen gelten Randbedingungen zu jedem Zeitpunkt t. Im Fall der Navier-Stokes–Gleichungen werden Randbedingungen für das Geschwindigkeitsfeld $\vec{u}$ vorgegeben[3]. Bei diesen unterscheidet man zwischen Dirichlet- und Neumann-Randbedingungen. Dirichlet-Randbedingungen setzen feste Werte für Geschwindigkeiten, während Neumann-Randbedingungen Werte für Geschwindigkeitsgradienten vorgeben.

Im Folgenden sollen einige mögliche Randbedingungen vorgestellt werden. Dazu wird in Anlehnung an [13] mit $\vec{n}$ der Normalenvektor auf dem Rand Γ bezeichnet, während $\vec{t}$ für alle zum Rand tangential verlaufenden Richtungen steht. $\delta_{\vec{n}}$ steht für die Ableitung in Normalenrichtung. Selbstverständlich lassen sich für verschiedene Bereiche des Randes auch verschiedene Randbedingungen angeben.

Als *Haftbedingung* wird die Forderung nach einer am Rand verschwindenden Geschwindigkeit

$$\vec{u}|_\Gamma = 0 \tag{2.5}$$

bezeichnet. Dies ist eine Dirichlet-Randbedingung. Wie der Name schon sagt, soll durch diese Forderung das Verhalten eines am Rande des Gebiets haftenden Fluids bewirkt werden.

Eine weitere Dirichlet-Randbedingung ist die *fixe Ein-* oder *Ausströmbedingung*. Durch diese soll ein Ein-/Ausströmen des Fluids am Rand mit einer festen Geschwindigkeit beschrieben werden. Wenn man diese feste Geschwindigkeit als $\vec{u}_0$ bezeichnet, erhält man als Randbedingung

$$\vec{u}|_\Gamma = \vec{u}_0\,. \tag{2.6}$$

Das Gegenstück zur Haftbedingung sind die *Rutschbedingungen*. Diese sind kombinierte Neumann- und Dirichlet-Randbedingungen der Form[4]

$$(\vec{u}\cdot\vec{n})|_\Gamma = 0\,, \qquad (\delta_{\vec{n}}(\vec{u}\cdot\vec{t}))|_\Gamma = 0\,. \tag{2.7}$$

Mit der verschwindenden Geschwindigkeit in Normalenrichtung wird ausgedrückt, dass kein Fluid das Gebiet verlässt. Die Neumann-Bedingung beschreibt die reibungsfreie Bewegung des Fluids am Rand.

[3] Physikalische Randbedingungen für den Druck werden bei den Navier-Stokes–Gleichungen grundsätzlich nicht benötigt. Dennoch müssen aus technischen Gründen bei der numerischen Lösung der Gleichungen Randbedingungen vorgegeben werden. Genaueres hierzu findet sich in [22].

[4] Der Vektor $\vec{t}$ versteht sich hier als beliebige tangentiale Richtung.

Eine weitere Neumann-Bedingung beschreibt die *natürliche Ausströmbedingung*. Bei dieser verschwindet die Ableitung der Geschwindigkeit in Normalenrichtung am Rand,

$$(\delta_{\vec{n}}\vec{u})|_\Gamma = 0\,. \tag{2.8}$$

Anschaulich gesprochen besagt diese Forderung, dass sich die Geschwindigkeit des Fluids beim Übertritt über den Rand nicht ändert. Hierdurch kann ein Abfließen des Fluids realisiert werden.

Neben den bisher erwähnten Bedingungen gibt es noch *periodische Randbedingungen* für einzelne Koordinatenrichtungen. Nimmt man für eine Richtung diese Bedingungen an, so werden die Werte an den in einer Richtung gegenüberliegenden Rändern gleichgesetzt. Diese Wahl ermöglicht ein periodisches Fortsetzen des Gebiets in die entsprechende Richtung.

Insgesamt stellen die Anfangs- und Randbedingungen einen essentiellen Bestandteil der Fluidsimulation dar. Durch diese kann das Verhalten des Fluids gesteuert werden: Es können Bereiche angegeben werden, in denen Fluid einströmt oder ausströmt. Es ist möglich, die Geschwindigkeit des Gases oder der Flüssigkeit festzulegen oder Eigenschaften wie das Rutschverhalten des Fluids am Rand vorzugeben. Aus diesem Grund wird das Thema der Vorgabe von geeigneten Randbedingungen in den Kapiteln zur Konfiguration von Fluidsimulationen (Kapitel 3 und 4) noch einmal eindringlich behandelt werden.

2.1.2 Numerische Lösung

Die Lösung der Navier-Stokes–Gleichungen und damit die Simulation von Strömungen ist eine komplexe Aufgabe. Zunächst läge es nahe, nach einer analytischen Lösung für die Gleichungen zu suchen. Leider ist jedoch bis zum heutigen Tage keine geschlossene, starke Lösung der dreidimensionalen Navier-Stokes–Gleichungen für beliebige Anfangsbedingungen und beliebige Zeitwerte bekannt. Die Frage nach einer solchen Lösung ist von so großem Interesse, dass sie im Rahmen der von dem *Clay Mathematics Institute* in Cambridge (Massachusetts) aufgestellten Liste der *Millenium Problems* [29] genannt wird. Für jedes der sieben Millenium Probleme ist ein Preisgeld von einer Million US-Dollar ausgeschrieben. Sie gelten mit als die wichtigsten und schwierigsten mathematischen Fragestellungen unserer Zeit.

Da sich die Navier-Stokes-Gleichungen derzeitig nicht analytisch lösen lassen, müssen sie numerisch gelöst werden. Es ist also nötig, sie in eine Form zu überführen, mit der moderne Hochleistungsrechner sie näherungsweise berechnen können.

Diskretisierung

Um eine Differentialgleichung numerisch lösen zu können, muss sie zunächst *diskretisiert* werden. Hierfür gibt es verschiedene Methoden. Im Folgenden soll der Begriff der *Diskretisierung* an Hand der *Finite-Differenzen–Methode* erklärt werden. Unter einer Diskretisierung versteht man bei dieser den Übergang von der kontinuierlichen Betrachtung der Differentialgleichung zu einer punktweisen Betrachtung. Es sei zum Beispiel das Randwertproblem

$$u : [0,1] \to \mathbb{R} \qquad f : [0,1] \to \mathbb{R} \tag{2.9}$$

$$\Delta u = f \tag{2.10}$$

$$u(0) = u_0 \qquad u(1) = u_1 \tag{2.11}$$

gegeben.

Die Diskretisierung dieser Differentialgleichung erfolgt nun durch die Überführung des Differentialoperators in seine diskrete Form. Eine solche Überführung soll nun für die erste Ableitung der Funktion u beschrieben werden. Man betrachte dazu deren Definition

$$\frac{\partial u}{\partial x} = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h} \,. \tag{2.12}$$

Eine diskrete Form der Ableitung ergibt sich durch das Weglassen der Grenzwertbildung,

$$\frac{\partial u}{\partial x} \approx \frac{u(x+h) - u(x)}{h} \,. \tag{2.13}$$

Motiviert durch diese Näherung wird nun der Raum, auf dem u definiert ist, nur noch in einzelnen Punkten x_i des Abstands h betrachtet und somit diskretisiert. Für das kontinuierliche Intervall $[0, 1]$ führt dies zu der Wahl

$$x_i := ih \,, \qquad i \in 0, \ldots, N \qquad \text{mit } h = \frac{1}{N}. \tag{2.14}$$

Hiermit wird das Intervall $[0, 1]$ durch $N + 1$ Punkte beschrieben. Der Wert h entspricht, wie erwähnt, dem Abstand zwischen den diskreten Punkten und wird häufig auch als *Maschenweite* bezeichnet. Ist die Lösung u einer diskretisierten Differentialgleichung gesucht, bestimmt man sie an den Punkten x_i.

Die durch (2.13) eingeführte Approximation an die Ableitung wird in der Literatur als *Vorwärtsdifferenz* bezeichnet. Auf analoge Weise kann man auch die *Rückwärtsdifferenz*

$$\frac{\partial u}{\partial x} \approx \frac{u(x) - u(x-h)}{h} \tag{2.15}$$

einführen. Bei beiden Verfahren entsteht für die Ableitung an den diskreten Stellen x_i ein Fehler, der in der Größenordnung von h liegt, also $O(h)$ ist.

Möchte man eine Diskretisierung der zweiten Ableitung durchführen, so kann man zuerst die erste Ableitung mittels einer Vorwärtsdifferenz diskretisieren und anschließend mit einer Rückwärtsdifferenz die Ableitung der diskreten ersten Ableitung bilden. Durch dieses Vorgehen erhält man für die zweite Ableitung $\frac{\partial^2 u}{\partial x^2}$ die diskrete Form

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \,. \tag{2.16}$$

Führt man eine solche Diskretisierung der zweiten Ableitung durch, entsteht dabei an den Stellen x_i ein Fehler von $O(h^2)$.

Insgesamt kann man erkennen, dass sich mit der gerade beschriebenen *Finite-Differenzen–Methode* auf einfache Weise Diskretisierungen von beliebigen Ableitungen durchführen lassen. Wendet man die Finite-Differenzen–Methode auf Gleichung (2.10) an, so erhält man die diskrete

Darstellung der Differentialgleichung in jedem Punkt x_i,

$$\frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-i})}{h^2} = f(x_i) \quad \forall i = 1 \dots N-1\,. \tag{2.17}$$

Mit den Randbedingungen in (2.11) führt dies zum linearen Gleichungssystem

$$\begin{pmatrix} 1 & & & & & \mathbf{0} \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ \mathbf{0} & & & & & 1 \end{pmatrix} \cdot \begin{pmatrix} u(x_0) \\ u(x_1) \\ \\ \vdots \\ \\ u(x_N) \end{pmatrix} = \begin{pmatrix} u_0 \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{N-1}) \\ u_1 \end{pmatrix}, \tag{2.18}$$

das es zu lösen gilt. In der Numerik sind viele effiziente Verfahren bekannt, um eine solche Lösung zu finden.

Als Ergebnis der Diskretisierung und anschließenden Lösung des daraus resultierenden linearen Gleichungssystems erhält man die Werte der Lösung der Differentialgleichung ausgewertet an den *Gitterpunkten* x_i.

Die Differentialgleichung, an der hier die Diskretisierung vorgestellt wurde, war nur auf einem Intervall in $\mathbb{R}$ definiert. Von wissenschaftlichem Interesse sind aber meistens Differentialgleichungen die auf Gebieten des $\mathbb{R}^2$ oder $\mathbb{R}^3$ definiert sind. Für solche Gleichungen funktioniert die Finite-Differenzen–Methode auf analoge Weise, so dass zum Beispiel Δv mit $v : \mathbb{R}^2 \to \mathbb{R}$ mittels

$$\Delta v(x,y) = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \approx \frac{v(x+h,y) + v(x,y+h) - 4v(x,y) + v(x-h,y) + v(x,y-h)}{h^2} \tag{2.19}$$

diskretisiert wird.

Für diese Diskretisierung wird vorausgesetzt, dass der Raum $\mathbb{R}^2$ mit einem *uniformen* Gitter der Maschenweite h versehen wurde. Uniform bedeutet hier, dass die Maschenweite h in jede Raumrichtung die selbe ist. Eine derartige Diskretisierung würde für das Einheits-Quadrat $[0,1] \times [0,1]$ Gitterpunkte (x_i, y_j) der Form

$$(x_i, y_j) \in \mathbb{R}^2\,, \qquad x_i = ih\,, \qquad y_j = jh\,, \qquad h = \frac{1}{N} \tag{2.20}$$

liefern. Für Differentialgleichungen, die auf dem $\mathbb{R}^3$ definiert sind, setzt sich die Vorgehensweise auf analoge Weise fort.

Neben den hier vorgestellten Diskretisierungen von Ableitungen gibt es im Bereich der Finiten Differenzen auch noch andere Formeln für diskrete Ableitungen. Mit diesen lassen sich bessere Fehlerordnungen erreichen. Für eine tiefer gehende Einführung sei deshalb auf die Fachliteratur zur Numerik von Differentialgleichungen (zum Beispiel [39]) verwiesen. Selbstverständlich ist die Finite-Differenzen–Methode auch nicht der einzige Weg, um Differentialgleichungen zu diskretisieren. Da es aber ein häufig verwendeter Weg ist, der zudem anschaulich ist, ist dies das

einzige Verfahren, das in dieser Einführung vorgestellt werden soll. Andere Verfahren sind zum Beispiel das *Finite-Elemente–Verfahren* oder das *Finite-Volumen–Verfahren*. Eine Einführung in diese Methoden bietet [2].

Zeitdiskretisierung

Bei dem im vorherigen Abschnitt beschriebenen Vorgehen zur Diskretisierung von Differentialgleichungen wurde implizit angenommen, dass die Differentialgleichung ein von der Zeit unabhängiges Verhalten beschreibt. Die Diskretisierung erfolgte somit nur im *Ort*, kann also auch als *Ortsdiskretisierung* bezeichnet werden. Wenn eine Differentialgleichung ebenso von der Zeit abhängig ist, wird ein etwas anderes Vorgehen sinnvoll. Dieses Vorgehen soll anhand der Wärmeleitungsgleichung

$$\frac{\partial u}{\partial t} = a\frac{\partial^2 u}{\partial x^2} \tag{2.21}$$

vorgestellt werden.

Die Funktion u beschreibt die Temperatur an einer Stelle x zum Zeitpunkt t. Der Koeffizient a beschreibt die Temperaturleitfähigkeit eines Materials. Um das Beispiel einfach zu gestalten, soll u nur auf einem eindimensionalen Intervall mit entsprechenden Randwerten definiert sein,

$$u : [0,1] \times [0,\infty) \to \mathbb{R} \qquad a \in \mathbb{R} \tag{2.22}$$

$$u(0,t) = u_0 \qquad u(1,t) = u_1 \qquad \forall t \in [0,\infty)\,. \tag{2.23}$$

Mit diesen Voraussetzungen beschreibt Gleichung (2.21) die Temperaturverteilung in einem dünnen Stab über die Zeit hinweg. Die Temperatur an den jeweiligen Stab-Enden sei dabei durch u_0 und u_1 gegeben. Die Grundtemperatur, also eine Anfangsbedingung für die Temperatur zum Zeitpunkt $t = 0$, sei noch entsprechend zu wählen.

Zunächst muss für Gleichung (2.21) eine Ortsdiskretisierung durchgeführt werden. Hier soll die Diskretisierung aus (2.14) verwendet werden. Damit ergibt sich die ortsdiskrete Fassung der Differentialgleichung mit

$$\frac{\partial}{\partial t}u(x_i,t) = a\frac{u(x_{i+1},t) - 2u(x_i,t) + u(x_{i-1},t)}{h^2} \quad \forall i = 1 \ldots N-1\,. \tag{2.24}$$

Der nächste Schritt ist nun die Diskretisierung der Zeit. Sei dazu die *Zeitschrittweite*[5] Δt gegeben. Ab sofort wird dann die Zeit nur noch an den diskreten Zeitpunkten t_n mit

$$t_n = n\Delta t \tag{2.25}$$

betrachtet. Zur übersichtlicheren Darstellung führt man nun die in der Literatur übliche Schreibweise

$$u(x_i,t_n) := u^{(n)}(x_i) \tag{2.26}$$

ein. Analog zur Ortsdiskretisierung kann man eine Diskretisierung für die Zeitableitung be-

[5] Die Zeitschrittweite ist das Analogon zur Maschenweite in der Zeit.

stimmen,

$$\frac{\partial}{\partial t}u^{(n+1)}(x_i) = \frac{u^{(n+1)}(x_i) - u^{(n)}(x_i)}{\Delta t}\,. \tag{2.27}$$

Durch Einsetzen in Gleichung (2.24) erhält man schließlich die Vorschrift

$$u^{(n+1)}(x_i) = a\frac{u^{(n)}(x_{i+1}) - 2u^{(n)}(x_i) + u^{(n)}(x_{i-1})}{h^2}\Delta t + u^{(n)}(x_i) \tag{2.28}$$

zur Berechnung der Temperatur zu einem Zeitpunkt t_{n+1} aus den Größen zum Zeitpunkt t_n.

Diese Vorgehensweise nennt man *explizit*, da nur Größen des alten Zeitpunktes zur Berechnung des neuen Wertes verwendet werden. Wegen der Form der Diskretisierung der Zeitableitung in (2.27) hat das Verfahren den Namen *Euler-Verfahren*. Insgesamt wurde hier also die Wärmeleitungsgleichung mit dem *expliziten Euler-Verfahren* diskretisiert.

Mit obiger Vorschrift (2.28) erhält man ein Iterationsverfahren, mit dem man Schritt für Schritt die Differentialgleichung zu jedem Zeitpunkt lösen kann. Hierfür muss noch nicht einmal ein lineares Gleichungssystem gelöst werden. Jedoch sind explizite Verfahren sehr schlecht für größere Zeitschrittweiten Δt geeignet, da der Fehler bei zu großen Schritten nicht mehr akzeptabel wird. Eine Möglichkeit besteht nun darin, bessere Zeitdiskretisierungen statt der in Gleichung (2.27) zu verwenden oder Δt sehr klein zu machen. Insbesondere Letzteres erfordert dann aber eine sehr hohe Rechenzeit.

Unabhängig von dem großen Fehler bei zu großen Zeitschrittweiten haben explizite Verfahren noch einen weiteren Nachteil: Sie sind nur eingeschränkt *stabil*. Dies bedeutet, dass sich Rundungsfehler während des Berechnungsprozesses eventuell derart aufschaukeln können, dass das Ergebnis unbrauchbar wird. Im Kontext der Strömungsmechanik gibt es deshalb die sogenannten *Courant-Friedrichs-Levi-Bedingungen*, mit denen die Stabilität der Verfahren sichergestellt werden kann. Allerdings werden durch diese Bedingungen Einschränkungen an die Zeitschrittweite gemacht, die zum Teil drastisch sein können.

Alternativ kann man für die diskrete Ortsableitung Werte zum Zeitpunkt $t = n + 1$ statt $t = n$ nehmen. Diese sind aber zunächst unbekannt, so dass man dann für jeden Zeitschritt wieder ein lineares Gleichungssystem lösen muss. Das zuletzt beschriebene Vorgehen wird in der Literatur als *implizit* (statt explizit) bezeichnet. Es ermöglicht die Wahl von recht großen Zeitschritten, ohne dass der Fehler zu groß wird. Gleichzeitig treten bei dem Vorgehen keine Probleme bei der Stabilität auf, da implizite Verfahren uneingeschränkt stabil sind.

Die bisher vorgestellten Werkzeuge zur Diskretisierung bilden eine gute Basis zum groben Verständnis von Lösungsverfahren für Differentialgleichungen. Als Beispiele wurden ausschließlich einfache Gleichungen genommen, damit die Idee klar herausgestellt wird. Für die Diskretisierung eines komplexen partiellen Differentialgleichungssystems, wie den Navier-Stokes–Gleichungen, benötigt man meist noch ausgeklügeltere Techniken. So kann zum Beispiel ein solches Gleichungssystem nicht einfach nach obigem Schema direkt diskretisiert werden, sondern muss erst in bestimmte Teile zerlegt werden, für die es dann wiederum speziell optimierte Verfahren gibt. Insgesamt wäre also die Vorstellung einer Diskretisierung für die Navier-Stokes–Gleichungen sehr umfangreich und kaum geeignet für eine grobe Einführung in die Thematik. Deshalb wird an dieser Stelle darauf verzichtet. Interessierte Leser sollten sich hierfür die anfangs erwähnten Quellen [22],[13] und [7] anschauen, da diese eine Diskretisierung der Navier-Stokes–Gleichungen beschreiben.

2.1.3 Parallelisierung

Die numerische Lösung der Navier-Stokes–Gleichungen bleibt trotz des Einsatzes hochmoderner Workstations sehr zeitaufwendig[6]. Gerade bei sehr hoch aufgelösten[7] Simulationen, die für einen hohen Detailgrad nötig sind, muss die Rechenlast auf mehrere Computer aufgeteilt werden. Man spricht hierbei von *Parallelisierung*, da man versucht, zeitgleich auf verschiedenen Rechnern an ein und demselben Problem zu arbeiten. So ist es durchaus üblich mehrere hundert Maschinen in einem Netzwerk zu einem *Cluster* zusammenzuschließen. Diese bewältigen dann ein Problem gemeinsam.

Für die praktische Umsetzung von Parallelisierungen wird in der Numerik meistens das *Message Passing Interface* (MPI) [56] verwendet.[8] Diese Programmierschnittstelle bietet ein Framework für alle bei einer Parallelisierung benötigten Funktionalitäten.

Eine wesentliche Fragestellung zur parallelen Simulation mit den Navier-Stokes–Gleichungen ist die Frage nach einer sinnvollen Strategie zur Zerlegung der Aufgabenstellung in einzelne Teilaufgaben. Hier ist die Verwendung eines Gitters zur Diskretisierung der Gleichungen eine ideale Voraussetzung. Die *Gebietszerlegungsmethode* von Schwarz [63] schlägt vor, das diskretisierte Gebiet Ω_h in P Teilgebiete $\Omega_h^1, \ldots, \Omega_h^P$ zu zerlegen und auf diesen das Problem lokal zu lösen. Sei dazu beispielsweise ein Gebiet Ω_h der Form

$$\Omega_h = \bigcup_{i=1..I} \bigcup_{j=1..J} \bigcup_{k=1..K} \{(x_i, y_j, z_k)\}\,, \tag{2.29}$$

$$x_i = ih\,, \qquad y_j = jh\,, \qquad z_k = kh \tag{2.30}$$

gegeben. Dabei seien I, J und K jeweils die Anzahl der Zellen in die jeweilige Raumrichtung unter der Annahme, dass (x_i, y_j, z_k) die Position eines Zellmittelpunktes angibt. Schwarz schlägt also vor, dieses diskrete Gebiet in disjunkte Quader zu zerlegen. Dann wird auf jedem Quader von einem Rechner die Navier-Stokes–Gleichung gelöst. Gleichzeitig kann auch das Geschwindigkeitsfeld, der Druck, usw. auf verschiedene Rechner verteilt werden. Somit wird nicht nur die Rechenlast verteilt, sondern auch der nötige Speicher in analoger Weise aufgeteilt.

Bei diesem Ansatz stellt sich jedoch ein Problem: Wie man im Abschnitt 2.1.2 zur Diskretisierung sehen konnte, benötigen die Finiten Differenzen Werte aus angrenzenden Zellen, um die Ableitung zu berechnen. Am Rand eines jeden Teilgebietes Ω_h^p fehlt also mindestens eine Zellschicht. Um diesem Problem aus dem Weg zu gehen, werden zusätzliche Zellschichten eingeführt, die die Werte der Randzellen aus den benachbarten Teilgebieten enthalten. Nachdem eine entsprechende Rechnung parallel auf allen Teilgebieten durchgeführt wurde, muss also eine Kommunikation zwischen den Rechnern mit den benachbarten Teilgebieten stattfinden. Bei dieser werden die Werte an den Randzellen untereinander ausgetauscht und in die jeweilige zusätzliche Randzellenschicht übertragen. Die Darstellung eines solchen Datenaustauschs findet sich in Abbildung 2.1 auf Seite 20.

Unter Anwendung eines solchen Parallelisierungsverfahrens kann eine Navier-Stokes–Rechnung erheblich beschleunigt werden. Es ist jedoch nicht möglich, eine Rechnung durch die

[6] Verfahren, die die Navier-Stokes–Gleichungen lösen und gleichzeitig sehr schnell sind, verwenden meistens vereinfachende Annahmen, bei denen die Lösung nicht mehr im physikalischen Sinne korrekt ist.

[7] Hoch aufgelöst bedeutet hier, dass bei einem großen Simulationsgebiet die Maschenweite sehr klein ist.

[8] Das MPI kommt auch beim NaSt3DGPF-Löser zum Einsatz.

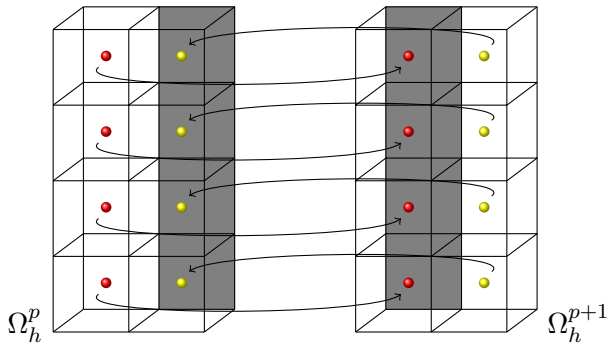


Abb. 2.1: Datenaustausch zwischen Randzellen und zusätzlicher Randzellenschicht (grau hinterlegt).

Aufteilung auf P Teilgebiete und anschließende parallele Rechnung, um den Faktor P zu beschleunigen. Der zusätzliche Kommunikationsaufwand zwischen den Rechnern verhindert dies. Dennoch ist die Parallelisierung *das* wesentliche Verfahren zum Simulieren von Fluiden.

2.2 NaSt3DGPF - Ein Löser für die Navier-Stokes–Gleichungen

In diesem Abschnitt sollen die Details zu dem in dieser Arbeit verwendeten Navier-Stokes–Löser NaSt3DGPF beschrieben werden. Die bisher vorgestellten Techniken und Grundlagen sollten auf jeden numerischen Löser gleichermaßen zutreffen. Tatsächlich machen aber erst zusätzliche Funktionen und die Erweiterungen auf mehrphasige Fluide einen Löser interessant. Hier stellt sich heraus, wie gut die Ergebnisse werden und wofür sich der Löser überhaupt eignet.

2.2.1 Zweiphasigkeit mit Hilfe von Level-Set–Techniken

Eine fundamentale Eigenschaft des in dieser Arbeit verwendeten Navier-Stokes–Lösers ist die Zweiphasigkeit. Es lassen sich also in einem Simulationsgebiet gleichzeitig zwei verschiedene Fluide, wie zum Beispiel Luft und Wasser, mit entsprechend verschiedenen Viskositäten und Dichten simulieren. Um dies zu erreichen muss für jede *Gitterzelle*, also für jeden diskreten Punkt im Raum, eine Zuordnung zu einer der Phasen gefunden werden. Der Strömungslöser verwendet hierzu die Level-Set–Technik. Diese soll nun vorgestellt werden.

Die Level-Set–Technik wurde von Osher und Sethian [57] entwickelt. Sie verwendet eine vorzeichenbehaftete Distanzfunktion ϕ, um implizit eine *freie Oberfläche* im Raum zu definieren. Entsprechend gibt $|\phi|$ immer den Abstand zur freien Oberfläche an. Im Falle der zweiphasigen Navier-Stokes–Gleichungen ist die freie Oberfläche die Grenzschicht zwischen den beiden Fluid-Phasen, also zum Beispiel die *Wasseroberfläche*.

Sei nun die Level-Set–Funktion ϕ durch

$$\phi : \Omega \times [0, t_{end}] \to \mathbb{R} \tag{2.31}$$

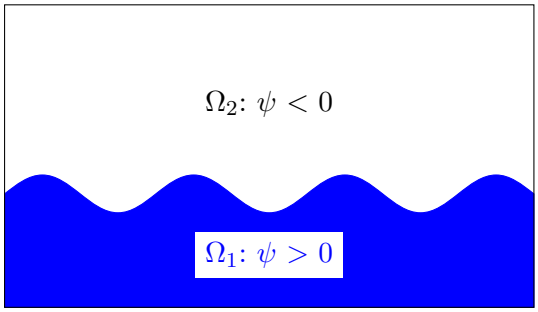


Abb. 2.2: Vorzeichenwahl der Level-Set–Funktion ϕ.

gegeben. Die freie Oberfläche Γ_f zum Zeitpunkt t ergibt sich dann aus der Nullniveaumenge von ϕ zu

$$\Gamma_f(t) = \{\vec{x} \in \Omega |\, \phi(\vec{x}, t) = 0\}\,. \tag{2.32}$$

Über das Vorzeichen der Abstandsfunktion bezüglich der Oberfläche lässt sich die gewünschte Zuordnung der Gitterzellen zu den beiden Fluid-Typen finden. Bezeichnet man die Gebiete, auf der die beiden Phasen vorliegen, mit Ω_1 und Ω_2 so führt [17] für den vorliegenden Löser die Festlegung

$$\phi(\vec{x}, t) \left\{ \begin{array}{ll} < 0 & \text{falls } \vec{x} \in \Omega_1 \\ = 0 & \text{falls } \vec{x} \in \Gamma_f \\ > 0 & \text{falls } \vec{x} \in \Omega_2 \end{array} \right\} \tag{2.33}$$

ein. Im typischen Simulationsfall von einem Gas und einer Flüssigkeit wird dann die Region Ω_1 für das Gas und Ω_2 für die Flüssigkeit verwendet. Abbildung 2.2 illustriert diese Wahl.

Eine implizite Darstellung des Phasenübergangs bringt sehr gute Eigenschaften mit. Treffen im typischen Anwendungsfall zwei Flüssigkeitsfronten aufeinander und vereinigen sich, so bedeutet dies eine Topologieänderung der freien Oberfläche. Während andere Verfahren zur Trennung von verschiedenen Phasen hier Probleme bereiten, verschmelzen bei der Level-Set–Technik die Gebiete ohne Schwierigkeiten. In Abbildung 2.3 auf Seite 22 wird der Vorgang des Abtrennens zweier Nullniveaumengen einer Level-Set–Funktion dargestellt. Durch eine noch zu treffende Annahme an die Level-Set–Funktion sind die Übergänge von zwei sich treffenden freien Oberflächen Lipschitz-stetig. Dies führt zu numerisch besseren Resultaten.

Zusätzlich zu den genannten Vorteilen lässt sich die Verwaltung der freien Oberfläche auch sehr einfach parallelisieren (siehe Abschnitt 2.1.3). Die Level-Set–Funktion kann wie jedes andere Feld auf einem Gitter im Raum diskretisiert werden. Auf eine solche Diskretisierung lässt sich dann trivial die Gebietszerlegung anwenden.

Am Anfang dieses Abschnitts war von einer vorzeichenbehafteten *Distanz*funktion die Rede. Die Funktion ϕ soll also den exakten Abstand eines jeden Punktes im Raum zur freien Oberfläche Γ_f angeben. Die Forderung

$$|\nabla \phi| = 1 \tag{2.34}$$

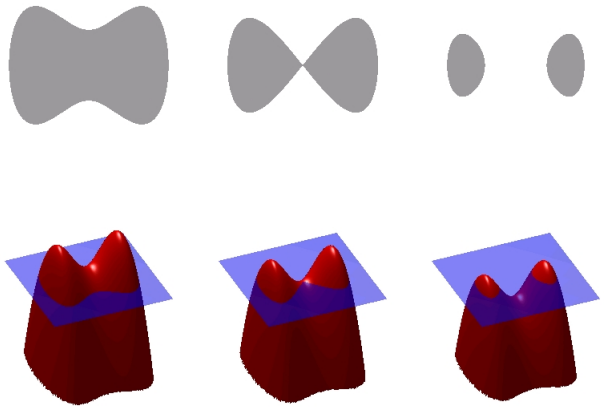


Abb. 2.3: Topologieänderung einer Level-Set–Nullniveaumenge in 2D (Quelle: [31]).

sorgt dafür, dass ϕ tatsächlich immer den Abstand beschreibt. Diese Forderung wird in der Literatur als *Eikonalgleichung* bezeichnet. Durch sie wird die Level-Set–Funktion auch, wie eben erwähnt, zu einer Lipschitz-stetigen Funktion bei zwei aufeinander treffenden Oberflächen.

Die freie Oberfläche soll sich über die Zeit hinweg in Abhängigkeit vom Geschwindigkeitsfeld $\vec{u}$ der Navier-Stokes–Gleichungen bewegen. Hierzu muss die Differentialgleichung

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0 \tag{2.35}$$

mit einer entsprechenden initialen Belegung von ϕ für $t = 0$ gelöst werden. Die initiale Belegung entspricht einer Festlegung, an welcher Stelle im Raum sich zum Beginn einer Simulation die beiden Phasen befinden. Es wird die Aufgabe dieser Arbeit sein, einen einfachen Weg zum Festlegen einer solchen Belegung zu finden und insbesondere in Autodesk Maya anzubieten.

Durch die Lösung von Gleichung (2.35) wird die Bedingung aus Gleichung (2.34) verletzt. Die Lösung dieses Problems besteht in einer Reinitialisierung der Level-Set–Funktion nach einem Zeitschritt der Berechnung von (2.35). In [13] wird ein Reinitialisierungs-Verfahren für ϕ vorgestellt.

Um die Level-Set–Technik in die Navier-Stokes–Gleichungen einzubinden, diese also zweiphasig zu machen, muss man sich von der Annahme entfernen, dass die Dichte ρ und die Viskosität μ im Simulationsgebiet Ω konstant sind. Um genau zu sein muss am freien Rand ein Sprung der beiden Größen vorliegen. Durch die Verwendung der Level-Set–Technik lässt sich der Sprung sehr einfach durch

$$\rho(\phi) = \rho_1 + (\rho_2 - \rho_1) H(\phi)\,, \tag{2.36}$$

$$\mu(\phi) = \mu_1 + (\mu_2 - \mu_1) H(\phi) \tag{2.37}$$

modellieren. In dieser Notation ist dann zum Beispiel ρ_1 die Dichte des Fluids in der ersten Phase. Die Funktion H ist die Heavyside'sche Sprungfunktion

$$H(\phi) := \begin{cases} 0 & \text{falls} \quad \phi < 0\,, \\ \frac{1}{2} & \text{falls} \quad \phi = 0\,, \\ 1 & \text{falls} \quad \phi > 0\,. \end{cases} \tag{2.38}$$

Die genaue Beschreibung der Integration der Zweiphasigkeit und somit der Level-Set–Funktion in die Navier-Stokes–Gleichung und insbesondere auch die Diskretisierung dieser neuen Form werden in [13] sehr detailliert beschrieben.

Ein Vorteil der Level-Set–Technik wurde bisher noch nicht erwähnt. Durch die Darstellung der Grenzschicht zwischen den beiden Phasen als implizite Fläche wird ihre Visualisierung sehr stark vereinfacht. Es gibt sehr effiziente und zuverlässige Verfahren, um implizite Flächen graphisch darzustellen oder zu extrahieren. Verwendet man nun für die beiden Phasen Wasser und Luft, so lässt sich die Wasseroberfläche bei einer Simulation sehr gut bestimmen. Ein wichtiger Teil dieser Arbeit wird es sein, die freie Oberfläche mit entsprechenden Methoden zu extrahieren und in einem späteren Schritt zu visualisieren. Dafür bietet die Level-Set–Technik die ideale Voraussetzung.

2.2.2 Optimierte Diskretisierungsmethoden

In Abschnitt 2.1.2 wurde bereits die Grundidee einer Diskretisierung beschrieben. Für die konkrete Anwendung der Navier-Stokes–Gleichungen haben sich bestimmte Diskretisierungsansätze für den Ort und die Zeit durchgesetzt. So wird im NaSt3DGPF-Löser für die Ortsdiskretisierung ein *Staggered Grid* verwendet. Die Zeitdiskretisierung erfolgt mit einer adaptiven Schrittweitensteuerung. Beide Vorgehensweisen sollen vorgestellt werden, da sie die Implementierung von Erweiterungen für den NaSt3DGPF-Code stark beeinflussen.

Das Staggered Grid

Der vorliegende Löser verwendet ein *Staggered Grid* (zu deutsch: versetztes Gitter) um die Navier-Stokes–Gleichungen im Ort zu diskretisieren. Die Idee besteht darin, nicht jede Größe auf den selben Gitterpunkten darzustellen. Betrachtet man die Diskretisierung des Raumes als eine Unterteilung in *Zellen*, so werden im Löser die Werte für den Druck p, die Dichte ρ, die Viskosität μ und die Level-Set–Funktion ϕ für die Zell-Mittelpunkte gespeichert. Dahingegen wird die Geschwindigkeit $\vec{u} = \begin{pmatrix} u & v & w \end{pmatrix}^t$ für die einzelnen Raumrichtungen getrennt auf den Mittelpunkten der jeweiligen Seitenflächen verwaltet. Dieses Vorgehen illustriert Abbildung 2.4 auf Seite 24. Die Indizierung der Form $u_{i+\frac{1}{2},j,k}$ deutet an, dass die Werte der Geschwindigkeit nicht im Mittelpunkt liegen, sondern erst mittels einer Interpolation wie

$$u_{i,j,k} = \frac{1}{2}(u_{i-\frac{1}{2},j,k} + u_{i+\frac{1}{2},j,k}) \tag{2.39}$$

bei Bedarf für den Mittelpunkt berechnet werden können. Der Grund für die Verwendung dieses Diskretisierungsansatzes liegt in der Reduktion nötiger Interpolationen und numerisch besseren Eigenschaften für die Kopplung von Geschwindigkeit und Druck.

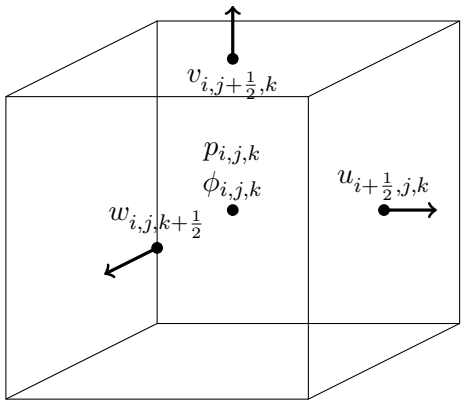


Abb. 2.4: Eine Zelle im *Staggered Grid*.

Adaptive Zeitschrittweiten-Steuerung

Innerhalb eines Zeitschritts darf sich bei der numerischen Berechnung der Navier-Stokes–Gleichungen die *Information* nur maximal um eine Zelle bewegen bzw. ausbreiten. Dies ist die Grundaussage der bereits in Abschnitt 2.1.2 erwähnten CFL-Bedingungen. Unter Information kann man dabei Daten wie Geschwindigkeit, Druck usw. verstehen. Sollte diese Bedingung nicht eingehalten werden, wird die Rechnung instabil.

Um Probleme zu verhindern muss deshalb für jeden Zeitschritt im vornherein abgeschätzt werden, wie weit sich vermutlich die Information, auf Basis der aktuellen Daten, ausbreiten wird. Hierfür werden in [13] entsprechende Berechnungsvorschriften vorgestellt. Mit diesen lässt sich dann festlegen, wie groß der nächste Zeitschritt Δt sein darf, ohne obige Bedingung zu verletzen.

Aus diesem Mechanismus heraus entsteht eine adaptive Zeitschrittweiten-Steuerung, die im NaSt3DGPF-Löser dafür sorgt, dass immer mit einer angemessenen Zeitschrittweite gearbeitet wird.

2.2.3 Simulation auf beliebigen Gebieten

Bisher war immer nur von der Simulation auf einem allgemeinen Gebiet Ω_h die Rede. Im Abschnitt 2.1.3 zur Parallelisierung wurde implizit schon angenommen, dass eine Lösung der Navier-Stokes–Gleichungen auf einem quaderförmigen Gebiet geschieht. Sonst würde eine Gebietszerlegung in kleinere Quader recht schwer werden. Der Navier-Stokes–Löser NaSt3DGPF bietet jedoch auch die Möglichkeit, die Navier-Stokes–Gleichungen auf beliebigen Gebieten zu lösen.

Das hierzu verwendete Verfahren beruht auf einem quaderförmigen diskretisierten Gebiet Ω_h. Für jede Zelle in diesem Gebiet ist es möglich, ein *Flag* zu setzen, das die Zelle eindeutig als eine *Fluid-Zelle* oder eine *Solid-Zelle* markiert. Durch diese Unterscheidung kann man entsprechende Bereiche, Ω_h^{fluid} für die Region, in der sich ein Fluid befinden soll, und Ω_h^{solid} für die Region, die kein Fluid darstellt, finden. So ist es also möglich, in einem Fluid-Quader einzelne Bereiche als fest zu markieren. Dadurch kann man zum Beispiel Objekte in einer Strömung realisieren.

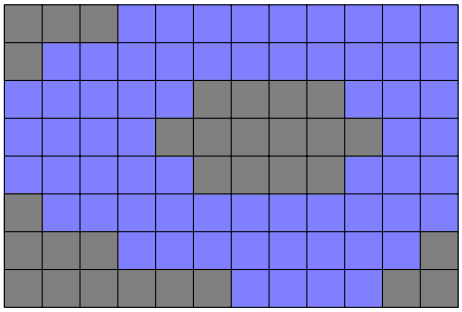


Abb. 2.5: Darstellung einer zweidimensionalen Geometrie mit einem Flag-Feld (graue Zellen stehen für Solid-Zellen).

Ist das Gebiet Ω_h^{fluid} erst einmal bekannt, so können dort die Navier-Stokes–Gleichungen berechnet werden. Für die Zellen aus Ω_h^{solid} wird keine richtige Berechnung durchgeführt. Einzig die Zellen in Ω_h^{solid} die an die Fluid-Region stoßen, fließen in die Randbedingungen für Ω_h^{fluid} ein. Es ist also möglich für Solid-Zellen Randbedingungen wie Rutsch- oder Haftbedingungen vorzugeben, die Einfluss auf das benachbarte Fluid haben.

Man beachte, dass die Kennzeichnung einer Zelle mit einem Solid/Fluid-Flag völlig unabhängig von der Level-Set–Funktion ϕ ist. Diese ist immer auf dem gesamten Raum Ω_h definiert. Paradoxerweise kommt es also dazu, dass solide Bereiche damit auch einen Fluidtyp haben. So kann also anschaulich gesprochen ein Stein durchaus durch die Level-Set–Funktion als Luft oder Wasser bezeichnet werden oder es kann sogar die *Wasseroberfläche* quer durch den Stein verlaufen. Selbstverständlich muss also die Reinitialisierung der Level-Set–Funktion immer auch in Ω_h^{solid} erfolgen.

Unter Einsatz des gerade beschriebenen Verfahrens lassen sich beliebige Geometrien, wie zum Beispiel jene aus Abbildung 2.5 vorgeben und als Simulationsgebiet verwenden. Die Aufgabe dieser Arbeit soll es sein, die Konfiguration von einem Simulationsgebiet sehr einfach zu gestalten. Während im Allgemeinen bevorzugt Dreiecksnetze verwendet werden, um Geometrien zu beschreiben, muss für den Strömungssimulator NaSt3DGPF eine diskrete Geometriedarstellung, also ein Flag-Feld für das Solid/Fluid-Flag, generiert werden. Den Vorgang der Umwandlung eines Netzes in ein Flag-Feld, kann man als *Voxelisierung* bezeichnen. In dieser Arbeit soll ein Verfahren entwickelt werden, das die Voxelisierung durchführt. Dies dient dann der Erzeugung eines Flag-Feldes aus beliebigen polygonalen Netzen in Maya. Es soll also möglich werden, mit Maya ein Gebiet für eine Fluidsimulation zu generieren.

2.2.4 Konfigurationsmöglichkeiten

Der Navier-Stokes–Löser NaSt3DGPF kann mit einer ASCII-Datei konfiguriert werden. In diese können Parameter für die Anfangs- und Randbedingungen, für die Fluideigenschaften, für numerische Verfahren und viele andere Werte eingetragen werden. Einige interessante Konfigurationsmöglichkeiten sollen hier vorgestellt werden. Für ein tiefer gehendes Studium der Konfigurationsmöglichkeiten sei auf [14] verwiesen.

Auflösung

Mit der Konfigurations-Datei kann eine Gitter-Auflösung für die Lösung der Navier-Stokes–Gleichungen gewählt werden. Insbesondere können die nötigen Parameter für ein *uniformes* Gitter angegeben werden. Unter einem uniformen Gitter versteht man ein Gitter mit senkrecht zueinander stehenden Koordinatenachsen und jeweils einer festen Maschenweite pro Koordinatenrichtung. Alternativ sind auch *rektilineare*[9] Gitter möglich. Diese werden aber im Rahmen dieser Arbeit nicht behandelt.

Für die zeitliche Auflösung, also die Zeitschritte, können auch Parameter vorgegeben werden. Man kann eine minimale Zeitschrittweite angeben. Ebenso kann eingestellt werden, ob das adaptive Verfahren eher kleine, oder größere Zeitschritte wählt.

Dimensionslose Rechnung

Neben den anfangs des Kapitels vorgestellten *dimensionsbehafteten* Navier-Stokes–Gleichungen gibt es auch eine *dimensionslose* Ausführung. Dabei versteht man unter dimensionsbehaftet, dass alle Größen in den Navier-Stokes–Gleichungen eine physikalische Einheit haben. Man kann die Gleichungen auch so umschreiben, dass keine Einheiten mehr vorkommen, sie also dimensionslos sind. Eine solche Darstellung ist in bestimmten wissenschaftlichen Zusammenhängen sinnvoll.

In der Konfigurations-Datei können deshalb entweder dimensionsbehaftete Werte für die Fluide, also Viskosität, Dichte, usw. vorgegeben werden oder entsprechende dimensionslose Größen, auf die hier aber nicht weiter eingegangen werden soll. Bei einer dimensionsbehafteten Konfiguration wird zudem eine Größe des Simulationsgebietes im Sinne der physikalischen Größe mit angegeben.

Datenausgabe

Der Navier-Stokes–Löser arbeitet mit einer einzigen Binär-Datei in die er nach einer festen Anzahl von Zeitschritten alle in der Simulation vorkommenden Größen und Datenfelder herausschreibt. Die Anzahl der Zeitschritte kann eingestellt werden. Eine solche Datei kann je nach Auflösung einige Gigabyte groß werden. Die Datei dient der Wiederaufnahme einer zwischenzeitig beendeten Simulation bei dem letzten gespeicherten Zeitpunkt.

Zusätzlich können für feste physikalische Zeiten weitere solche Binär-Dateien erzeugt werden, die für die Visualisierung einzelner Informationen zu verschiedenen Zeitpunkten verwendet werden können. Diese Dateien haben eventuell ebenso Gigabyte-Größen. Schreibt man die Daten an vielen Zeitpunkten heraus, können die Datenmengen damit fast in den Terabyte-Bereich gehen.

Da die herausgeschriebenen Daten in einem proprietären binären Format vorliegen, müssen diese nachträglich pro Datei in herkömmliche Datenformate zur Visualisierung, wie zum Beispiel VTK-Dateien für das Visualisierungs-Tool Paraview, umgewandelt werden.

Ein Ziel dieser Arbeit ist es, den Aufwand für die Visualisierung der Level-Set–Nullniveaumenge erheblich zu reduzieren. So soll direkt zur Laufzeit die freie Oberfläche extrahiert und in platzsparender polygonaler Form in Dateien herausgeschrieben werden. Diese sind dann nur

[9] Hier ist die Maschenweite pro Koordinatenrichtung variabel.

noch maximal ein paar Megabytes, aber keine Gigabytes groß. Das herausgeschriebene Datenformat ist zudem eines der Standardformate für polygonale Netze, so dass keine aufwendigen Konvertierungsschritte mehr erfolgen müssen. Auf Visualisierungs-Tools, wie Paraview, kann ebenso verzichtet werden, da ein Plugin für Maya die 3D-Meshes automatisiert einladen wird.

Anfangs- und Randwerte

Eine Festlegung von Anfangs- und Randwerten erfolgt im Navier-Stokes–Löser über geometrische Objekte. Dabei seien unter Anfangswerten auch die Solid/Fluid-Flags zu verstehen, die das Simulationsgebiet definieren. Der Konfigurationsmechanismus des Lösers kennt verschiedene geometrische Objekte, wie Boxen (Quader), Sphären, Zylinder oder allgemeine Polytope. Aus diesen lassen sich mittels der CSG-Operationen *Vereinigung*, *Differenz* und *Schnitt* beliebige Objekte erzeugen. Intern werden diese Objekte in eine voxelisierte Darstellung (siehe dazu Abschnitt 2.2.3) überführt.

Für jedes Objekt lassen sich bestimmte Eigenschaften angeben. So kann beispielsweise ein Objekt Zellen beschreiben auf denen Rutsch-Randbedingungen gelten. Ebenso lässt sich angeben, dass ein Objekt für einen Fluid-Bereich steht, der ein Gas oder eine Flüssigkeit enthält. Dadurch wird neben den Solid/Fluid-Flags auch eine initiale Belegung für die Level-Set–Funktion gefunden. Initiale Geschwindigkeiten für das auf den Fluid-Zellen definierte Geschwindigkeitsfeld lassen sich so natürlich auch vorgeben.

Neben den geometrischen Objekten können auch alternativ Dateien eingebunden werden, in denen bereits voxelisierte Darstellungen einer Geometrie enthalten sind. Für solche Darstellungen können ebenso die Eigenschaften für Rand- und Anfangsbedingungen vergeben werden. Dieses Interface wird in dieser Arbeit benutzt, um eigene, in Maya voxelisierte Geometrien, in die Konfiguration des Strömungslösers einzubinden.

Beispiel

Im Folgenden soll ein kurzes Beispiel einer Konfigurationsdatei für den Löser angegeben werden. Das Beispiel wurde den mit dem Löser mitgelieferten Beispielen entnommen und leicht modifiziert.

Beispiel 2.1:

```
//
// Beispiel: Wasserkanal mit einem Hindernis
//

dimension {
    length     <0.6, 0.25, 0.4>  // physikalische Größe
    resolution <60 ,25, 40>      // Auflösung des Diskretisierungsgitters
}

parameter {
    prstep    10                 // Anzahl der Zeitschritte, nach denen die
                                 // Binär-Datei geschrieben wird
    iteropt   20                 // gewünschte Anzahl an BiCGStab-Iterationen
    itermax   200                // maximale Zahl an BiCGStab-Iterationen
    eps       1.e-3              // Abbruchswert für Genauigkeit des
```

```
                                  // Residuums im BiCGStab-Löser
    omega     1.70                // Relaxations-Parameter
    Tfin      1.0                 // physikalische Endzeit der Simulation
    deltmax   0.5                 // maximale Zeitschrittweite
    tfdiff    0.2                 // Sicherheitsfaktor für adaptive
                                  // Zeitschrittweiten-Berechnung
    tfconv    0.2                 // CFL-Zahl (für adaptive
                                  //           Zeitschrittweite)

    mu_l      1.0002e-3           // Viskosität der Flüssigkeits-Phase Ω_2
    mu_g      17.2e-6             // Viskosität der Gas-Phase Ω_1
    rho_l     998.2               // Dichte der Flüssigkeits-Phase Ω_2
    rho_g     1.2929              // Dichte der Gas-Phase Ω_1

    sigma     0.0                 // Oberflächenspannungs-Koeffizient
    thickness 1.9                 // Dicke des geglätteten Bereiches um die
                                  // freie Oberfläche

    gx        0.0                 // Gravitation
    gy       -9.8
    gz        0.0

    alpha     1                   // Parameter für Diskretisierung

    zeta      1.0                 // Parameter für Initialisierung der Level-Set-
                                  // Funktion


    TargetDirectory   /tmp/       // Ausgabeverzeichnis der Binärdateien
                                  // für Visualisierungszwecke
    TimePrintStep     0.01        // physikalische Zeitschrittweite für Ausgabe
                                  // der Binärdateien

    dimensionless     OFF         // Wahl, ob dimensionslos gerechnet wird
    ReinitOrder       3           // Parameter für Level-Set-Reinitialisierung
    ReInitStepsOpt    5           // Parameter für Level-Set-Reinitialisierung
    TimeDis           AB2         // Wahl der Zeitdiskretisierung
    ConvectiveTerms   VONOS       // Wahl der Ortsdiskretisierung für Teile der
                                  // Navier-Stokes-Gleichungen
    PoissonSolver     BiCGStab    // Wahl eines numerischen Lösers für lineare
                                  // Gleichungssysteme
    FixedOutflow      0.1, north      // Fixierung der freien Oberfläche an
                                      // einer Seite der Simulationsbox
}


// Beschreibung des Bereiches mit "Wasser"
box{
 coords<0,0,0>,<60,10,40>
 init <1.0,0.0,0.0>,0.0
```

```
 fluid
 liquid
}

// Festlegung der "Luft"-Region
box{
 coords<1,11,1>,<60,25,40>
 init <0.0,0.0,0.0>,0.0
 fluid
 gas
}

// Beschreibung des Hindernisses
// (Haftbedingungen werden standardisiert gesetzt)
box{
  coords<15,1,18>,<20,10,23>
}

box{
  coords<15,11,18>,<20,22,23>
}

// fixe Einström-Randbedingung
box{
 coords<0,1,1>,<0,10,40>
 inflow <1.0,0.0,0.0>
}

// natürliche Ausström-Randbedingung
box{
 north
 inout 2,1
}

// Rutschbedingungen für verschiedene "Wände" des Simulationsgebiets
box{
 east
 slip
}

box{
 west
 slip
}

box{
 top
 slip
}
```

```
box{
 bottom
 slip
}
```

Mit dem Beispiel wird eine zweiphasige Simulation von Luft und Wasser aufgesetzt, bei der das Wasser um ein Hindernis herum strömt. Abbildung 2.6 zeigt eine photorealistische Visualisierung dieser Strömung.

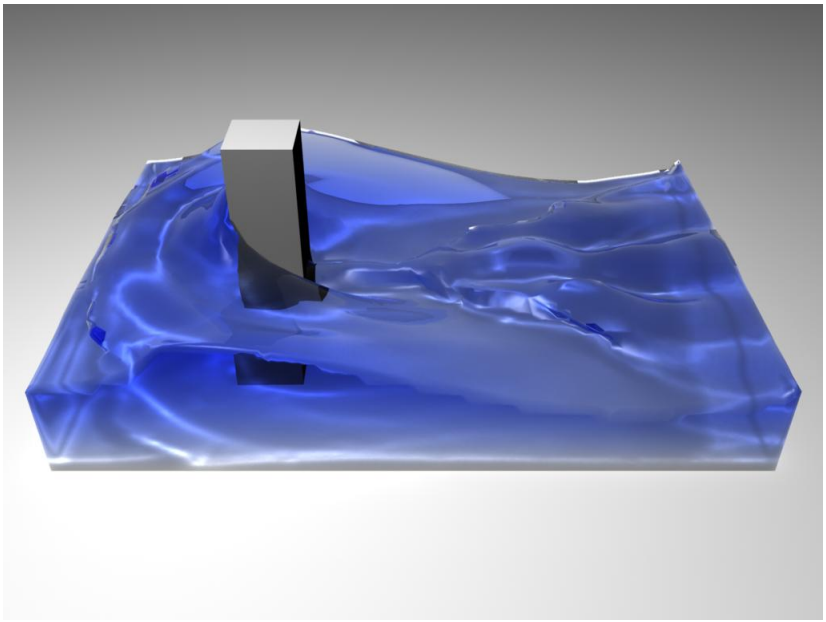


Abb. 2.6: Visualisierung des Simulationsbeispiels zum Zeitpunkt $t \approx 0.6\,s$.

Bei der Betrachtung der Konfigurations-Datei wird sehr schnell klar, dass eine korrekte Konfiguration einer Simulation recht komplex und zum Teil auch umständlich ist. Mit dieser Arbeit entsteht ein Plugin für die Animationssoftware Maya, mit dem das händische Erstellen solcher Konfigurations-Dateien entfällt. Vielmehr wird in Maya ein intuitives Interface zum Navier-Stokes–Löser integriert, mit dem sich alle nötigen Parameter einstellen lassen. Die Geometrie für Anfangs- und Randbedingung kommt dann direkt aus Maya. Insgesamt reichen mit dem Plugin schon wenige Maus-Klicks um eine komplexe Simulationsumgebung zu erschaffen.

2.3 Das Graphik- und Animationsprogramm Autodesk Maya

In diesem Abschnitt soll ein kurzer Überblick über das Graphik- und Animationsprogramm Autodesk Maya gegeben werden. Besonderes Augenmerk wird dabei auf dessen Erweiterbarkeit und die damit verbundenen Frameworks gerichtet.

Maya ist ein von der Firma Autodesk vertriebenes Programm zur Gestaltung von dreidimensionalen Szenen und Animationen. Es bietet dem Benutzer eine Umgebung, in der er 3D-

Objekte über polygonale Netze oder über NURBS-Flächen modellieren kann. Hierzu steht eine umfangreiche Palette an Werkzeugen zur Verfügung. Die modellierten Geometrien lassen sich zeitabhängig auf beliebige Weise verschieben, deformieren, rotieren oder skalieren. Hierdurch werden Animationen realisiert. Auf die Oberflächen der Objekte kann man einen großen und beliebig erweiterbaren Satz an Materialien und Texturen anwenden, so dass die Geometrien nach einem Rendering möglichst wie echte Gegenstände aussehen.

Für das Rendering einer zusammengestellten dreidimensionalen Szene lassen sich unterschiedlichste Lichtquellen und Beleuchtungseffekte vorgeben. Außerdem werden in den Szenen, ganz wie in einem echten Studio, Kameras eingesetzt. Diese können beliebig platziert, bewegt und modifiziert werden. Das Sichtfeld der Kameras kann für jeden beliebigen Zeitpunkt gerendert werden. Aus Sequenzen von solchen gerenderten Bildern entstehen ganze Animationen oder Filme.

In Maya kommt entweder ein im Zusammenhang mit dem Programm von Autodesk entwickelter Renderer oder das ebenfalls mitgelieferte Rendering-Paket *Mental Ray* zum Einsatz. Der letztgenannte Renderer zählt zu den leistungsfähigsten Programmen dieser Art und liefert Funktionen auf dem neuesten Stand der Technik.

Neben den bisher genannten Funktionen bringt Maya, je nach eingesetzter Version, Werkzeuge zur Animation von verschiedenen physikalisch motivierten Effekten mit. So sollen Partikelsysteme zum Beispiel Feuer, Wasser oder Rauch möglichst realistisch animieren. Genauso lassen sich inzwischen auch Stoffe oder Haare im Computer nachbilden.

Die vorliegende Arbeit kann als Erweiterung dieser Palette verstanden werden. So soll es nun auch möglich sein, physikalisch korrekte zweiphasige Fluidsimulationen in Maya durchzuführen. Diese Funktionalität soll sich möglichst natürlich in die vorhandene Umgebung integrieren, so dass der zusätzliche Lernaufwand minimal wird.

2.3.1 Die Arbeitsumgebung

In Abbildung 2.7 auf Seite 32 wird die Arbeitsumgebung von Autodesk Maya gezeigt. Auf der linken Seite findet sich die dreidimensionale Szene in der gearbeitet wird. Diese wird bei dem hier gezeigten Darstellungsmodus perspektivisch gezeigt. Die gitterartige Grundfläche dient dem Benutzer als Bezugspunkt im dreidimensionalen Raum. Im Bild ist im Mittelpunkt dieser Grundfläche schon ein Objekt platziert und selektiert. Dabei handelt es sich um eine über Polygone modellierte Sphäre.

Abbildung 2.7 zeigt auf der rechten Seite der Arbeitsumgebung den sogenannten *Attribute Editor*. Dieses Bedienelement ermöglicht es, Eigenschaften des gerade selektierten Objektes zu verändern. So kann für die Sphäre in der Abbildung beispielsweise der Radius und die Feinheit des polygonalen Meshes modifiziert werden. Der *Attribute Editor* ist eines *der* zentralen Bedienelemente von Maya.

Im oberen Bereich des Fensters befinden sich verschiedene Toolbars. Über Icons lassen sich dort bestimmte Funktionen aufrufen. Es ist möglich, die Toolbars und auch die eigentliche Menüzeile beliebig zu konfigurieren. Außerdem kann man zwischen verschiedenen Belegungen für beide Bedienelemente umschalten. So erhält der Benutzer trotz eines knappen Bildschirmplatzes einen Zugang zu allen Funktionen.

Unterhalb der 3D-Szene ist ein weiteres wesentliches Werkzeug von Maya: die *Timeline*. Die Timeline ermöglicht es dem Benutzer auf einfache Weise zwischen verschiedenen Zeitpunkten

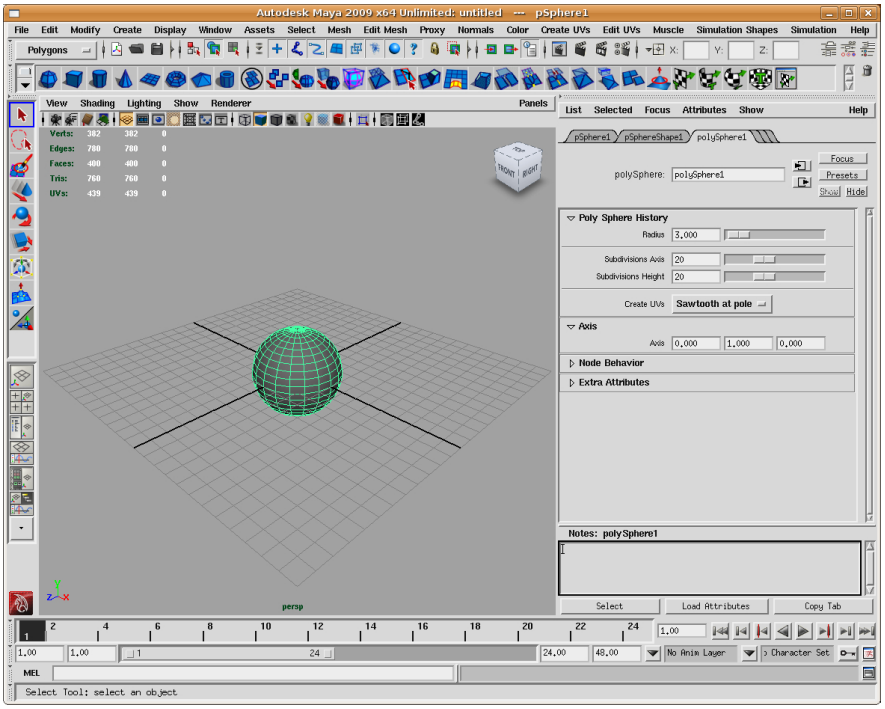


Abb. 2.7: Die Arbeitsumgebung von Maya.

in der Simulation hin- und herzuspringen. Das geschieht über die Leiste, die in Abbildung 2.7 aufsteigend mit den Zahlen 1 bis 24 beschriftet ist. Abhängig von der angewählten Position wird das jeweilige *Frame*, also das jeweilige Einzelbild in der Animation, aufgerufen. Die Timeline wird vor allem beim Animieren der Objekte in der Szene gebraucht.

Selbstverständlich bietet Autodesk Maya noch einen sehr viel größeren Umfang an Werkzeugen und Bedienelementen an. Da diese aber für die in dieser Arbeit vorgestellte Anwendung von geringerer Bedeutung sind, sei bei weitergehendem Interesse an dieser Stelle auf entsprechende Standardwerke [52], [21] zu Maya verwiesen.

2.3.2 Die Architektur

Im Rahmen dieser Diplomarbeit soll Maya um einige Funktionen erweitert werden. Um dies zu können und um allgemein einige Grundkonzepte von Maya besser nachzuvollziehen, soll an dieser Stelle die Architektur des Programms anhand einiger wesentlicher Grundkomponenten diskutiert werden. Die folgenden Ausführungen orientieren sich an dem Buch [37].

Die Architektur von Maya lässt sich in drei wesentliche Ebenen aufteilen. Als obere Ebene fungiert dabei die *Benutzeroberfläche*. Alle dortigen Eingaben werden in Kommandos der Maya-eigenen Skriptsprache MEL umgewandelt. Auf der zweiten Ebene steht der *Interpreter* für MEL-Skripte. Dieser verarbeitet die übergebenen Kommandos. Typischerweise führt der Aufruf eines Kommandos dazu, dass Veränderungen an dem sogenannten *Dependency Graph* vorgenommen werden. Der Dependency Graph bildet die untere logische Schicht in der Architektur von Maya.

Knoten im Dependency Graph

Der *Dependency Graph* oder auch Abhängigkeits-Graph ist das Kernstück von Maya. Eine 3D-Szene in Maya ist aus technischer Sicht nur ein spezieller Dependency Graph. Jeder solche Graph besteht aus untereinander verbundenen Knoten bzw. *Nodes*. Knoten sind in Maya einzelne, völlig voneinander unabhängige Einheiten, die einerseits Daten speichern und andererseits darauf Berechnungen durchführen können. Daten werden in einem Knoten in sogenannten *Attributen* gespeichert. Dies sind Datenfelder mit einem Namen und einem Datentyp.

In einem Knoten können einzelne Attribute voneinander abhängen. Es kann also vorgegeben werden, dass ein Attribut B aus einem Attribut A hervorgeht. Dies führt dazu, dass bei einer Änderung von A, das andere Attribut vollautomatisch neu *berechnet* wird. Eine solche Berechnung erfolgt bei Knoten in der Methode `compute`. Diese stellt die *Intelligenz* des Knotens dar.

Durch das Verknüpfen von Attributen lassen sich einzelne Knoten miteinander verbinden. Angenommen das Attribut B aus dem vorher genannten Beispiel wäre mit einem Attribut C eines anderen Knotens verbunden. Dann würde bei einer Änderung von Attribut A die Berechnung für B ausgeführt und das Ergebnis automatisch an C propagiert.

Auf diese Weise lassen sich komplexe Graphen aus beliebig vielen Knoten herstellen. Jeder einzelne Knoten hat nur eine begrenzte Funktionalität, aber zusammen genommen, können nahezu beliebige Aufgaben durchgeführt werden. Abbildung 3.3 auf Seite 46 zeigt einen Ausschnitt aus einem Dependency Graph.

Unter den Knoten gibt es noch eine wichtige Unterklasse: Die sogenannten *DAG-Knoten*. DAG steht für *direct acyclic graph*. Ein direkter azyklischer Graph ist eine Verallgemeinerung von einem Baum, bei der es mehr als einen Vater pro Knoten geben darf. Die spezielle Klasse der DAG-Knoten hat zusätzlich zu den normalen Knoteneigenschaften von Maya, die Möglichkeit Hierarchien zu definieren. So lässt sich ein DAG-Knoten nicht nur über die Attribute verknüpfen, sondern auch über Vater- oder Kind-Relationen hierarchisch eingliedern. Dies hat in manchen Zusammenhängen wesentliche Vorteile, da zum Beispiel das Entfernen eines Vater-Knotens automatisch auch zum Entfernen seiner Kinder führt. Zu beachten ist aber, dass nicht jeder Knoten im Dependency Graph auch ein DAG-Knoten ist.

Beispiele von Standard-Knoten

Nach dieser allgemeinen Übersicht ist es nun sinnvoll, einige wenige Standard-Knoten vorzustellen: `transform`-Knoten zählen zu den wichtigsten Standard-Knoten. Solche Knoten sind DAG-Knoten. Sie führen Koordinatentransformationen von Attributen mit lokalen Koordinaten in globale Koordinaten durch. Bei diesen Transformationen werden Translationen, Rotationen, Scherungen und Skalierungen angewendet, die sich über Attribute des Knotens vorgeben lassen.

Jedes Objekt, das in irgend einer Art und Weise in einer dreidimensionalen Szene in Maya dargestellt oder platziert werden soll, wird einem `transform`-Knoten in einer DAG-Hierarchie untergeordnet. Soll die Position des Objekts verändert werden, geschieht das über den `transform`-Knoten, der dem Objekt-Knoten übergeordnet ist.

Objekte in einer dreidimensionalen Szene werden typischerweise über `shape`-Knoten repräsentiert, die ebenfalls der Klasse der DAG-Knoten angehören. Es gibt verschiedenste Arten

von `shape`-Knoten. So können diese Knoten für polygonale Netze, NURBS-Flächen oder auch Kameras im Raum stehen.

Eine andere Art von Knoten ist ein `locator`-Knoten. Dieser DAG-Knoten hat als Besonderheit, dass er im Gegensatz zu einem einfachen Knoten eine graphische Repräsentation in der Darstellung der 3D-Szene in Maya hat. Wohlgemerkt wird diese aber nicht bei dem Rendering der Szene gezeichnet. Wie der Name schon sagt, dient der Knoten häufig dazu, eine Position oder einen Bereich in der 3D-Szene zu markieren.

Kommandos und Manipulatoren

Neben den vorgestellten Grundkomponenten des Maya-Frameworks gibt es noch zwei weitere, wesentliche Konzepte, auf die hier eingegangen werden soll, da sie auch in der vorliegenden Arbeit zur Anwendung kommen.

Das erste wesentliche Konzept sind *Kommandos*. Wie eingangs erwähnt ist die gesamte Maya-Oberfläche letztendlich nur ein graphisches Interface zu der Skriptsprache MEL. Diese liefert von Haus aus viele Kommandos beziehungsweise Befehle mit. Jeder Befehl ist eine eigenständige Einheit, die von außen einzelne Knoten oder gleich den gesamten Dependency Graph manipulieren kann. Nur durch Kommandos wird das praktische Arbeiten mit dem Dependency Graph möglich.

Als zweites wichtiges Konzept sollen hier noch *Manipulatoren* vorgestellt werden. Manipulatoren sind mit einem Knoten assoziiert. Sie dienen als graphische Alternative zum *Attribute Editor* um Attribute des Knotens zu verändern. So werden Manipulatoren in die 3D-Szene gezeichnet und lassen sich mit der Maus beeinflussen. Verändert man die Position, Größe, Ausrichtung,... des Manipulators, verändert sich entsprechend ein Attribut, das er kontrolliert.[10]

Selbstverständlich beschränkt sich das Framework aus dem sich die Applikation Maya zusammensetzt nicht nur auf die hier vorgestellten Grundkomponenten. Dennoch soll an dieser Stelle auf die Beschreibung weiterer Einzelaspekte verzichtet werden, da nur die hier vorgestellten Konzepte für das grundlegende Verständnis dieser Arbeit wichtig sind. Für ein tiefergehendes Interesse sei auf die bereits erwähnte Quelle [37] verwiesen.

2.3.3 Erweiterungsmöglichkeiten

Autodesk Maya bietet in seiner aktuellen Version[11] drei verschiedene Wege zur Erweiterung und Steuerung von Maya mittels Programmierung an. Dies sind die Maya-eigene Skriptsprache *MEL*, eine C++ - API und seit relativ kurzer Zeit eine Anbindung an die Skriptsprache Python.

Die Skriptsprache MEL

Die *Maya Embedded Language* ist eine eigens für Maya entwickelte Skriptsprache. Wie schon im Abschnitt zur Architektur von Maya erwähnt, bildet sie die Basis für die Benutzerinteraktion mit den Kernkomponenten. Die gesamte Oberfläche von Maya wird über Skripte realisiert. Entsprechend lässt sich diese auch über die Sprache MEL in beliebiger Art und Weise erweitern.

[10] Für ein praktisches Anwendungsbeispiel sei auf Abschnitt 3.2.3 verwiesen.
[11] Zum Zeitpunkt der Entstehung dieser Arbeit ist dies die Version *2009*.

Dies reicht von der Erzeugung eigener Menüs über die Entwicklung von sogenannten *Attribute Editor Templates* zur Verwaltung von Attributen eines neu erstellten Knotens im *Attribute Editor* bis hin zum kompletten Neuschreiben einer Oberfläche für Maya.

MEL ist aber nicht nur für die Erweiterung der Benutzeroberfläche wichtig. Da alle über die Oberfläche getätigten Eingaben intern in eigene MEL-Befehle umgewandelt werden, kann der gesamte Funktionsumfang, der auch über das Interface verfügbar ist, über MEL ausgenutzt werden. Gleichzeitig ermöglichen die für Skriptsprachen typischen Konstrukte wie Variablen, bedingte Abfragen oder Schleifen, dass sich viele Arbeitsabläufe im hohen Maße automatisieren lassen.

Die C++ - API

Während sich die Skriptsprache MEL vornehmlich um die automatisierte Verwendung von bereits bestehenden Komponenten kümmert, sollen mit der C++ - API von Maya vollständig neue Basiskomponenten geschaffen werden. So können über die API neue Kommandos, neue Manipulatoren, neue Knoten und vieles mehr geschrieben werden. Diese bündelt der Programmierer in *Plugins*, die sich zum Programmpaket Maya hinzuladen lassen und benutzt werden können.

Die API arbeitet dabei Objekt-orientiert. Bereits vorhandene Basis-Knoten stehen als Klassen zur Verfügung. Durch deren Ableitung können neue Knoten entwickelt werden. So kann beispielsweise ein allgemeiner Knoten durch das Ableiten von der Klasse `MPxNode` implementiert werden. Für neue *locator*-Knoten steht eine von `MPxNode` abgeleitete Klasse `MPxLocatorNode` zur Verfügung. Der große Vorteil der Objekt-Orientierung besteht in diesem Fall in der Leichtigkeit, mit der neue Knoten erzeugt werden können, ohne alle Mechanismen neu zu programmieren. Damit reduziert sich die Arbeit meist auf die Definition der im Knoten enthaltenen Attribute und das Überschreiben, der `compute`-Methode, die die Berechnungen des Knotens durchführt. Gleichzeitig ergibt sich durch die Verwendung von Klassen ein klar definiertes Framework.

Selbstverständlich lassen sich auch eigene Manipulatoren oder Kommandos über die API entwickeln. Kommandos implementiert man wie auch Knoten über Klassen: Jedes Kommando wird von der Klasse `MPxCommand` abgeleitet. Die Funktionalität des Kommandos ist über die bei der Implementierung überschriebende Methode `doIt` zu realisieren. `doIt` wird beim Aufruf des Kommandos gestartet.

Kommandos ermöglichen es dem Programmierer, beliebige C++ - Operationen in Maya auszuführen. Dies können Aufrufe von externen Programmbibliotheken sein, oder auch einfache Algorithmen. Zugriffe auf Knoten oder den Dependency Graph werden auch über Kommandos realisiert.

Um hier ein klar geregeltes Interface zu haben, verwendet Maya die sogenannten *Function Sets*. Alle wesentlichen Komponenten des Animationsprogramms werden von der Klasse `MObject` abgeleitet. Deshalb betrachtet man zunächst jeden Knoten als ein ganz allgemeines *Objekt*. Erst durch das Verknüpfen dieses Objektes mit einem *Function Set*, kann auf die Daten beziehungsweise die Eigenschaften des Knotens zugegriffen werden. Im Kontext der Objekt-Orientierung sind Function Sets von der Mutterklasse `MFn` abgeleitete Klassen. Sie stellen für Objekte bestimmte Methoden zur Verfügung.

Es gibt unterschiedlichste Function Sets für verschiedene Objekttypen. Beispielsweise gibt

es eines für alle Knoten-Typen, die im Dependency Graph vorkommen können: `MFnDependencyNode`. Hierüber lassen sich Informationen über Knoten abfragen, aber auch neue erzeugen oder manipulieren. Hochspezialisiert ist `MFnMesh`. Dieses Function Set kann nur auf Objekte angewendet werden, die polygonale Netze repräsentieren. Hiermit lassen sich zum Beispiel sämtliche Kanten des Netzes abfragen, löschen oder verändern.

Neben den Function Sets gibt es noch ein weiteres wesentliches Konzept: `Iterator`-Klassen. Um Zugriff auf ein Objekt zu erhalten muss es erst einmal bekannt sein. Deshalb liefern die Instanzen von bestimmten Iterator-Klassen zum Beispiel alle Knoten des Abhängigkeits-Graphen oder alle Objekte, die in der 3D-Szene markiert wurden.

Ganz allgemein ist die Möglichkeit ein beliebiges Kommando für Maya programmieren zu können von sehr großer Bedeutung. Auf den ersten Blick scheint es keine Begründung für die gemeinsame Existenz der Skriptsprache MEL und des Konzeptes der in C++ implementierbaren Kommandos zu geben: Die mit Maya mitgelieferten Kommandos sollten eine ausreichende Ausdrucksstärke haben, um alle benötigten Arbeiten durchzuführen. Das ist aber nicht der Fall. Zusammen bieten die beschriebenen Function Sets wesentlich mehr und tiefer gehende Funktionen an, als die Kommandos für MEL. Ein Einbinden von externen Bibliotheken ist mit MEL ebenfalls nicht möglich. Die für diese Arbeit entscheidende Einschränkung ist aber die Geschwindigkeit der Sprache. Wie jede Skriptsprache wird MEL interpretiert. Sollen Algorithmen auf große Datenmengen angewendet werden, ist die Leistung des Interpreters völlig unzureichend.

Aus den gerade besprochenen Gründen soll in dieser Arbeit so viel wie möglich direkt mit der API von Maya gearbeitet werden. Nur die wenigen Anteile der Programmierarbeit, die nicht über die API realisiert werden können, werden in MEL implementiert. Einer dieser Anteile ist die Erweiterung der Benutzeroberfläche und insbesondere des Attribute Editors. Überraschenderweise ist es unmöglich, dies über die API durchzuführen.[12]

Das Python-Interface

Neben den beiden schon länger verfügbaren Programmierinterfaces bestehend aus MEL und der C++ - API, bietet Autodesk Maya seit wenigen Versionen[13] eine Anbindung an die Programmiersprache Python an. Diese neuere Einrichtung versucht die zum Teil recht starke Trennung zwischen den Anwendungsbereichen von MEL und der API zu überbrücken: Über Python lassen sich auf der einen Seite die von MEL bekannten Kommandos ansprechen. Andererseits können aber auch viele API-Funktionen aufgerufen werden und sogar Python-Skripte als neue Knoten eingebunden werden.

Das Python-Interface scheint vielversprechend zu sein. Schwierig wird die Arbeit aber gerade dadurch, dass die Schnittstelle neu ist. Auf der einen Seite sind immer noch nicht alle Funktionen eins zu eins auch in Python verfügbar. So muss noch etwas mehr Arbeit seitens der Entwickler von Maya investiert werden. Auf der anderen Seite wird die schwierige Dokumentationslage (siehe nächster Abschnitt) für die Programmierbarkeit von Maya noch dadurch

[12] Es können zwar auch MEL-Befehle über die API abgesetzt werden und somit auch die Benutzeroberfläche gestaltet werden. Das führt aber zu einem eher unübersichtlichen Programmierstil.

[13] Das Python-Interface existiert seit Maya 8.5. In dieser Arbeit kommt Maya 2009 zum Einsatz, das zwei Versionssprünge nach der Version 8.5 liegt.

verschlechtert, dass Python als Programmierschnittstelle zu neu ist und somit weniger Dokumentationen als für MEL und die API verfügbar sind.

Als Ausschlusskriterium muss aber, wie auch bei MEL, die Geschwindigkeit von Python betrachtet werden. Weil Python-Programme interpretiert werden, können Sie auf großen Datenmengen kaum so schnell arbeiten, wie dies kompilierte Programme können. Da aber in dieser Arbeit gerade *mit* solchen großen Datenmengen gearbeitet wird, muss erneut der API-Zugriff mit C++ bevorzugt werden.

2.3.4 Literatur

Eine unerwartet große Hürde ergibt sich beim Versuch Literatur zum Thema *Programmierung von Maya* zu finden. Während Standardwerke wie zum Beispiel [21] noch kurz auf die Programmiersprache MEL eingehen und es Bücher gibt, die sich nur diesem Thema widmen [47], findet sich nur ein einziger Autor der Bücher zum Thema API-Programmierung veröffentlicht hat. David Gould gibt in seinem zweibändigen Werk [37] und [38] zur Programmierung von Maya eine Einführung in die API-Programmierung. Gleichzeitig ist aber der erste und grundlegende Band des Werkes zum Zeitpunkt der Entstehung dieser Arbeit schon sechs Jahre alt, so dass einige Informationen über Maya und die API wegen häufiger Versionssprünge bereits veraltet sind.

Neben dieser Quelle bietet die Firma Autodesk frei verfügbar nur die im Hilfesystem verankerte API-Dokumentation und eine Klassenreferenz sowie einige Beispiel-Plugins an.

Als zusätzliche Hauptinformationsquelle für die Maya-API muss deshalb die Maya-*Gemeinde* im Internet gelten. Besonders hervorzuheben ist der Internet-Auftritt von Robert Bateman [4], bei dem eine umfangreiche Anzahl an sehr übersichtlichen und gut dokumentierten Beispiel-Plugins zu finden ist. Ein Howto mit sehr nützlichen Informationen findet sich auf der Seite von Bryan Ewert [23]. Ebenfalls empfehlenswert ist ein Blick auf den Internet-Auftritt von Michael B. Comet [11].

Zusätzlich zu den persönlichen Seiten ist das Maya-Forum auf der Computer-Graphik–Seite *Highend3d* [40] sehr zu empfehlen. Über die Suchfunktion ist hier ein großer Fundus an guten Ratschlägen zu finden.

Alle vorgenannten Quellen sind im Rahmen der Implementierung von Erweiterungen für Maya in diese Arbeit eingeflossen. Wegen der hohen Verbreitung von Autodesk Maya überrascht es aber, dass es nicht mehr Literatur zu dem Thema gibt. Ein aktuelles, praxisorientiertes Grundlagenwerk könnte hier den Einstieg noch deutlich erleichtern.

3 Interaktive Konfiguration von Fluidsimulationen

In dieser Arbeit wird die *Konfiguration* des Strömungslösers NaSt3DGPF vereinfacht. Die Konfiguration besteht aus der Festlegung von Simulationsparametern, Randbedingungen und Fluid- sowie Hindernisgeometrien. Für ein einfaches und interaktives Zusammenstellen einer Fluidsimulation im Sinne eines *virtuellen Labors* soll Autodesk Maya erweitert werden. Mit einer graphischen Benutzeroberfläche wird das wenig intuitive, manuelle Erstellen einer textbasierten Datei für die Konfiguration des Lösers abgelöst. Durch die einfache Bedienbarkeit wird der Einsatz von komplexen Geometrien in Strömungssimulationen möglich. Die für eine Erweiterung notwendigen Komponenten werden in diesem und dem nachfolgenden Kapitel konzipiert und ihre Implementierung vorgestellt.

Zunächst werden im vorliegenden Kapitel die Grundstruktur und der gewünschte Funktionsumfang der über Plugins realisierten Erweiterung zur Konfiguration beschrieben. Einen wichtigen Anteil an den Plugins haben verschiedene Maya-Knoten: ein Knoten zur Verwaltung der Gesamtsimulation und weitere Knoten für die Geometrie-Repräsentation und die einfache Angabe von Randbedingungen. Im vorliegenden Kapitel werden diese Knoten vorgestellt.

Aus den in den Knoten gesammelten Informationen muss eine Szenenbeschreibung für den NaSt3DGPF-Löser gewonnen werden. Ein Hauptaspekt hierbei ist die Entwicklung einer geeigneten Technik zur Übertragung von polygonalen Geometrien aus dem Animationsprogramm in eine für den Strömungssimulator geeignete Darstellung. Das Schlagwort hierzu ist die *Voxelisierung*. Zur besseren Übersicht wird die Erklärung des Umwandlungsprozesses aus den Informationen in Maya in eine geeignete Szenenbeschreibung in einem separaten Kapitel besprochen.

In dem nun folgenden Kapitel wird abschließend eine Erweiterung zum Aufruf des Simulationsprogramms aus Maya beschrieben. Sie ermöglicht eine einfache Steuerung des Lösers.

3.1 Entwurf einer Erweiterung für Maya

3.1.1 Gewünschter Funktionsumfang

Eine Erweiterung zur Konfiguration von Strömungssimulationen in Maya soll, wie eingangs erwähnt, sehr einfach zu bedienen sein. Die Idee besteht darin, einen quaderförmigen Bereich in einer Maya-Szene zu markieren und als Simulationsgebiet zu definieren. Nur innerhalb dieser Umgebung soll eine Berechnung durchgeführt werden. Des weiteren soll es möglich sein, alle wichtigen Parameter des Strömungslösers durch Anwahl des Simulations-Bereichs einzustellen. Als sinnvolles Interface zu den Parametern soll ein Template für den *Attribute Editor* dienen. Selbstverständlich müssen alle dort angebbaren Werte automatisiert überprüft werden, so dass keine fehlerhaften Eingaben möglich sind.

Objekte, die als Hindernisse in der Strömungsrechnung vorkommen, werden über polygonale Formen dargestellt. Diese können wie üblich erzeugt und in den Simulationsbereich platziert werden. Durch einen Menübefehl werden sie der Simulationsumgebung als Hindernisse (Solide) zugewiesen. Ein Attribute Editor - Template ermöglicht die Auswahl der Rutsch- beziehungsweise Haft-Randbedingung.

Über die Konfigurationsoptionen der Simulationsumgebung soll vorgegeben werden können, ob das Gebiet grundsätzlich mit einem Gas oder einer Flüssigkeit gefüllt ist.[1] Hiervon abweichende Fluidbereiche werden mit Hilfe von polygonalen Objekten definiert, die mit einem Menübefehl die Fluid-Eigenschaft zugewiesen bekommen. Das zum Fluidbereich zugehörige Template erlaubt die Vorgabe des Fluidtyps sowie die Festlegung von den Anfangsbedingungen für Geschwindigkeit und Druck. Zum besseren Verständnis soll als Beispiel kurz die Modellierung eines Wassertropfens in einer mit Luft gefüllten Simulationsumgebung skizziert werden: Zunächst wird für die Simulationsumgebung als Standard-Fluidtyp das Gas *Luft* vorgegeben. Anschließend wir der Tropfen als polygonale Späre konstruiert und positioniert. Die Zuweisung der Fluid-Eigenschaft und die Vorgabe, eine Flüssigkeit zu sein, macht die Sphäre schließlich in der Simulation zu einem Wassertropfen.

Für eine dynamische Fluidsimulation muss es möglich sein, Flüssigkeiten oder Gase an bestimmten Stellen einströmen oder abfließen zu lassen. Die Einström- oder Abfließ-Bereiche sollen in der Simulationsumgebung markiert werden können. Dabei müssen die Markierungen entweder an den Außenflächen der quaderförmigen Simulationsumgebung oder auf Hindernissen vorgenommen werden. Derart gekennzeichnete Bereiche sollen ebenfalls über den Attribute Editor konfiguriert werden, so dass die Einströmungsrichtung und die Geschwindigkeit oder aber das Ausströmverhalten vorgegeben werden können.

Schließlich muss die Möglichkeit bestehen, aus der zusammengestellten Simulation eine Szenenbeschreibung oder auch Konfiguration für den Strömungslöser zu erzeugen. Auch dies geschieht über einen Menübefehl vollautomatisch. Das Initialisieren und Starten der Strömungssimulation soll ebenfalls per Knopfdruck geschehen und die Bildschirmausgabe des Simulationsprogramms muss über ein Fenster überwacht werden können.

3.1.2 Aufteilung in Knoten und Plugins

Die Programmierschnittstelle von Autodesk Maya erfordert einen sehr hohen Grad an Modularisierung. Deshalb muss die im vorangegangenen Abschnitt beschriebene Funktionalität in mehrere Teilkomponenten aufgeteilt werden. Als Grundstock sollte ein Knoten implementiert werden, der die Position und Größe der Simulationsumgebung festlegt und gleichzeitig alle Basisparameter der Simulation abspeichert. Aus diesen Daten muss ein Kommando eine Konfiguration für die Simulation erzeugen können. Beide Komponenten, der Knoten und das Kommando, sollen in einem Plugin zusammengefasst werden.

Davon völlig unabhängig müssen Knoten entwickelt werden, die Fluid- und Solid-Eigenschaften beschreiben und an polygonale Netze angekoppelt werden. Es ist sinnvoll diese zusammen mit Knoten zur Verwaltung der Randbedingungen in einem Plugin zu vereinen, da sie zusammen alle Anfangs- und Randbedingungen der Simulation vorgeben. Insgesamt sollen also zwei Plugins implementiert werden.

[1] Soforn entsprechende Viskositätswerte vorgegeben werden, können auch zwei Gase oder zwei Flüssigkeiten simuliert werden.

In der nun folgenden Beschreibung der Implementierung der Plugins wird zunächst der Knoten für die Simulationsumgebung beschrieben. Hieran schließt sich eine Darstellung der Knoten für die Anfangs- und Randbedingungen an. Das Kommando zur Erzeugung der Szenenbeschreibung für den Strömungslöser wird erst im nachfolgenden Kapitel beschrieben. Ein solches Vorgehen ist sinnvoll, da das Kommando die Informationen von allen hier beschriebenen Knoten benutzt. Zudem bedarf es bei dem Kommando umfangreicher Erläuterungen, so dass ein eigenes Kapitel der besseren Übersicht dient.

3.2 Implementierung des Knotens zur Verwaltung der Simulationsumgebung

Der Knoten für die Repräsentation der Simulationsumgebung trägt den Namen `SimulationArea`. Er wird abgeleitet von einem `locator`-Knoten. Ein solcher Knoten beschreibt eine Position in einer Maya-Szene und hat eine geometrische Ausdehnung. Das Wichtigste an diesem Knotentyp ist die Fähigkeit, als graphisches Element in einer Szene gezeichnet werden zu können. In dem hier besprochenen Anwendungsfall sollen die Außenränder der Simulationsumgebung in der Szene dargestellt werden.

Eine Implementierung der Simulationsumgebung besteht aus vier Komponenten. Zunächst muss ein Kommando implementiert werden, das den `SimulationArea`-Knoten erzeugt. Die Hauptkomponente ist der `SimulationArea`-Knoten. Außerdem sollte der Benutzer die Größe der Simulationsumgebung direkt in der Szene durch eine Modifikation der Größe des gezeichneten Kubus verändern können. Hierfür bedarf es der Implementierung von *Manipulatoren*. Schließlich muss, wie eingangs erwähnt, ein Attribute Editor - Template geschrieben werden, damit sämtliche für die Simulation benötigten Parameter über das Benutzerinterface verändert werden können. Im Folgenden werden die Komponenten beschrieben.

3.2.1 Ein Kommando zur Erzeugung des Knotens

Es ist immer sinnvoll ein eigenes Kommando für die Erzeugung eines Knotens zu implementieren. Dies bietet mehr Flexibilität als der Standardbefehl `createNode`, mit dem gewöhnlich in Maya Knoten erzeugt werden: So können mit der Generierung auch gleichzeitig weitere Befehle abgesetzt werden oder zusätzlich benötigte Knoten oder Verbindungen hergestellt werden.

Das zu implementierende Kommando `createSimulationArea` stellt zunächst den `SimulationArea`-Knoten durch den Aufruf einer Methode der Klasse `MDagModifier` her. Anschließend wird der erzeugte Knoten automatisiert selektiert. Schließlich wird die Funktionalität des `transform`-Knotens der `SimulationArea` derart eingeschränkt, dass nur noch deren Position beeinflusst werden kann. Die Größe soll nicht durch den `transform`-Knoten, sondern über Manipulatoren kontrolliert werden. Ein Rotieren oder Scheren der Simulationsumgebung muss unterbunden werden. Die Beschränkung des `transform`-Knotens geschieht über die Möglichkeit, ein Attribut zu blockieren.

3.2.2 Details zum Knoten

Der `locator`-Knoten `SimulationArea` entsteht durch das Ableiten der Klasse `MPxLocatorNode`. Er enthält die Parameter der Simulation, die nicht direkt mit den Rand- und Anfangsbe-

Attribut	Typ	Beschreibung
`sizeX`	`kDistance`	Größe der Simulationsumgebung in X-Richtung
`sizeY`	`kDistance`	Größe der Simulationsumgebung in Y-Richtung
`sizeZ`	`kDistance`	Größe der Simulationsumgebung in Z-Richtung
`voxelCountX`	`kInt`	Auflösung der Simulationsumgebung in X-Richtung
`voxelCountY`	`kInt`	Auflösung der Simulationsumgebung in Y-Richtung
`voxelCountZ`	`kInt`	Auflösung der Simulationsumgebung in Z-Richtung
`duration`	`kDouble`	Dauer der Simulation (in Sekunden)
`shapes`	`kByte`	Attribut, über das sich Knoten für die Anfangs- und Randbedingungen ankoppeln
`defaultFluidType`	`kByte`	Standard-Fluidart, mit der die Simulationsumgebung gefüllt ist (Gas bzw. Flüssigkeit)
`projectDirectory`	`kString`	Ausgabeverzeichnis der zu erstellenden Dateien für die Konfiguration

Tabelle 3.1: Eine Auswahl von wenigen wichtigen Attributen des `SimulationArea`-Knotens.

dingungen zusammenhängen. Diese werden bei der Initialisierung des Knotens als Attribute erzeugt und mit den vom Strömungslöser bekannten Standardeinstellungen belegt. Da die Liste der Attribute dieses Knotens zu lang ist, soll hier auf eine tabellarische Darstellung aller Werte verzichtet werden. Tabelle 3.1 enthält daher nur eine Auswahl wichtiger Attribute mit Beschreibung.

Die drei Attribute zur Bestimmung der Größe der Simulationsumgebung bilden die Basis der graphischen Darstellung des `SimulationArea`-Knotens in der Maya-Szene. Durch das Überschreiben der Methode `draw` kann eine beliebige Geometrie für den Knoten in die Szene gezeichnet werden. In diesem Fall soll ein Quader die Abmessungen der Simulationsumgebung anzeigen. Das Zeichnen geschieht mittels geeigneter OpenGL-Befehle. Die Darstellung der `SimulationArea` in einer Maya-Szene findet sich in Abbildung 3.1 auf Seite 43.

Für die Szenenverwaltung benötigt Autodesk Maya eine Bounding-Box von dem gezeichneten Objekt. Diese wird mit der Größe des Quaders definiert und kann über die überschriebene Methode `boundingBox` angefordert werden.

Auch wenn die Struktur dieses Knotens auf den ersten Blick sehr einfach scheint, ist eine vollständig korrekte Implementierung eine Herausforderung. Dies liegt an der Zahl von fast hundert Attributen, die verwaltet werden müssen.

3.2.3 Manipulatoren

Maya bietet die Möglichkeit mit den schon in Abschnitt 2.3.2 erwähnten Manipulatoren Werte von Attributen über das graphische Benutzerinterface zu verändern. Manipulatoren sind gezeichnete Objekte in einer Szene, die mit der Maus oder der Tastatur beeinflusst werden können. Um sie für einen neuen Knoten bereitzustellen, muss eine Klasse von der Klasse `MPxManipContainer` abgeleitet werden. Eine solche initialisiert und verwaltet als ein Container die Manipulatoren und definiert, welche Attribute eines angebundenen Knotens verändert werden sollen.

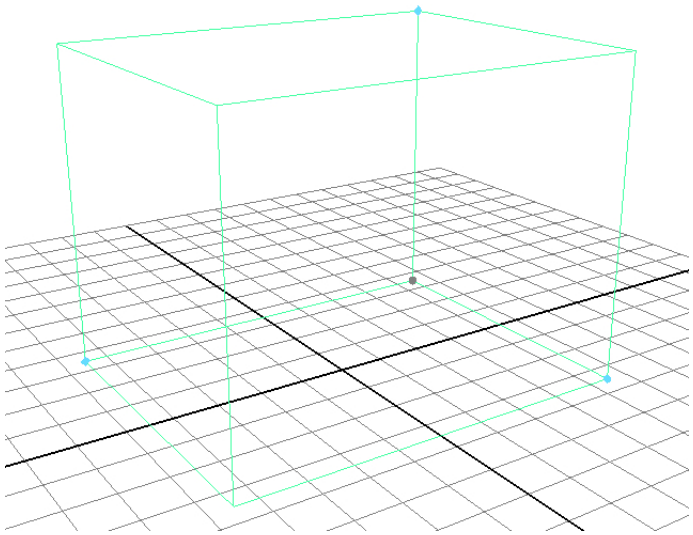


Abb. 3.1: Der `SimulationArea`-Knoten in einer Maya-Szene mit Manipulatoren für die Größe.

Für den `SimulationArea`-Knoten werden in der Klasse `SimulationAreaManip` drei sogenannte *Distance*-Manipulatoren verwaltet, die die Größe der Simulationsumgebung in die jeweilige Raumrichtung verändern. Durch das Erben von der Basisklasse werden dem Programmierer viele Details erspart. So muss beispielsweise für Standardmanipulatoren, wie den hier verwendeten Distance-Manipulatoren, kein neues graphisches Objekt zur Interaktion gezeichnet werden. Maya benutzt einfach einen kleinen Kreis, der markiert und bewegt werden kann. Es muss nur noch vorgegeben werden, wie dieser sich bewegen darf. Für den Manipulator des `sizeX`-Attributs sind zum Beispiel nur Bewegungen in X-Richtung erlaubt. In Abbildung 3.1 können die Manipulatoren für die Größe der `SimulationArea` als blaue Kreise an drei Ecken des Quaders erkannt werden.

3.2.4 Ein Attribute Editor - Template

Die Implementierung des Attribute Editor - Templates für die `SimulationArea` hat sich als sehr zeitaufwendig erwiesen. Für fast hundert Attribute müssen Eingabefelder erzeugt werden. Im Vordergrund steht dabei die besonders einfache Handhabung durch den Benutzer und die Fehlertoleranz gegenüber falschen Eingaben. So werden beispielsweise Parameter-Eingabefelder für aktuell nicht verwendete Funktionen des Strömungslösers ausgegraut.[2] Parameter, die verschiedene Wahlmöglichkeiten anbieten, werden über Dropdown-Listen realisiert. Funktionen, die an- oder abgeschaltet werden müssen, werden ebenfalls über Dropdown-Listen oder mit Check-Boxen kontrolliert. Attribute, bei denen Pfade oder Dateien gewählt werden müssen, erhalten ein Texteingabefeld mit einem zusätzlich aufrufbaren Dialogfeld für die Verzeichnis-

[2] Gerade das Ausgrauen von Eingabeflächen erweist sich als sehr hartnäckiges Problem: Es existiert ein Bug in Maya, der zum Teil die Reaktivierung ausgegrauter Schaltflächen verhindert. Ein Bug-Report wurde an die Firma Autodesk geschickt, von dieser aber auch nach Monaten noch nicht weiter bearbeitet.

Attribut	Typ	Beschreibung
`boundaryCondition`	`kByte`	Randbedingung des Hindernisses (Haft- oder Rutsch-Bedingung)
`attachedMesh`	`kMesh`	Mesh Knoten, dem die Solid-Eigenschaft zugewiesen wird
`simulationArea`	`kByte`	Attribut, das an den `SimulationArea`-Knoten angebunden wird

Tabelle 3.2: Aufstellung der Attribute des `SimulationSolidShape`-Knotens.

bzw. Dateiauswahl. Die große Zahl der veränderbaren Einstellungen wird am besten bei der Betrachtung des implementierten Templates klar. Hierzu sei auf Abbildung 3.2 auf Seite 45 verwiesen.

3.3 Knoten zur Verwaltung von Anfangs- und Randbedingungen

Es müssen Knoten zur Verwaltung von Anfangs- und Randbedingungen implementiert werden. Auf der einen Seite sind dies Knoten, die aus polygonalen Netzen Hindernisse oder Fluid-Bereiche in der Simulation machen. Auf der anderen Seite müssen Ein- und Ausströmungsbedingungen festgelegt werden. Die Implementierung dieser Knoten wird nachfolgend beschrieben.

3.3.1 Hindernisse

Der zu entwickelnde Knoten vom Typ `SimulationSolidShape` wird von der Klasse `MPxNode` abgeleitet und ist damit ein Standardknoten. Mit diesem soll ein geschlossenes polygonales Mesh in Maya als ein Hindernis in einer Strömung ausgewiesen werden. Tabelle 3.2 gibt die Attribute des Knotens an. Das einzige für den Anwender interessante Attribut ist `boundaryCondition`. Über dieses kann gewählt werden, ob für das Hindernis Haft- oder Rutsch-Bedingungen gelten sollen.

Einem `mesh`-Knoten kann mit dem zu implementierenden Befehl `attachSolidShape` ein `SimulationSolidShape`-Knoten zugewiesen werden. Damit wird dieser, wie erwähnt, zu einem Hindernis- oder Solid-Bereich in einer Strömungssimulation. Der Befehl kann mit zwei oder ohne Parameter aufgerufen werden. Ist kein Parameter angegeben, werden ein selektierter `mesh`-Knoten und eine ausgewählte `SimulationArea` erwartet. Bei zwei Parametern werden die Namen der beiden sonst selektierten Knoten erwartet. Neben dem Knoten für das Mesh wird auch der Knoten der Simulationsumgebung benötigt, damit bei mehreren `SimulationArea`-Knoten das Hindernis der richtigen Simulationsumgebung zugewiesen wird.

In der Implementierung des Befehls werden zunächst die übergebenen Parameter verarbeitet. Bevor ein neuer Knoten erzeugt wird, überprüft eine Routine, ob dem Mesh schon ein anderer Knoten für einen Solid- oder Fluid-Bereich zugewiesen ist. Ist dies der Fall, wird abgebrochen. Andernfalls wird ein neuer `SimulationSolidShape`-Knoten erzeugt. Dessen Attribut `attachedMesh` wird mit dem Attribut `outMesh` des `mesh`-Knotens verbunden. Für die Anbindung an die `SimulationArea`, wird deren `shapes`-Attribut an das `simulationArea`-Attribut des neuen Knotens angeschlossen.

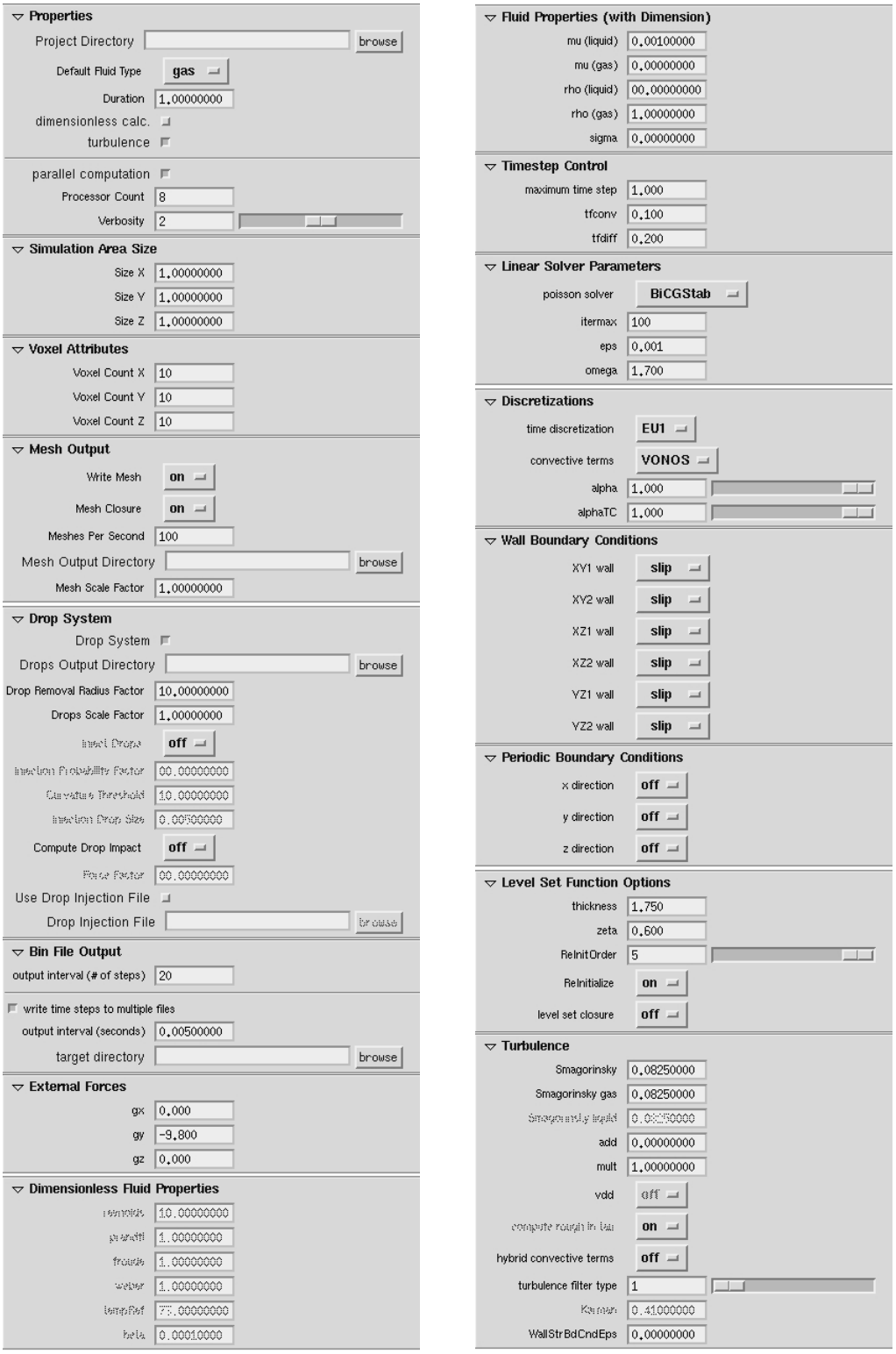


Abb. 3.2: Das *Attribute Editor* - Template zur `SimulationArea`.

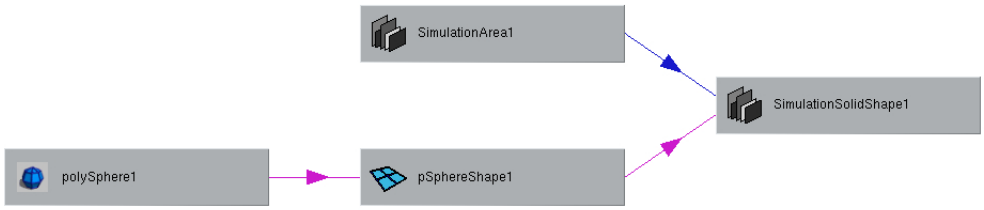


Abb. 3.3: Darstellung der Verbindungen eines `SimulationSolidShape`-Knotens nach der Ausführung des `attachSolidShape`-Kommandos.

Attribut	Typ	Beschreibung
`fluidType`	`kByte`	Phase des Fluids (Gas oder Flüssigkeit)
`initialVelocity`	`k3Double`	initiale Geschwindigkeit des Fluids (vektoriell)
`initialPressure`	`kDouble`	initialer Druck
`attachedMesh`	`kMesh`	Mesh, dem die Fluid-Eigenschaft zugewiesen wird
`simulationArea`	`kByte`	Attribut, das an den `SimulationArea`-Knoten angebunden wird

Tabelle 3.3: Beschreibung der Attribute des `SimulationFluidShape`-Knotens.

Wendet man den gerade beschriebenen Befehl auf das Mesh `polySurfaceShape1` und den Knoten `SimulationArea1` an, ergibt sich die in Abbildung 3.3 gezeigte Struktur der Knoten. Die Graphik zeigt die Verbindungen zwischen den einzelnen Knoten, wie sie in der *Hypergraph*-Ansicht von Autodesk Maya gezeigt werden.

Selbstverständlich ist der neue Befehl *undo*-fähig. Außerdem steht für das einfache Ändern der Attribute ein Template für den *Attribute Editor* zur Verfügung.

3.3.2 Fluidbereiche

Mit dem `SimulationFluidShape`-Knoten soll, wie in Abschnitt 3.1.1 gewünscht, einem geschlossenen Mesh die Fluid-Eigenschaft zugewiesen werden. Bei der Erzeugung der Strömungssimulator-Szenenbeschreibung wird für den durch das Mesh vorgegebenen Bereich eine Fluid-Region mit entsprechenden Flüssigkeits- oder Gas-Eigenschaften generiert.[3] Die Attribute des Knotens sind in Tabelle 3.3 beschrieben.

Für den Benutzer sind hier die Attribute `fluidType`, `initialVelocity` und `initialPressure` von Bedeutung. Das erste gibt an, welcher Fluid-Phase die Fluid-Region in der Strömungssimulation angehören soll. Die beiden anderen Attribute sind selbsterklärend.

Analog zum `SimulationSolidShape`-Knoten wird für `SimulationFluidShape` ein Kommando `attachFluidShape` implementiert. Dieses verhält sich exakt so wie `attachSolidShape`, weshalb hier auf eine nochmalige Erläuterung verzichtet wird. Ein Template dient auch bei dem `SimulationFluidShape`-Knoten der einfacheren Kontrolle der Attribute.

[3] Grundlegende Details hierzu finden sich im Grundlagenkapitel in Abschnitt 2.2.4.

Attribut	Typ	Beschreibung
`sizeX`	`kDistance`	Größe des Einströmungsbereiches in X-Richtung
`sizeY`	`kDistance`	Größe des Einströmungsbereiches in Y-Richtung
`sizeZ`	`kDistance`	Größe des Einströmungsbereiches in Z-Richtung
`orientation`	`kInt`	Orientierung der Zuflussfläche (parallel zur X-Y- / X-Z- / Y-Z-Ebene)
`direction`	`kInt`	Haupt-Einströmungsrichtung
`inputVelocity`	`k3Double`	Geschwindigkeit des Fluids
`inflowFluidType`	`kByte`	Fluid-Phase des einströmenden Fluids
`timeVelo`	`kByte`	Parameter zum Ein-/Ausschalten der zeitabhängigen Geschwindigkeit *(Eine Beschreibung dieser Funktion folgt in Abschnitt 4.4.3.)*
`timeVeloSamplesPerSecond`	`kInt`	Anzahl der Geschwindigkeits-Samples pro Sekunde, die bei zeitabhängigen Geschwindigkeiten gemacht werden
`simulationArea`	`kByte`	Attribut, an das die Simulationsumgebung angekoppelt wird

Tabelle 3.4: Beschreibung der Attribute des `SimulationInflowBoundary`-Knotens.

3.3.3 Einfache Einströmrandbedingungen

Der Knoten `SimulationInflowBoundary` soll Bereiche in der Simulationsumgebung markieren, in denen ein Fluid einströmen soll. Genauer gesagt, soll eine *Dirichlet-Randbedingung* vorgegeben werden können. `SimulationInflowBoundary` wird abgeleitet von der Klasse `MPx-LocatorNode` und ist damit genau wie die `SimulationArea` ein `locator`-Knoten. Gezielt wird bei der Implementierung die Form der Einströmungsbereiche auf Flächen reduziert, die zu den *Wänden* der Simulationsumgebung parallel sind und deren Ränder achsenparallel sind. Dies ist die einfachst mögliche Form einer solchen Randbedingung und erleichtert die Erzeugung einer entsprechenden Szenenbeschreibung für den Strömungssimulator enorm. Für beliebig geformte und orientierte Einflussbereiche sei auf den Knoten `SimulationCustomInflowBoundary` verwiesen, der im nächsten Abschnitt beschrieben wird.

In Tabelle 3.4 findet sich die Beschreibung der Attribute des neuen Knotens. Als erstes fällt bei den Parametern ins Auge, dass hier Größen in drei verschiedene Richtungen für den Einströmbereich einstellbar sind, obwohl eigentlich nur ein Rechteck die Fläche beschreiben soll. Tatsächlich wird in Abhängigkeit von der vorgegebenen Orientierung, also der Angabe gegenüber welcher Wand die Fläche parallel sein soll, immer eine der Größen zu Null gesetzt.

Die Implementierung des Knotens läuft wie folgt ab. In der Zeichenroutine des neuen Knotens wird ein Quader mit den Größe `sizeX`×`sizeY`×`sizeZ` gemalt. Dadurch aber, dass eine

Abb. 3.4: Die Konstellation der Knoten nach der Ausführung des `createInflowBoundary`-Kommandos.

der Größen immer Null ist, wird ein Rechteck gezeichnet. Aus technischen Gründen[4] muss zusätzlich zur Orientierung eine Hauptrichtung des einströmenden Fluids angegeben werden. Unter der Hauptrichtung versteht man hier die Angabe, auf welcher Seite der Einströmfläche eine Flüssigkeit oder ein Gas erzeugt werden soll. Diese Richtung wird zusätzlich durch einen auf der jeweiligen Seite senkrecht stehenden Strich gezeigt (siehe Abbildung 3.5 auf Seite 49).

Um die Abhängigkeit des Knotens von der Orientierung zu gewährleisten, wird eine Callback-Routine `attributeChangedCB` implementiert, die auf Veränderungen des Parameters `orientation` reagiert. Diese setzt die Größe in der jeweiligen Richtung zu Null und blockiert deren Veränderbarkeit. Wird auf eine andere Orientierung umgeschaltet, werden die Größen so zwischen den Raumrichtungen umkopiert, dass die Einströmfläche umgeklappt, aber ihre Grundfläche beibehalten wird. Zur einfachen Einstellung der Größe des Einströmbereiches werden für den `SimulationInflowBoundary`-Knoten Manipulatoren zur Verfügung gestellt. Deren Implementierung erfolgt genau wie bei den Manipulatoren des `SimulationArea`-Knotens. Auch in diesem Fall wird mit Manipulatoren für drei Raumrichtungen gearbeitet. Die jeweils zu Null gesetzte Größe kann korrekterweise nicht verändert werden, da sie, wie zuvor erwähnt, blockiert wird.

Die Erzeugung des `SimulationInflowBoundary`-Knotens erfolgt über ein neu erstelltes Kommando `createInflowBoundary`. Als Parameter bekommt dies die Simulationsumgebung über die MEL-Kommandozeile oder als Selektion übergeben. Die Vorgehensweise bei der Implementierung des Befehls ist analog zu den in den vorherigen Abschnitten beschriebenen Kommandos: Erst werden die Parameter verarbeitet. Dann wird der Knoten erzeugt und die Simulationsumgebung mit dem `simulationArea`-Attribut verbunden. Schließlich wird der Knoten selektiert und es findet eine Blockade der meisten Funktionen des Transform-Knotens analog zur `SimulationArea` statt. Durch die Anwendung des neuen Befehls auf die Simulationsumgebung `SimulationArea1` ergibt sich die in Abbildung 3.4 dargestellte Konstellation der Knoten. Abbildung 3.5 zeigt eine Simulationsumgebung in Maya mit einer eingefügten rechteckigen Einströmbedingung.

Auch für den `SimulationInflowBoundary`-Knoten wird ein Template für den *Attribute Editor* erzeugt. Dieses dient der einfachen Veränderung der Parameter.

3.3.4 Beliebig geformte Einströmrandbedingungen

Bereits im letzten Abschnitt wurde darauf hingewiesen, dass der dort beschriebene `SimulationInflowBoundary`-Knoten nur eine sehr eingeschränkte Möglichkeit zur Vorgabe von Einströmungsbereichen anbietet. Diesem Zustand wird durch den Knoten `SimulationCustomInflowBoundary` Rechnung getragen. Mit ihm wird es möglich, beliebig geformte und orientierte

[4] Diese Gründe werden genauer im nächsten Kapitel in Abschnitt 4.4.3 besprochen.

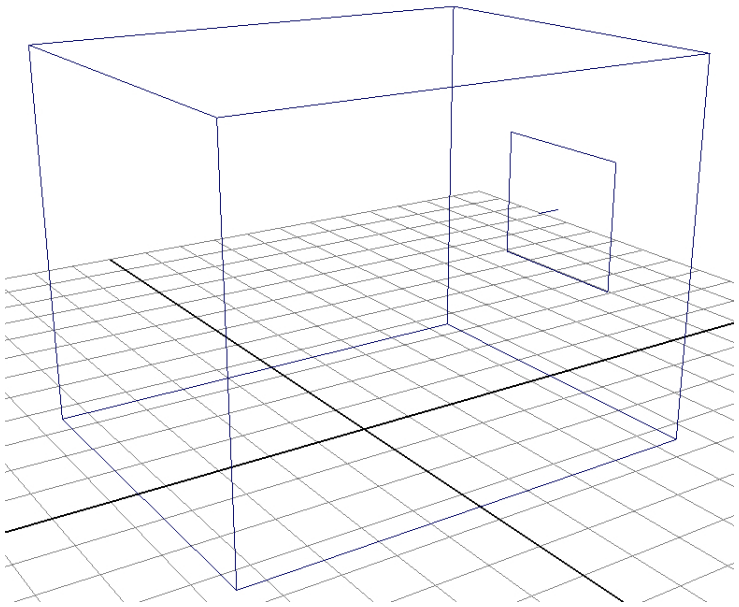


Abb. 3.5: Eine Simulationsumgebung in Maya mit einem `SimulationInflowBoundary`-Knoten zur Vorgabe eines rechteckigen Einflussbereiches.

Einflussbereiche anzugeben. Hierzu definiert der neue `locator`-Knoten ebenfalls eine rechteckige Fläche, die aber beliebig rotiert werden kann. Ihr Schnitt mit einem geschlossenen polygonalen Netz, an das der Knoten angekoppelt wird, bildet die Einströmregion. Diese etwas komplexe Konstruktion muss durchgeführt werden, um eine planare[5], beliebig geformte, Einflussfläche vorzugeben. Abbildung 3.6 auf Seite 50 zeigt ein Anwendungsbeispiel des Knotens, bei dem ein kreisrunder Einströmungsbereich durch den Schnitt mit einem Zylinder definiert wird. Der Knoten ist in dem Bild grün gezeichnet.

In Tabelle 3.5 auf Seite 50 werden die Attribute des Knotens beschrieben. Hier kann man erkennen, dass sich die Attribute nur in geringem Maße von den Parametern des `SimulationInflowBoundary` unterscheiden: Wegen der beliebigen Orientierung, die über eine Rotation mit dem zugehörigen `transform`-Knoten realisiert wird, wird in diesem Knoten nur die Abmessung des Rechtecks in U- und V-Richtung angegeben. Ein Parameter für die Orientierung ist nicht nötig. Dafür kommt aber das `attachedMesh`-Attribut hinzu.

Das Zeichnen des `SimulationCustomInflowBoundary`-Knotens geschieht ähnlich wie bei den anderen Knoten, beruht aber auf den zwei Größenangaben in U- und V-Richtung. Analog werden auch nur Manipulatoren für diese Größen realisiert. Ein leicht modifiziertes Template für den *Attribute Editor* wird ebenfalls erstellt.

Erzeugt wird der Knoten mit dem Kommando `attachCustomInflowBoundary`, das als Parameter oder mittels Selektion den `mesh`-Knoten und den `SimulationArea`-Knoten übergeben bekommt. An diese beiden Knoten wird der neu generierte Knoten automatisch angekoppelt.

[5]Die Forderung nach der Planarität der Fläche hat technische Gründe und wird in Abschnitt 4.4.4 genauer erklärt.

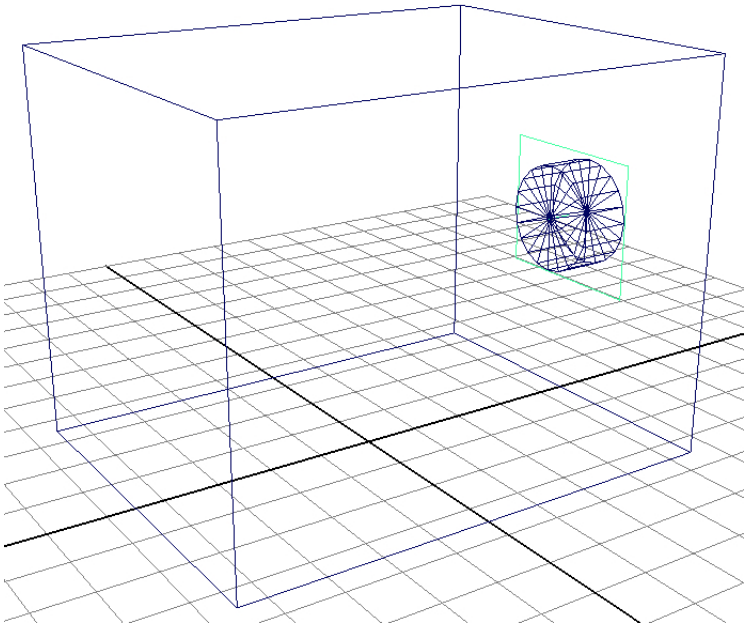


Abb. 3.6: Darstellung des `SimulationCustomInflowBoundary`-Knotens (hier grün gezeichnet): Hier wird ein kreisrunder Einflussbereich vorgegeben.

Attribut	Typ	Beschreibung
`sizeU`	`kDistance`	Größe des Einströmungsbereiches in U-Richtung
`sizeV`	`kDistance`	Größe des Einströmungsbereiches in V-Richtung
`direction`	`kInt`	Haupt-Einströmungsrichtung
`inputVelocity`	`k3Double`	Geschwindigkeit des Fluids
`inflowFluidType`	`kByte`	Fluid-Phase des einströmenden Fluids
`timeVelo`	`kByte`	Parameter zum Ein-/Ausschalten der zeitabhängigen Geschwindigkeit *(Eine Beschreibung dieser Funktion folgt in Abschnitt 4.4.3.)*
`timeVeloSamplesPerSecond`	`kInt`	Anzahl der Geschwindigkeits-Samples pro Sekunde, die bei zeitabhängigen Geschwindigkeiten gemacht werden
`simulationArea`	`kByte`	Attribut, an das die Simulationsumgebung angekoppelt wird
`attachedMesh`	`kMesh`	Mesh, mit dem die Rechtecksfläche geschnitten wird

Tabelle 3.5: Beschreibung der Attribute des `SimulationCustomInflowBoundary`-Knotens.

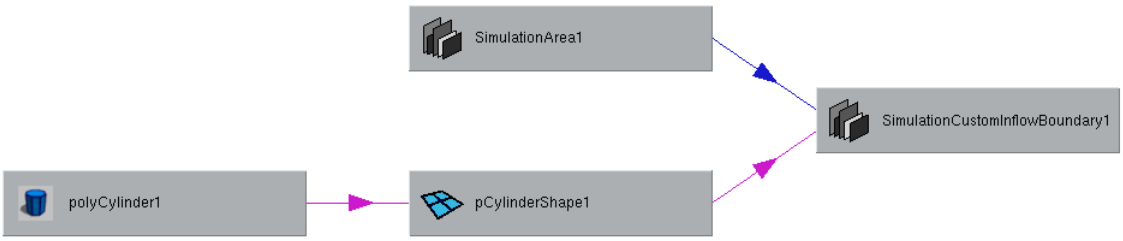


Abb. 3.7: Verbindungen des `SimulationCustomInflowBoundary`-Knotens nach der Ausführung des `createCustomInflowBoundary`-Kommandos.

Attribut	Typ	Beschreibung
`sizeX`	`kDistance`	Größe des Ausströmungsbereiches in X-Richtung
`sizeY`	`kDistance`	Größe des Ausströmungsbereiches in Y-Richtung
`sizeZ`	`kDistance`	Größe des Ausströmungsbereiches in Z-Richtung
`orientation`	`kInt`	Orientierung der Abfließfläche (parallel zur X-Y- / X-Z- / Y-Z-Ebene)
`direction`	`kInt`	Haupt-Ausströmungsrichtung
`inOutType`	`kInt`	Art des Ausströmbereiches
`velocityCorrection`	`kInt`	Art der vorgenommenen Geschwindigkeitskorrektur
`simulationArea`	`kByte`	Attribut, an das die Simulationsumgebung angekoppelt wird

Tabelle 3.6: Aufstellung der Attribute des `SimulationInOutBoundary`-Knotens.

In Abbildung 3.7 kann man entsprechend erstellte Verbindungen erkennen. Beim Blockieren der Funktionen des `transform`-Knotens bleiben in diesem Fall die Parameter für die Rotation und Position veränderbar, um die beliebige Orientierung möglich zu machen.

3.3.5 Ausströmrandbedingungen

Wenn ein Fluid in eine Simulationsregion einströmt, muss eine Flüssigkeit oder ein Gas an einer anderen Stelle abfließen. Diese Forderung ergibt sich aus der Beschränkung der Simulation auf inkompressible Fluide und der damit verbundenen konstanten Dichte. Zur Markierung eines Abfließbereichs dient der `SimulationInOutBoundary`-Knoten. Um genau zu sein markiert dieser eine *homogene Neumann-Randbedingung*.

Die Implementierung des Knotens verläuft vollständig analog zum `SimulationInflowBoundary`. Es soll also eine zu den Rändern des Simulationsgebiets parallele Fläche mit achsenparallelen Kanten über einen `locator`-Knoten vorgegeben werden. Der einzige Unterschied zwischen den beiden Knoten besteht in einigen wenigen abweichenden Attributen, die speziell auf diesen Typ von Randbedingung zugeschnitten sind. Hierzu betrachte man Tabelle 3.6, die eine Aufstellung aller Attribute des Knotens enthält.

Analog zu dem im vorletzten Abschnitt beschriebenen Knoten, wird ebenfalls ein Erzeugungs-Befehl `createInOutBoundary`, entsprechende Manipulatoren und ein `Attribute Editor`-Template implementiert.

3.4 Der Aufruf des Strömungslösers aus Maya

Nach der Erzeugung der Szenenbeschreibung für den Strömungslöser, die erst im nachfolgenden Kapitel beschrieben wird, soll das Simulationsprogramm aus Autodesk Maya heraus gestartet werden. Dies ist ein zweistufiger Prozess: In einem ersten Schritt startet man das Programm `navsetup`, das auf Basis der Szenenbeschreibung alle wesentlichen Datenfelder für die Simulation initialisiert und in einer Binärdatei abspeichert. Erst mit dieser Datei kann der eigentliche Löser `navcalc` arbeiten.

Der Aufruf des Programms `navsetup` verbirgt sich hinter einem Menüpunkt *Simulation* → *Set Up Simulation*. Umgesetzt wird der Aufruf über ein MEL-Skript, das erst ein geeignetes Kommandozeilen-Skript zum Start von `navsetup` erstellt. Anschließend wird ein Kommandozeilen-Fenster geöffnet und darin das neu erstellte Skript gestartet. Analog dazu erfolgt der Aufruf des Lösers über den Menüpunkt *Simulation* → *Start Simulation*. Auch hier wird ein Kommandozeilen-Fenster geöffnet, in dem ein Skript den Löser startet.

Auf den ersten Blick scheint der hier gewählte Ansatz zum Start der beiden Programme etwas umständlich. Tatsächlich aber garantiert die Vorgehensweise sowohl eine von Maya unabhängige Simulation als auch eine übersichtliche Darstellung der Ausgabe der beiden Programme: Es wird für die Kommandozeilen-Fenster jeweils ein eigener Prozess gestartet. Damit kann der Strömungssimulator auch dann weiterlaufen, wenn Maya beendet wird. Vor dem Hintergrund von sehr hoch aufgelösten und damit langwierigen Berechnungen ist dies ein entscheidender Vorteil. Gleichzeitig sichert das geöffnete Ausgabefenster stets eine einfache Kontrolle des aktuellen Zustands der Simulation zu.

Zusammen genommen ermöglichen die in diesem Kapitel vorgestellten Knoten und Befehle die vollständige Konfiguration und den Start einer Strömungssimulation. Es ist möglich, alle wesentlichen allgemeinen Simulationsparameter sowie die Anfangs- und Randbedingungen des Strömungssimulators NaSt3DGPF vorzugeben. Alle im Entwurfs-Abschnitt geforderten Eigenschaften und Funktionalitäten werden erfüllt. Insbesondere wird die Komplexität des Simulationsprogramms vor dem Nutzer verborgen. Das Ergebnis ist eine sehr einfache Benutzerschnittstelle für die Verwaltung der Simulation und der damit verbundenen Geometrien.

4 Erstellung einer Szenenbeschreibung für den Strömungslöser

Die Hauptarbeit bei der Konfiguration der Strömungssimulation ist *für den Anwender* von Maya das Zusammenstellen der Konfigurationsparameter und Geometrien. Sind diese erst einmal gefunden, wird die wesentliche Aufgabe der Erstellung einer geeigneten Szenenbeschreibung für den Simulator NaSt3DGPF vollständig durch einen einfachen Befehlsaufruf gekapselt. Die Szenenbeschreibung besteht zum einen aus einer Konfigurationsdatei für allgemeine Simulationsparameter und zum anderen aus Dateien zur diskreten Darstellung der Geometrien (siehe Abbildung 4.1). In diesem Kapitel wird die Erzeugung dieser Beschreibung detailliert vorgestellt.

Das Kapitel teilt sich analog zum vorangegangenen Kapitel auf. Zunächst wird die Erzeugung der Konfigurationsdatei aus dem Knoten zur Beschreibung der allgemeinen Grundparameter einer Simulation beschrieben. Im Anschluss wird überblicksmäßig auf die Ausgabe der Daten der Knoten für Anfangs- und Randbedingungen eingegangen. Bei diesen müssen nicht nur Parameter in die Konfigurationsdatei gespeichert, sondern auch die Geometrien in eine diskrete Form überführt werden. Dazu wird die Verwendung von *Voxelisierungen* motiviert und ein geeigneter Algorithmus zu deren Umsetzung vorgestellt. Mit diesem Hintergrundwissen kann im darauf folgenden Abschnitt die Umformung der Anfangs- und Randbedingungen vollständig vorgestellt werden.

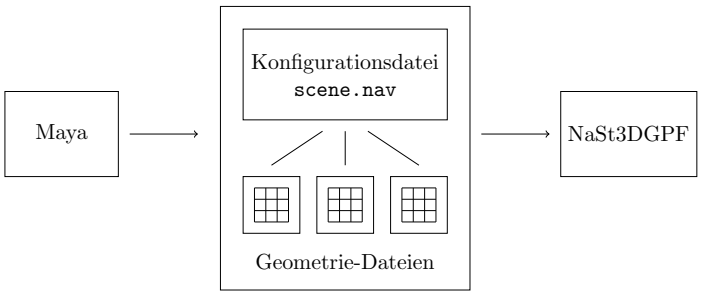


Abb. 4.1: Eine Szenenbeschreibung für den Strömungslöser besteht aus einer Konfigurationsdatei und weiteren Dateien mit voxelisierten Geometrien.

4.1 Abspeicherung der allgemeinen Simulationsparameter

Im Abschnitt 3.1.1 wird gefordert, dass eine geeignete Szenenbeschreibung für den Strömungssimulator NaSt3DGPF nur durch den Aufruf eines Kommandos realisiert wird. Dieses Kommando hat in der Implementierung den Namen `createNavFile`. Es bekommt als Parameter

bzw. Selektion die Simulationsumgebung übergeben und generiert vollautomatisch alle für den Start des Strömungssimulators benötigten ASCII-Dateien.

Die Implementierung von `createNavFile` beruht wie gewohnt auf der Ableitung der Klasse `MPxCommand` und dem Überschreiben der darin enthaltenen Methode `doIt`. In dieser wird zunächst die übergebene Simulationsumgebung bestimmt. Außerdem wird der Name und Pfad für die Dateien zur Szenenbeschreibung für den Strömungssimulator zusammengesetzt: Generell erhält eine durch das Maya-Plugin erzeugte Konfigurationsdatei für den Strömungslöser den Dateinamen `scene.nav`. Diese Konvention erleichtert die Verwaltung der Szenen-Dateien. Als Pfad für Dateien zur Szenenbeschreibung wird das Verzeichnis aus dem `projectDirectory`-Attribut der `SimulationArea` genommen.

Sämtliche Routinen zum Schreiben der Simulationsparameter werden aus der Methode[1] `createNavFile` heraus aufgerufen. Diese wird in `doIt` gestartet.

Als Erstes öffnet die Methode `createNavFile` die Datei `scene.nav` im entsprechenden Pfad. Dann übernimmt die zu implementierende Funktion `writeDimensions` das Auslesen und Schreiben der Informationen, die im Block `dimension` (siehe Abschnitt 2.2.4) der Konfigurationsdatei stehen. Dies sind die physikalische Größe des Berechnungsgebietes und die Auflösung.

Jene allgemeinen Simulationsparameter, die im Abschnitt `parameter` von `scene.nav` aufgeführt werden sollen, werden über die eigens erstellte Methode `writeParameters` aus dem `SimulationArea`-Knoten ausgelesen und im nötigen Format abgespeichert. Dabei ist die Implementierung so aufgebaut, dass die erzeugte Datei möglichst übersichtlich bleibt: Alle Parameter, die gerade nicht benötigt werden, werden auch nicht abgespeichert. Zudem werden die Verzeichnisse, die für Daten-Ausgaben bei der Simulation gewählt wurden, auf ihre Existenz und Schreibrechte hin überprüft. Das geschieht durch das testweise Erstellen einer temporären Datei. Lässt sich diese nicht zum Schreiben öffnen, bricht das Kommando mit einer Fehlermeldung ab. Ohne diese Fehlerabfrage könnte die Simulation ungewollt scheitern.

Mit der Methode `writeDefaultFluidType` wird als Nächstes die Fluidphase festgelegt, mit der die Simulationsumgebung grundlegend gefüllt wird. Dazu wird das entsprechende Attribut der `SimulationArea` abgefragt. In der Konfigurationsdatei wird das *Füllen* der Umgebung durch die Erzeugung eines Fluid-Quaders der Größe des Simulationsbereichs erreicht. Soll zum Beispiel eine Berechnungsregion mit einer Auflösung von $80 \times 80 \times 80$ mit der Gasphase gefüllt werden, wird der folgende Eintrag in die Datei `scene.nav` geschrieben:

```
box {
     coords<1, 1, 1>,<80, 80, 80>
     fluid
     gas
}
```

Durch den `SimulationArea`-Knoten lässt sich für einzelne *Wände* des Simulationsgebiets vorgeben, ob diese Rutsch- oder Haft-Randbedingungen besitzen. Die Attribute hierzu tragen im Namen den Präfix `wallBoundaryCondition`. Die Übertragung dieser Parameter in das Format vom Strömungslöser erfolgt über die Funktion `writeWallBoundaryConditions`. Sie überprüft,

[1] Es handelt sich hierbei um eine Methode der von `MPxCommand` abgeleiteten Klasse und nicht etwa um das gleichnamige Kommando, das implementiert wird.

ob bereits für eine der Wände, für die eine Randbedingung vorgegeben wird, periodische Randbedingungen festgelegt wurden. Ist dies der Fall, wird die Kollision[2] durch das Ignorieren der Haft- oder Rutsch-Bedingung aufgelöst. Nach der Kontrolle werden die Randbedingungen in die Einstellungen für den Strömungssimulator übernommen. Für eine Rutschbedingung, die über das Attribut `wallBoundaryConditionXZ1` gesetzt wird, wird beispielsweise die nachfolgende Option abgespeichert:

```
box {
     east
     slip
}
```

4.2 Übersicht über die Ausgabe der Anfangs- und Randbedingungen

Mit dem Abspeichern der Randbedingungen für die *Außenwände* der Simulationsumgebung ist die Konfiguration der allgemeineren Simulationsparameter abgeschlossen. Anschließend erzeugt die Methode `createNavFile` aus den Knoten für die Rand- und Anfangsbedingungen die vom Simulationsprogramm benötigten Daten bestehend aus weiteren Einträgen in der Konfigurationsdatei `scene.nav` und zusätzlichen Dateien mit diskretisierten Geometrieinformationen. Dazu bedient sich die Implementierung des Attributs `shapes` des `SimulationArea`-Knotens. Alle Knoten für Anfangs- und Randbedingungen sind mit diesem Attribut verbunden und lassen sich darüber abfragen.

Für jeden Knotentyp zur Verwaltung der Anfangs- und Randbedingungen wird eine eigene Methode zum Abspeichern der Daten implementiert. Diese trägt den Namen des jeweiligen Knotentyps und als Präfix das Wort `write`. Auf die genaue Funktionsweise der Methoden soll erst im Abschnitt 4.4 eingegangen werden, da zunächst einige Grundlagen zur Umwandlung von polygonalen in voxelbasierte Geometrien erklärt werden müssen. Die Reihenfolge der Parameterausgabe für die verschiedenen Anfangs- und Randbedingungstypen ist wichtig. So wird in Abschnitt 4.4.2 die Notwendigkeit der Ausgabe der Hindernisgeometrien nach der Ausgabe der Fluid-Bereiche erörtert. Ebenso ist es sinnvoll, Ein- und Ausströmungsbereiche nach den Fluid- und Hindernisgeometrien auszugeben, da andernfalls ungewollte Überschneidungen entstehen könnten. Demzufolge werden in der Methode `createNavFile` zunächst die an das Attribut `shapes` angebundenen Knoten bestimmt und anschließend gebündelt für jeden Typ entsprechende Ausgabemethoden in der richtigen Reihenfolge aufgerufen (siehe Algorithmus 4.2.1 auf Seite 56).

Beendet wird die Funktion `createNavFile` mit dem Schließen der Konfigurationsdatei `scene.nav`. Damit ist auch das *Kommando* `createNavFile` fertig und eine geeignete Szenenbeschreibung für den Strömungssimulator NaSt3DGPF erzeugt. Zur besseren Übersicht wird im Algorithmus 4.2.1 auf der grobe Funktionsumfang der `createNavFile`-Methode umrissen.

[2] Periodische Randbedingungen und Rutsch-/Haft-Bedingungen auf einer Wand schließen sich gegenseitig aus.

Algorithmus 4.2.1: CREATENAVFILE($SimulationArea, configFile$)

$open(configFile)$

$writeDimensions(SimulationArea, configFile)$
$writeParameters(SimulationArea, configFile)$
$writeDefaultFluidType(SimulationArea, configFile)$
$writeWallBoundaryConditions(SimulationArea, configFile)$

$shapeNodes = getNodesConnectedTo(SimulationArea.shapes)$

for each $node \in shapeNodes.getNodesByType(SimulationFluidShape)$
do $writeSimulationFluidShape(node, SimulationArea, configFile)$

for each $node \in shapeNodes.getNodesByType(SimulationSolidShape)$
do $writeSimulationSolidShape(node, SimulationArea, configFile)$

for each $node \in shapeNodes.getNodesByType(SimulationInflowBoundary)$
do $writeSimulationInflowBoundary(node, SimulationArea, configFile)$

for each $node \in shapeNodes.getNodesByType(SimulationCustomInflowBoundary)$
do $writeSimulationCustomInflowBoundary(node, SimulationArea, configFile)$

for each $node \in shapeNodes.getNodesByType(SimulationInOutBoundary)$
do $writeSimulationInOutBoundary(node, SimulationArea, configFile)$

$close(configFile)$

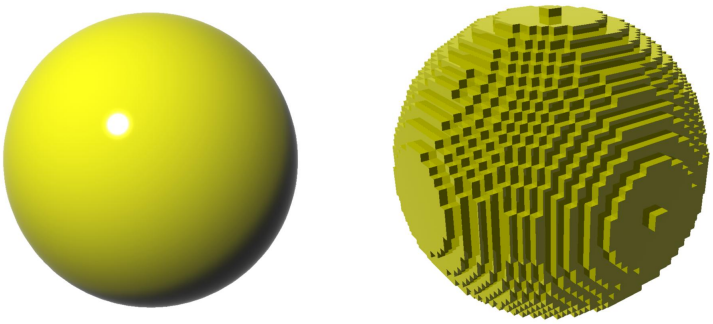


Abb. 4.2: Eine Kugel in polygonaler und voxelisierter Darstellung.

4.3 Geometrie-Umwandlung mittels Voxelisierung

4.3.1 Grundlagen

In der Computer Graphik wird ein einzelnes Volumenelement, in Anlehnung an Pixel, als *Voxel* bezeichnet. Davon leitet sich der Begriff der *Voxelisierung* ab. Diese Technik kommt aus dem Bereich der volumetrischen Graphik und bezeichnet die Umwandlung einer beliebigen Geometrie-Repräsentation in eine Blöckchen- bzw. Voxel-basierte Darstellung. Sie dient als Vorstufe für bestimmte volumetrische Renderingverfahren [10]. Abbildung 4.2 zeigt eine Kugel und ihre voxelisierte Repräsentation.

Anstatt für ein Rendering, soll das Verfahren in dieser Arbeit zur Erzeugung der zellbasierten beziehungsweise diskreten Geometriedarstellung[3] des Strömungslösers verwendet werden. Die Grundidee der Voxelisierung ist sehr einfach: Zunächst überzieht man das Gebiet mit den umzuwandelnden geometrischen Objekten mit einem Gitter einer entsprechenden Maschenweite. Für jede Zelle in dem Gitter wird dann entschieden, ob sie sich innerhalb eines der Objekte befindet. Davon abhängig wird ein boolescher Wert gespeichert. Ein Feld bestehend aus diesen booleschen Werten beschreibt schließlich die geometrischen Objekte.

Das Voxelisieren beliebiger Geometrien ist ein in der Literatur wohl bekanntes und viel diskutiertes Problem. Frühe Verfahren [49] nutzen direkte Abfragen nach der *innerhalb-/außerhalb-Eigenschaft* von Punkten, um eine voxelisierte Geometrierepräsentation zu finden. Da dies hohe Laufzeiten nach sich zieht, wurden in der Folge sogenannte *scan conversion*-Algorithmen entwickelt, die speziell auf einzelne geometrische Primitive (Linien, Dreiecke,...) zugeschnitten sind und diese effizient umwandeln. Das Paper von Kaufman und Shimony [46] beschreibt einige klassische Vertreter dieser Gattung. Gleichzeitig wurden auch Verfahren weiterentwickelt, die ganze Szenen voxelisieren. Nicht zuletzt durch moderne Graphikhardware sind in den letzten Jahren sehr schnelle Umsetzungen der Voxelisierungsidee entstanden. In Publikationen von Eisemann und Décoret [19], [20] werden Verfahren vorgestellt, die die Renderingpipeline von Grafikkarten benutzen, um ganze 3D-Szenen auf einmal umzuwandeln.

[3] Die Details zur Festlegung von Geometrien im Strömungslöser NaSt3DGPF finden sich im Abschnitt 2.2.3.

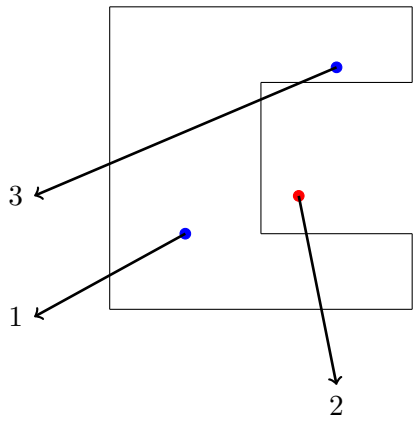


Abb. 4.3: Charakterisierung der *innerhalb-/außerhalb-Eigenschaft* von Punkten mit Hilfe von Strahlenschnitten.

In dieser Arbeit sollen direkt mit den Mitteln von Autodesk Maya und auf algorithmisch einfache Weise Voxelisierungen von Geometrien in Maya-Szenen erzeugt werden. Das nachfolgend beschriebene Verfahren erfüllt diese Anforderungen, besitzt aber im Gegensatz zu den zuvor zitierten Methoden keine optimale Laufzeitkomplexität. Dennoch wird mit dieser Herangehensweise gearbeitet, da einerseits die Laufzeiten für praxisrelevante Probleme akzeptabel[4] sind und andererseits bewusst nur mit den Mitteln des Animationsprogramms gearbeitet werden soll.

4.3.2 Voxelisierung mittels Strahlenschnitten

Der hier verfolgte Ansatz zum Voxelisieren beruht auf der Berechnung von Strahlenschnitten. Angenommen, in einem Raum befinden sich nur geschlossene Polyeder. Dann lässt sich für jeden Punkt in diesem Raum feststellen, ob er innerhalb oder außerhalb eines dieser Objekte ist. Hierzu wird ein Strahl von einem Punkt in eine beliebige Richtung betrachtet. Die Anzahl an Schnitten mit Flächen der Polyeder gibt nun Auskunft über die Position des Punktes. Ist die Anzahl gerade, so ist der Punkt außerhalb der Polyeder, andernfalls ist der Punkt innerhalb mindestens eines Polyeders. Diese Aussage beruht auf einer dreidimensionalen Erweiterung des Jordan-Theorems für Kurven in der Ebene [34]. Abbildung 4.3 illustriert das Vorgehen.

In Algorithmus 4.3.1 auf Seite 59 wird nun gerade die Idee der Strahlenschnitte genutzt um für jedes Voxel eines Gitters `voxelField` zu entscheiden, ob dieses innerhalb mindestens eines Polyeders aus der Liste `polyhedronList` liegt. Damit entsteht ein Voxelisierungsverfahren.

Die Wahl des Voxel-Mittelpunktes als Startposition ist dabei reine Willkür. Tatsächlich entscheidet aber der Ursprung des Strahls über die resultierende Voxelisierung. So bestimmt das Strahlschnitt-Verfahren nur für diesen Punkt die Lage. Demzufolge ist es möglich, dass Algorithmus 4.3.1 ein Voxel als *außerhalb* der Polyeder charakterisiert, tatsächlich aber ein Teil des Voxels ein Objekt schneidet und nur der Mittelpunkt nicht innerhalb des Objektes ist. Zur Stabilisierung des Verfahrens werden von jeder Ecke eines Voxels Strahlenschnitt-Tests durch-

[4] Im Ergebniskapitel finden sich in Abschnitt 8.2 Performance-Messungen zu den in dieser Arbeit implementierten Voxelisierungsverfahren.

Algorithmus 4.3.1: VOXELIZE($polyhedronList, voxelField$)

for each $voxel \in voxelField$

do
$ray.startPosition = voxel.centerPosition$
$ray.direction = getRandomDirection()$
$intersectionList = ray.getIntersections(polyhedronList)$
if $odd(intersectionList.size)$
  then $voxel.insidePolyhedron = true$
  else $voxel.insidePolyhedron = false$

geführt. Tritt bei einem der Strahlen eine ungerade Anzahl von Schnitten auf, wird das Voxel als *innerhalb* klassifiziert.

Ein Nachteil des Verfahrens besteht in der anfangs gestellten Forderung nach geschlossenen Polyedern. Offensichtlich kann ein Strahl in eine *beliebige* Richtung bei einem offenen Polyeder gerade das Loch in der Geometrie treffen, wodurch die Charakterisierung fehlschlägt. Auch moderne Verfahren, wie das von Eisemann und Décoret vorgestellte GPU-basierte Verfahren zur Voxelisierung von massiven Körpern [20] können dem Problem nicht ausweichen. In dem Paper der beiden Autoren wird als Heuristik vorgeschlagen, bei offenen Polyedern nur die Randflächen zu voxelisieren und dann die Randdarstellung mit den voxelisierten geschlossenen Polyedern zu kombinieren. Aus numerischen Gründen[5] ist aber ein solches Vorgehen abzulehnen. Für eine Strömungssimulation ist folglich nur der Einsatz von geschlossenen Polyedern als geometrische Objekte sinnvoll.

4.3.3 Implementierung der Voxelisierung

Praktisch umgesetzt wird das Voxelisierungsverfahren über die Methode `voxelize` der Klasse `SimulationVoxelizer` im `SimulationArea`-Plugin. Diese neu implementierte Klasse ist eine Sammlung von verschiedenen Voxelisierungsmethoden, die alle im Rahmen der Konfiguration von Fluidsimulationen in Maya benötigt werden. Als Parameter bekommt die Methode `voxelize` ein polygonales Mesh, den Namen der Datei, in die die Voxelisierung gespeichert werden soll und Informationen über das Voxelisierungsgitter. Die Informationen bestehen aus der Position im Raum, der Größe des Gitters und der Anzahl an Zellen pro Raumrichtung.

Bei der Implementierung der Voxelisierung wird ein von Maya angebotener Mechanismus zur Berechnung von Strahl-Mesh-Schnitten verwendet. Wie sich herausstellte ist dieser nicht besonders robust implementiert. So werden Schnitte von Strahlen mit Kanten oder einzelnen außen liegenden Vertices der Meshes nur zum Teil korrekt erkannt. Strahlen, die in einer Außenfläche des Meshes verlaufen, machen ebenso Probleme. Auch ein Toleranz-Wert, der die Genauigkeit des Schnitt-Algorithmus steuert, liefert hier keine Abhilfe. Um den Problemen zu begegnen, werden pro Punkt vier Strahlen in beliebige Richtungen ausgesandt. Ein Punkt

[5] Der Strömungssimulator fordert eine Grunddicke von mehr als einem Voxel für Geometrien. Andernfalls schlagen die Diskretisierungen der Ableitungen fehl.

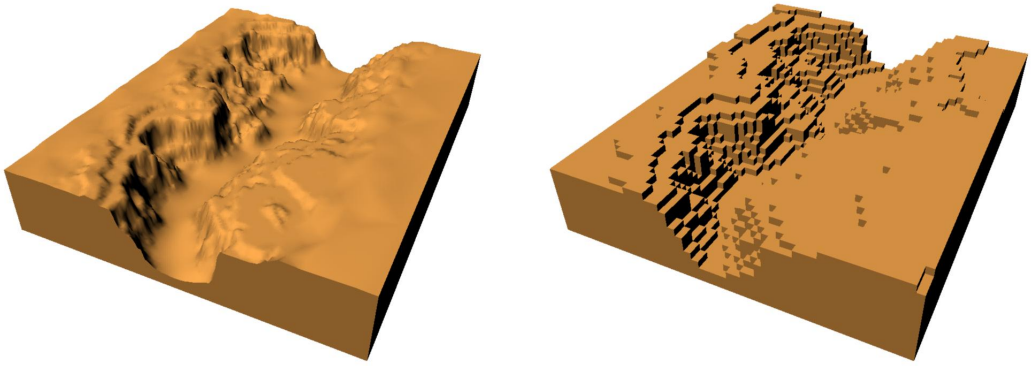


Abb. 4.4: Eine Canyon-Geometrie (links) und ihre grobe Voxelisierung (rechts) bei einer Auflösung von $50 \times 16 \times 63$ Voxeln.

wird als innerhalb des polygonalen Netzes klassifiziert wenn mindestens die Hälfte der Strahlen dieses Resultat angeben.

Abgesehen von dieser Schwierigkeit arbeitet die Methode `voxelize` genau wie der eben angegebene Algorithmus: Es werden alle Voxel des Gitters durchlaufen. Pro Voxel werden an allen Ecken vier Strahlen verschickt, die detektieren, ob sich das Voxel innerhalb des Netzes befindet. Das Ergebnis dieses Tests wird in die übergebene Datei gespeichert. Diese besteht aus ASCII-Zeichen. Ein Header gibt an, welche Auflösung das voxelisierte Gebiet hat. Über die Zeichen `0` und `1` wird in einer vom Strömungslöser vorgegebenen Durchlaufreihenfolge die *innerhalb*- bzw. *außerhalb*-Charakterisierung für jedes Voxel abgespeichert.

In Abbildung 4.2 auf Seite 57 konnte schon die Visualisierung der Voxelisierung einer Sphäre betrachtet werden. Diese wurde bereits mit der Methode `voxelize` durchgeführt. Das Verfahren arbeitet aber auch sehr gut auf beliebigen geschlossenen Geometrien, wie man Abbildung 4.4 entnehmen kann. Die linke Seite der Abbildung zeigt eine Canyon-Geometrie, auf die das vorgestellte Verfahren angewendet wird. Auf der rechten Seite findet sich eine Visualisierung der voxelisierten Darstellung. Man kann erkennen, dass schon bei einer recht grob aufgelösten Voxelisierung ($50 \times 16 \times 63$ Voxel) die Details gut erfasst werden.

4.4 Ausgabe der Parameter für Anfangs- und Randbedingungen

4.4.1 Umwandlung von Fluid-Geometrien

Mit dem `SimulationFluidShape`-Knoten kann einem geschlossenen Mesh die Fluid-Eigenschaft zugewiesen werden. Dies wurde schon in Abschnitt 3.3.2 beschrieben. Auf die Erläuterung der Methode `writeSimulationFluidShape` zum Herausschreiben dieser Fluid-Geometrie[6] wurde aber bisher verzichtet, da noch kein Verfahren zur Voxelisierung zur Verfügung stand. Dies wird nun nachgeholt.

[6] Für Details zur Vorgabe von Fluid-Geometrien sei an dieser Stelle nochmals auf Abschnitt 2.2.4 verwiesen.

Die Methode `writeSimulationFluidShape` wird von `createNavFile` für jeden `SimulationFluidShape`-Knoten aufgerufen. Sie bestimmt das mit dem Knoten verbundene Mesh und lädt die Größe, Position und Auflösung der Simulation aus der `SimulationArea`. Da das Mesh bezüglich der gesamten Simulationsumgebung voxelisiert werden soll, geben diese letzten drei Parameter die Einstellungen des Voxelisierungsgitters vor. Mit diesen Informationen wird der Voxelisierer in der Form der Methode `voxelize` aus dem letzten Abschnitt gestartet. Er schreibt seine Ausgabe in eine Datei im aktuellen Projektverzeichnis. Der Dateiname entspricht dem Namen des Meshes mit der Endung `.voxel`.

Zusätzlich liest die Methode die Parameter des `SimulationFluidShape`-Knotens, wie Druck, Geschwindigkeit und Fluidtyp aus und schreibt alle gesammelten Werte in die Konfigurationsdatei für den Strömungslöser. Sollen beispielsweise die Daten für einen `SimulationFluidShape`-Knoten mit einem Mesh mit dem Namen `pCubeShape1` herausgeschrieben werden, so könnte ein neuer Eintrag in der Konfigurationsdatei wie folgt aussehen:

```
Voxel
{
   VoxelOffset <1,1,1>
   VoxelFile pCubeShape1.voxel
   fluid
   liquid
   init <1.0, 1.0, 1.0>,0
}
```

Bei dieser Konfiguration ist der erzeugte Fluid-Bereich eine Flüssigkeit mit einem initialen Geschwindigkeitsvektor $(1.0, 1.0, 1.0)^t\, m/s$ und einem Anfangsdruck von $0\, Pa$.

4.4.2 Umwandlung von Hindernis-Geometrien

Im Wesentlichen arbeitet die Methode `writeSimulationSolidShape` zur Ausgabe von Hindernis-Geometrien ähnlich zur gerade vorgestellten Methode zum Abspeichern der Parameter für Fluid-Geometrien. Durch einen Aspekt der Numerik des Strömungslösers muss aber für vorgegebene Hindernisse in Strömungen das Ergebnis der Voxelisierung abgeändert werden: In Abschnitt 2.2.3 wurde bereits darauf hingewiesen, dass Solid-Bereiche bei der Simulation durch die Level-Set–Funktion entweder als *flüssig* oder als *gasförmig* definiert werden. In der Praxis hat sich gezeigt, dass es sinnvoll ist, bei einer Simulation zunächst für das gesamte Berechnungsgebiet die flüssig-/gasförmig-Eigenschaft zu vergeben. Dann erst wird festgelegt, welche Bereiche ein Fluid enthalten und welche Hindernisse sind. Mit dieser Vorgehensweise kann ein und dasselbe Objekt teilweise als gasförmig und teilweise als flüssig charakterisiert werden.

Abbildung 4.5 auf Seite 62 veranschaulicht noch einmal die Problematik. Es soll in einem mit Flüssigkeit gefüllten Becken ein quaderförmiges Hindernis stehen. Die Abbildung zeigt alle sinnvollen Vorgaben an die Level-Set–Funktion. Im linken Bild wird das Hindernis als gasförmig beschrieben, wohingegen es im mittleren Bild als flüssig definiert ist. Die rechte Graphik zeigt die hier gewünschte Aufteilung des starren Objekts in einen flüssigen und einen gasförmigen Teil.

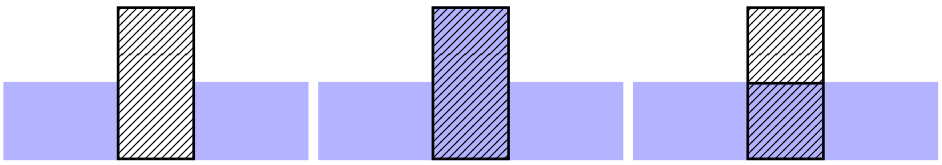


Abb. 4.5: Mögliche Verläufe der Level-Set–Funktion in einem Hindernis *(schraffiert)*: Das Hindernis kann gasförmig *(links)*, flüssig *(mitte)* oder gemischt gasförmig und flüssig *(rechts)* sein.

Die gerade beschriebene Herangehensweise wird durch die Methode `createNavFile` nachgebildet. Was bisher noch nicht erwähnt wurde ist die Tatsache, dass in der Methode ein Array verwaltet wird, das für jede Zelle des Simulationsgebiets festhält, ob sie eine Flüssigkeit oder ein Gas enthält. In der Funktion `writeDefaultFluidType` wird das Array mit einem entsprechenden Standardwert belegt. Bei jedem Aufruf von `writeSimulationFluidShape` werden die dort voxelisierten Bereiche mit dem jeweiligen Fluidtyp überschrieben. Somit ergibt sich eine Beschreibung der Fluidtypen für jede Zelle im Raum.

Mit Hilfe des Arrays wird nun in `writeSimulationSolidShape` das voxelisierte polygonale Netz aufgeteilt: Es wird für einen *flüssigen* und einen *gasförmigen* Solid-Bereich jeweils eine Voxeldatei erzeugt. Dabei kann durch das Array entschieden werden in welche der beiden Dateien ein als innerhalb des Meshes liegend markiertes Voxel abgespeichert wird. In der jeweils anderen Datei wird dann das Voxel als außerhalb markiert.

Nach dieser Aufteilung werden alle Parameter des `SimulationSolidShape`-Knotens sowohl für die flüssige als auch für die gasförmige Voxel-Geometrie abgespeichert. Damit könnte sich zum Beispiel ein Eintrag folgender Form in der Konfigurationsdatei für den Strömungslöser ergeben:

```
Voxel
{
        VoxelOffset <1,1,1>
        VoxelFile pSphereShape1.liquid.voxel
        liquid
        slip
}

Voxel
{
        VoxelOffset <1,1,1>
        VoxelFile pSphereShape1.gas.voxel
        gas
        slip
}
```

4.4.3 Erzeugung von einfachen Einströmungsbereichen

Modellierung eines zweiphasigen Einströmungsbereiches

Im Folgenden soll die Erzeugung eines rechteckigen, planaren und achsenparallelen Einströmungsbereiches im Strömungslöser NaSt3DGPF beschrieben werden. Dies erfordert zunächst einige technische Bemerkungen. In der Numerik wird der Einströmungsbereich als eine Dirichlet-Randbedingung an das Geschwindigkeitsfeld verstanden.[7] Eine solche Randbedingung wird auf den *Solidzellen* im Diskretisierungsgitter definiert, die an *Fluidzellen* angrenzen. Für Simulationen auf einem einfachen quaderförmigen Gebiet ohne Hinderniszellen bedeutet dies, dass die Randbedingungen auf einer zusätzlichen Randzellenschicht, den *Ghost-Zellen*, vorgegeben werden.

Nach dem klassischen Verständnis dringt durch das Festlegen einer Geschwindigkeit auf einem Einströmungsbereich Fluid in das Simulationsgebiet ein. Diese Anschauung ist vor dem Hintergrund von einphasigen Strömungsrechnungen sehr einfach nachzuvollziehen. Komplizierter wird die Situation aber bei zweiphasigen Simulationen. Bei diesen kann natürlich weiterhin der einfache Fall auftreten, dass im Bereich der Gas-Phase ein Gas und im Bereich der Flüssigkeits-Phase eine Flüssigkeit einströmen soll. Hier ist die Situation ganz analog zum einphasigen Fall. Soll aber ein Gas in eine Flüssigkeit oder eine Flüssigkeit in ein Gas einströmen, funktioniert die alte Anschauung nur noch bedingt:

Das Einströmen der einen Fluid-Phase in die jeweils andere erfordert nicht mehr nur das Setzen einer Geschwindigkeit am Rand. Vielmehr muss dafür Sorge getragen werden, dass ein Fluidbereich mit einer anderen Phase vom Rand her in das Berechnungsgebiet transportiert wird. Dazu muss in zwei Zellschichten des Einströmungsbereiches die Phase des einfließenden Fluids gesetzt werden: Dies ist sowohl die Schicht, auf der die Dirichlet-Randbedingung gesetzt wird, als auch die davor liegende. Insbesondere ist hier zu beachten, dass es sich bei der Zellschicht, auf der die Dirichlet-Bedingung gesetzt wird, um eine Hinderniszelle handelt. Die zweite Schicht aus Zellen ist ein Fluid.

Abbildung 4.6 auf Seite 64 veranschaulicht die Festlegung der Fluidphasen in zwei Raumdimensionen für eine einströmende Flüssigkeit in ein Gebiet mit einem Gas. Weiße Zellen stehen dabei für die Gas-Phase und blaue für die Flüssigkeitsphase. Die schraffierten Zellen sind Hinderniszellen. Pfeile markieren Zellen auf denen zusätzlich die Dirichlet-Randbedingung gesetzt ist.

Führt man mit dem gerade beschriebenen Verfahren eine Konfiguration für einen Einströmungsbereich durch, wird die Fluidoberfläche mit der gegebenen Geschwindigkeit transportiert. Es fließt also z.B. Wasser in Luft hinein. Bei der bisher verwendeten Konfiguration kann es jedoch dazu kommen, dass der Bereich im Verlauf der Simulation wegrutscht. So wird auf Grund der in einer Simulation herrschenden Gravitationskräfte die Isofläche zum Level-Set–Wert Null zum Boden hingezogen. Dadurch rutscht die obere Kante des Einfluss-Bereiches ab. Abhängig von der gewählten Auflösung und Genauigkeit der Simulation kann dieser Effekt so drastisch sein, dass nach einer gewissen Zeit kein Fluid der einströmenden Phase mehr in die Simulationsumgebung transportiert wird.

Zur Stabilisierung wird für die Fluidzellen mit der einströmenden Phase eine umrandende Schicht aus *Hinderniszellen* der selben Phase einführt. Die Hinderniszellen halten die Level-

[7] Alle wichtigen Randbedingungen wurden bereits im Grundlagenkapitel in Abschnitt 2.1.1 beschrieben.

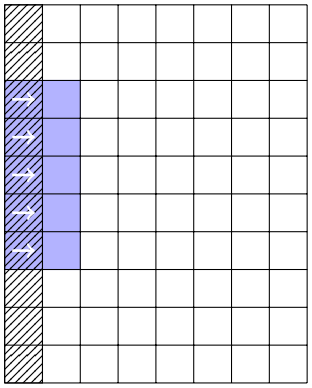


Abb. 4.6: Benötigte Festlegung der Fluidphasen bei einer einströmenden Flüssigkeit in ein Gas.

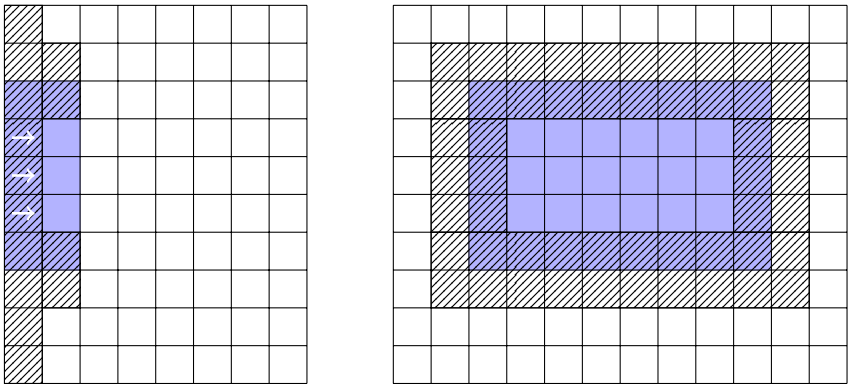


Abb. 4.7: Konfiguration eines Einströmrandes mit Stabilisierung (links: Querschnitt, rechts: frontale Ansicht).

Set–Funktion fest, weil dort die Navier-Stokes–Gleichungen nicht berechnet werden und somit auch kein Transport der freien Oberfläche auf Grund der Gravitation stattfindet.

Wegen gewisser numerischer Einschränkungen[8] darf die Umrandung aus Hinderniszellen nicht dünner als zwei Zellschichten sein. Setzt man nun zwei Schichten ein, sollte der Fluidtyp des äußeren Hindernisrandes auf die Standard-Fluidphase des Simulationsgebietes gesetzt werden.

Klarer dürfte die gerade beschriebene Vorgehensweise bei der Betrachtung von Abbildung 4.7 werden. Diese zeigt in der linken Graphik einen zweidimensionalen Querschnitt durch den Einströmungsbereich einer Flüssigkeit in ein Gas. Rechts wird die frontale Ansicht der Einströmregion skizziert. Wie in Abbildung 4.6 sind die schraffierten Zellen Hinderniszellen. Mit der dargestellten Konfiguration wird der Einströmvorgang stets korrekt verlaufen.

[8] Diese Einschränkungen ergeben sich aus der Forderung nach der Wohldefiniertheit von bestimmten Randbedingungen.

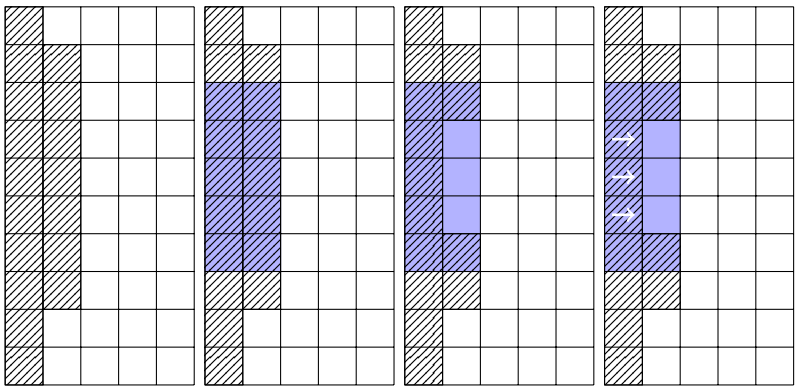


Abb. 4.8: Darstellung der schrittweisen Festlegung von Zelleigenschaften durch die Methode `writeSimulationInflowBoundary`.

Implementierung

Für die Festlegung des Einströmungsbereiches in der Konfigurationsdatei des Strömungslösers wird die Methode `writeSimulationInflowBoundary` implementiert. Diese wird, wie in Abschnitt 4.2 erwähnt, von der Methode `createNavFile` für jeden `SimulationInflowBoundary`-Knoten aufgerufen. Sie lädt alle Parameter des jeweiligen Knotens. Darauf basierend wird festgestellt, wie viele Zellen im Diskretisierungsgitter durch den Einströmungsbereich eingenommen werden. Außerdem wird die zellgenaue Position des Einströmungsbereiches bestimmt.

Eine Erzeugung der Belegung der Zellen analog zu Abbildung 4.7 geschieht nun *schichtweise*: Es ist bei der Konfiguration des Strömungslösers möglich, für einzelne Zellen mehrfach Eigenschaften zu setzen. Der jeweils letzte Eintrag in der Konfiguration für eine Zelle wird bei der Rechnung verwendet. Abbildung 4.8 illustriert die Vorgehensweise des Algorithmus, die verschiedenen Eigenschaften der Zellen im Einströmbereich durch schrittweises Überschreiben festzulegen.

Die Vergabe von Eigenschaften für die Zellen geschieht über Voxelisierungen. Für die Festlegung der Einfügepositionen und für die Bestimmung der Zellschichten wird sowohl die Orientierung als auch die Hauptrichtung des `SimulationInflowBoundary`-Knotens verwendet. Letztere legt fest in welcher Reihenfolge die Fluid-Schicht und die Hinderniszellenschicht mit den Dirichlet-Randwerten gesetzt werden. Soll beispielsweise bei dem in Abbildung 4.7 gezeigten Einströmbereich das Fluid in der linken Graphik nach links fließen, müssen die beiden Schichten vertauscht werden. Als Ausgabe in die Konfigurationsdatei erzeugt die Methode `writeSimulationInflowBoundary` zum Beispiel nachfolgende Einträge:

```
Voxel
{
        VoxelOffset <6,9,0>
        VoxelFile SimulationInflowBoundary1.inflow_outer_solid.voxel
        gas
}
```

```
Voxel
{
        VoxelOffset <7,10,0>
        VoxelFile SimulationInflowBoundary1.inflow_inner_solid.voxel
        liquid
}

Voxel
{
        VoxelOffset <8,11,1>
        VoxelFile SimulationInflowBoundary1.inflow_fluid.voxel
        fluid
        liquid
}

Voxel
{
        VoxelOffset <8,11,0>
        VoxelFile SimulationInflowBoundary1.inflow.voxel
        inflow <0.000000,0.000000,2.000000>
        liquid
}
```

In diesem Fall strömt eine Flüssigkeit in ein mit Gas gefülltes Simulationsgebiet mit einem Geschwindigkeitsvektor von $(0, 0, 2)^t\, m/s$ ein. Die *obere linke* Ecke des Einströmbereiches wird an die Voxelposition $< 7, 10, 0 >$ gesetzt. Zu den verschiedenen voxelisierten Quadern für äußere und innere Hinderniszellen-Umrandungen und für die Dirichlet-Randwerte sowie die Fluid-Zellschicht werden entsprechend verschobene Positionen festgelegt. Deren Größe ergibt sich aus den übergebenen Voxel-Dateien.

Erweiterung auf zeitabhängige Geschwindigkeiten

Es ist möglich, die am Einströmungsrand festgelegte Geschwindigkeit über die Simulationszeit hinweg zu verändern. Realisiert wird dies über den in Autodesk Maya verfügbaren Mechanismus der *Keyframe*-basierten Animation. Bei diesem werden Attributen für einzelne *Frames*, also Zeitpunkte in der Animation, feste Werte über sogenannte *Keys* zugeordnet. Für Zeitpunkte ohne vorgegebene Belegungen, werden die Werte per Interpolation aus den benachbarten Key-Werten bestimmt. Häufig kommt dabei einfache lineare Interpolation zum Einsatz.

Wird über das Attribut `timeVelo` des `SimulationInflowBoundary`-Knotens der Mechanismus für zeitabhängige Geschwindigkeiten angeschaltet, erwartet die Methode `writeSimulationInflowBoundary`, bei dem `inputVelocity`-Attribut gesetzte Key-Werte. Eine Methode liest die Geschwindigkeit zu äquidistanten Zeitpunkten[9] aus. Die Werte werden dann in eine ASCII-Datei geschrieben, die als Header die Anzahl der Einträge und zeilenweise Einträge der Form

[9] Der Abstand dieser *Abtastungen* wird über das Attribut `timeVeloSamplesPerSecond` festgelegt.

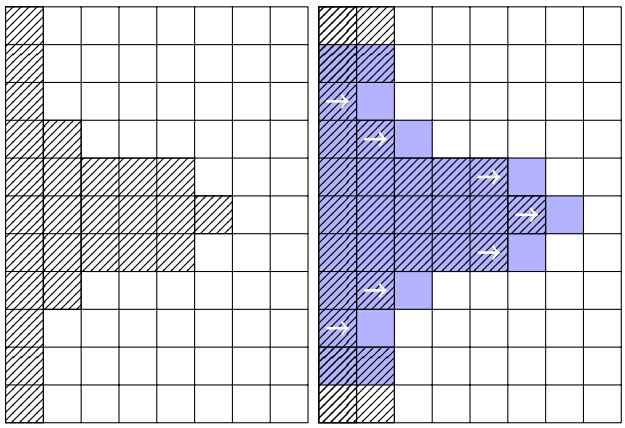


Abb. 4.9: Eine beliebig geformte Geometrie (links) mit einer möglichen Festlegung eines Einströmungsbereiches (rechts).

```
<Zeitpunkt> <X-Geschwindigkeit> <Y-Geschwindigkeit> <Z-Geschwindigkeit>
```

enthält. Eingebunden wird die Datei in die Konfiguration des Strömungslösers über den Parameter `timeveloFromFile`, der bei Bedarf zusätzlich zum Parameter `inflow` anzugeben ist.

4.4.4 Konstruktion von beliebig geformten Einströmungsbereichen

Einschränkungen an die Form

Das Erzeugen von beliebig geformten Einströmungsbereichen bei einer Simulation ist eine komplexe Aufgabe. Schon bei den einfachen rechteckigen, achsenparallelen und planaren Einflussregionen aus dem letzten Abschnitt werden zur korrekten Funktionsweise *vor* dem Bereich Fluidzellen mit der einströmenden Phase initialisiert. Außerdem stabilisieren umrandende Hinderniszellenschichten die Region. Ein beliebig geformter Einströmungsbereich muss selbstverständlich auch die Forderungen erfüllen. Dies erschwert insbesondere den Einsatz von *nicht planaren* Einflussbereichen. So kann die Umrandung nur noch bedingt die Einströmregion stabilisieren. Außerdem kann das benötigte Setzen der Fluidphase vor dem Einströmungsbereich sehr schwierig werden.

Abbildung 4.9 veranschaulicht die Problematik. Eine funktionierende Konfiguration für einen derart geformten Einströmungsbereich zeigt die rechte Graphik. Es wird in einer Raumrichtung eine Fluidzellenschicht der einströmenden Phase vor die Dirichlet-Randwerte gelegt. Unklar ist jedoch, ob diese Fluidschicht generell ausreicht, um ein korrektes Einfließen zu gewährleisten. Auch die Festigkeit der Umgrenzung des Einströmrandes ist nicht sicher. Schwieriger wird es noch bei einem Einströmungsrand, der beispielsweise auf einer Kugeloberfläche definiert wird. Bei dieser ist die Umsetzung des Konzeptes einer Umrandung nicht möglich.

Schließlich muss auch in Frage gestellt werden, ob ein nicht planarer Einflussbereich überhaupt von Nutzen ist. Es gibt kaum Anwendungen, bei denen sich ein solcher Bereich nicht auch über eine planare Fläche sinnvoll darstellen lässt.

Vor diesem Hintergrund wird im Rahmen der vorliegenden Arbeit auf die Erstellung von *nicht planaren* Einströmungsbereichen verzichtet. Sieht man davon ab, werden aber alle anderen Wünsche an eine beliebig geformte Randbedingung erfüllt: Eine entsprechende Einfließfläche kann beliebig im Raum positioniert und rotiert werden. Außerdem kann ihr Rand jede vom Benutzer gewünschte Form erhalten. Realisiert wird ein derartiger Einströmbereich durch den bereits in Abschnitt 3.3.4 vorgestellten Knoten `SimulationCustomInflowBoundary`.

Anforderungen an eine Voxelisierung

Soll ein Einflussbereich über den Knoten `SimulationCustomInflowBoundary` erzeugt werden, muss man eine Voxelisierung vornehmen. Dabei gibt es folgende Aufgaben: Das mit dem Knoten verbundene Mesh muss entlang der durch den Knoten vorgegebenen Fläche voxelisiert werden. Vor der so entstandenen Fläche sind Fluidzellen mit der jeweiligen Fluidphase einzufügen. Zudem muss, wie auch schon bei den einfachen Einflussrändern, eine doppelte Umrandungsschicht aus Hinderniszellen um die voxelisierte Fläche erzeugt werden.

Für eine korrekte Voxelisierung und Konfiguration des Einströmungsbereiches muss für die planare Einströmungsfläche eine *Hauptrichtung* $h_{\vec{n}}$ bestimmt werden. Diese entspricht der Koordinatenrichtung, in deren Komponente die Normale der Fläche ihren betragsmäßig größten Wert hat,

$$h_{\vec{n}} = \underset{i \in \{1,2,3\}}{\arg\max} \left| \vec{n}_i \right| . \tag{4.1}$$

Die Hauptrichtung gibt eine Art Orientierung des zu erstellenden beliebig geformten Einflussbereiches vor. In der Hauptrichtung darf die Voxelisierung des Meshes entlang der Fläche des Einströmbereiches nur die Dicke eines Voxels haben. Damit wird die Topologie der Konfiguration von einfachen Einströmrändern nachgeahmt.

In Abbildung 4.10 auf Seite 69 kann man sehen, wie unter Beachtung der Hauptrichtung eine Konfiguration für einen beliebig geformten Einfließbereich gefunden wird. Die Darstellung zeigt links den Querschnitt eines würfelförmigen Objektes, durch das die hier rot markierte Schnittfläche des `SimulationCustomInflowBoundary` gelegt wird. Eine die Hauptrichtung respektierende Voxelisierung und Konfiguration zeigt die rechte Graphik.

In der Voxelisierung wird für die Hinderniszellen mit Dirichlet-Randbedingung, wie gefordert, genau eine Zellschicht in der Hauptrichtung verwendet. Dirichlet-Randwerte auf Zellen, die in der Hauptrichtung hintereinander liegen, hätten keinen numerischen Effekt. Entsprechend werden sie ausgelassen.

Für die Fluidzellen mit der Phase des einströmenden Fluids werden zwei Zellschichten gesetzt. Eine solche Festlegung sichert zu, dass selbst bei schräg zu den Koordinatenachsen verlaufenden Einströmflächen genügend Fluid-Zellen mit der Phase der einströmenden Flüssigkeit beziehungsweise des einströmenden Gases zur Verfügung stehen.

Die doppelte Zellschicht für das einströmende Fluid wird aus der Voxelisierung der Hinderniszellen mit Dirichlet-Randbedingung mittels Verschiebung in die Richtung $h_{\vec{n}}$ gewonnen. Hier verhindert die Forderung nach einer Voxelisierung der Dicke von einer Zelle, dass sich die einzelnen Schichten überschneiden. Die stabilisierenden Hinderniszellen um den Einfließbereich liegen ebenfalls genau in der Hauptrichtung $h_{\vec{n}}$. Sie halten die beiden Fluidschichten fest.

Analog zu dem gerade beschriebenen Beispiel in Abbildung 4.10 soll stets vorgegangen werden. Wie das Beispiel zeigt, muss bei der Entwicklung eines Algorithmus zur Konfiguration

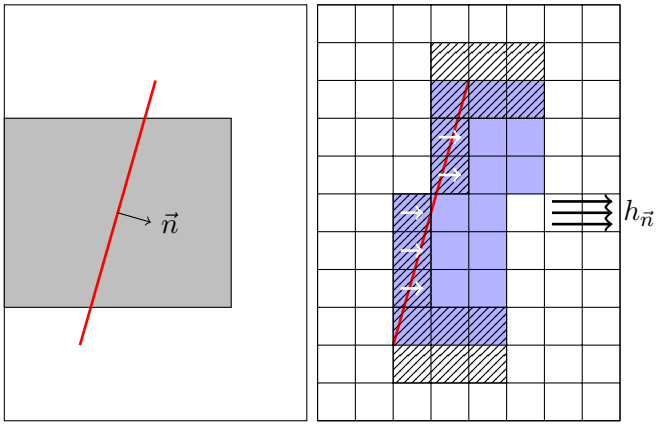


Abb. 4.10: **Links:** Die Fläche des `SimulationCustomInflowBoundary` markiert einen Einflussbereich durch den Schnitt mit einem Quader. **Rechts:** Eine resultierende Konfiguration für den Einströmungsbereich.

von beliebigen Einflussbereichen stets auf die korrekte Behandlung der Richtung $h_{\vec{n}}$ geachtet werden.

Algorithmus zur Konfiguration

Im Folgenden soll ein Algorithmus vorgestellt werden, der die Schnittfläche des `SimulationCustomInflowBoundary`-Knotens mit dem dazugehörigen Mesh in Abhängigkeit von der Hauptrichtung voxelisiert. Hierauf aufbauend werden die verschiedenen Zellschichten des Einströmungsbereiches erzeugt und mit geeigneten Hinderniszellen umrandet.

Die erste Hauptaufgabe des Algorithmus besteht in der Voxelisierung einer beliebig orientierten planaren Fläche. Eine solche Aufgabe ist ein in der Literatur wohl bekanntes Problem. So findet sich beispielsweise in [46] ein *Scan Conversion*-Algorithmus zum Erzeugen einer Voxelisierung für beliebige Polygone.

Für den speziellen Fall der Umformung einer rechteckigen beliebig orientierten Fläche unter Beachtung der Hauptrichtung soll hier ähnlich vorgegangen werden. Aus Gründen der Einfachheit der Implementierung wird aber im Wesentlichen auf inkrementelle Berechnungen verzichtet und die sichere Einhaltung der Hauptrichtungs-Forderung in den Vordergrund gestellt. Für die angestrebte Anwendung ist der hier vorgestellte Algorithmus ausreichend schnell:

Sei zunächst eine rechteckige Fläche F im Raum durch einen Aufpunkt $\vec{p} = (p_1, p_2, p_3)^t$ und zwei aufspannende Vektoren $\vec{a} = (a_1, a_2, a_3)^t$ und $\vec{b} = (b_1, b_2, b_3)^t$ gegeben. Die Normale $\vec{n} = (n_1, n_2, n_3)^t$ der Fläche, berechnet sich wie üblich durch

$$\vec{n} = \vec{a} \times \vec{b}. \tag{4.2}$$

Projiziert man die Fläche entlang der Hauptrichtung $h_{\vec{n}}$ erhält man eine zweidimensionale Darstellung. Die beiden Achsen der Projektionsebene mögen als U- und als V-Richtung bezeichnet werden. Die Algorithmus-Idee besteht nun darin, ein Voxelgitter in der U-V-Ebene zu durch-

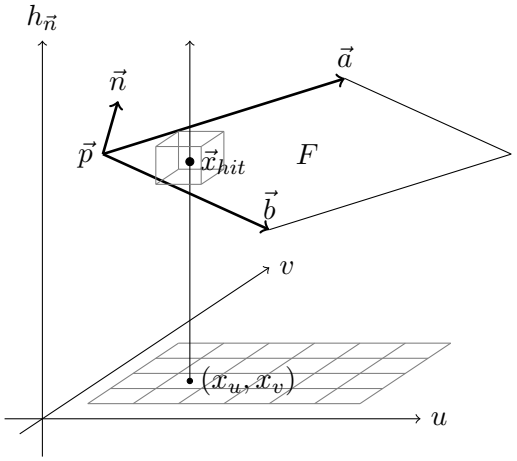


Abb. 4.11: Illustration des Voxelisierungsverfahrens für die Fläche F.

laufen und Zelle für Zelle vom Mittelpunkt Strahlen parallel zur Hauptrichtung auszusenden. Deren Schnitt mit der Fläche F ergibt, nach Rundung, die Voxelisierung.

Im Detail muss zunächst eine Bounding-Box für die Fläche F in der projizierten Ebene bestimmt werden. Sie schränkt die Anzahl der Strahlenschnitte ein. Die Positionen (x_u, x_v) der Mittelpunkte der Voxel in der U-V-Ebene ergeben sich einfach aus der Auflösung des Voxelgitters. Für die Schnittpunktberechnung eines Strahles von (x_u, x_v) entlang der Hauptrichtung wird der gesuchte Punkt $\vec{x}_{hit} = (x_u, x_v, x_{h_{\vec{n}}})^t$ einfach in die Standard-Ebenengleichung

$$(\vec{x} - \vec{p}) \cdot \vec{n} = 0 \tag{4.3}$$

eingesetzt. Nach Umformen ergibt sich $x_{h_{\vec{n}}}$ mit

$$x_{h_{\vec{n}}} = \frac{-(x_u - p_u)n_u - (x_v - p_v)n_v}{n_{h_{\vec{n}}}} + p_{h_{\vec{n}}} \tag{4.4}$$

Rundet man nun $\vec{x}_{hit}$ entsprechend auf den nächsten Mittelpunkt im Voxelgitter, ergibt sich die Mittelpunkts-Position des Voxels als $\vec{x}_{voxel}$. Abbildung 4.11 illustriert das gerade beschriebene Vorgehen.

Das bisher beschriebene Verfahren voxelisiert die auf die gesamte Bounding-Box erweiterte Ebene zu F. Mit einem zusätzlichen Schritt lassen sich nun die Voxel auch auf ihr *Enthaltensein* in der Fläche überprüfen.

Dazu bestimmt man zunächst ihre Position $\vec{x}_{rel}$ relativ zum Aufpunkt $\vec{p}$ der Fläche über

$$\vec{x}_{rel} = \vec{x}_{voxel} - \vec{p}. \tag{4.5}$$

Als Nächstes führt man für $\vec{x}_{rel}$ eine Basistransformation aus der kanonischen Basis in die Basis

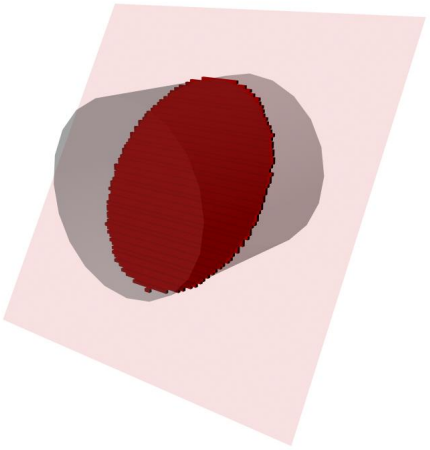


Abb. 4.12: Eine die Hauptrichtung beachtende Voxelisierung des Schnittes zwischen einer Fläche und einem Zylinder.

bestehend aus den Vektoren $\vec{a}$, $\vec{b}$ und $\vec{n}$ durch,

$$\vec{x}_{trans} = \begin{pmatrix} \alpha \\ \beta \\ \eta \end{pmatrix} = \begin{pmatrix} a_1 & b_1 & n_1 \\ a_2 & b_2 & n_2 \\ a_3 & b_3 & n_3 \end{pmatrix}^{-1} \cdot \vec{x}_{rel}\,. \tag{4.6}$$

Die ersten beiden Komponenten α und β des resultierenden Vektors geben nun die Position des Voxels bezüglich der die Fläche F aufspannenden Vektoren $\vec{a}$ und $\vec{b}$ an. Für die Voxel, die innerhalb der Fläche liegen gilt nun, dass sowohl α als auch β im Intervall $[0, 1]$ liegen. Alle anderen Voxel können eliminiert werden.

Mit dem gerade beschriebenen Algorithmus wird eine Voxelisierung der Fläche F unter der Beachtung der Hauptrichtung $h_{\vec{n}}$ generiert. Dies ergibt sich ganz klar aus der Konstruktion des Verfahrens.

Als nächster Hauptschritt im Algorithmus zur Konstruktion des beliebig geformten Einströmungsgebietes kommt die Voxelisierung des polygonalen Netzes entlang der Schnittfläche. Dieser Vorgang beruht erneut auf Strahlenschnitten. Abweichend von dem in Abschnitt 4.3.3 beschriebenen Verfahren, startet man hier Strahlen nur von den Zellen der voxelisierten Schnittfläche.

In Abbildung 4.12 kann man die Voxelisierung des Schnittes aus der Einströmungsfläche und einem Zylinder sehen. Die Fläche und der Zylinder sind hier leicht transparent dargestellt. Wie man erkennen kann, funktioniert das gerade beschriebene Verfahren auch bei angewinkelten Flächen einwandfrei.

Die Erzeugung der zwei Voxelschichten für die Fluidphase des einströmenden Fluids geschieht mit einer einfachen Kopieroperation, die entlang der Hauptrichtung durchgeführt wird. Dazu durchläuft man schrittweise die Voxelisierung in U- und V-Richtung. Findet der Algorithmus

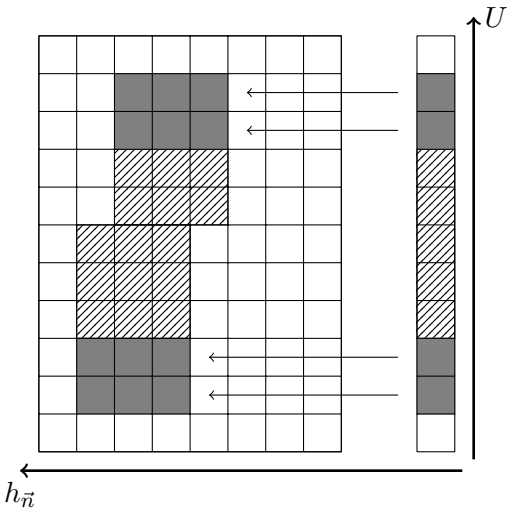


Abb. 4.13: Die beiden umrandenden Zellschichten (grau) werden über eine umgekehrte Projektion entlang der Hauptrichtung $h_{\vec{n}}$ neben den drei Voxelschichten (schraffiert) platziert.

entlang der Hauptrichtung ein Voxel, kopiert er dieses in die zwei nachfolgenden Zellen und versieht sie mit einem Flag, damit sie in einem späteren Schritt noch unterscheidbar sind.

Ein wichtiger Schritt des Algorithmus ist die Erzeugung der Voxelisierung der beiden umrandenden Hinderniszellenschichten. Hierzu wird zunächst die bereits bestimmte diskrete Geometriedarstellung auf die U-V-Ebene projiziert. In der Ebene lässt sich sehr einfach ein Rand generieren. Dazu durchläuft sie der Algorithmus und überprüft für jedes Voxel der Voxelisierung seine direkten Nachbarn. Sind diese nicht in der diskreten Geometrie enthalten, werden an deren Stelle *Randvoxel* eingefügt und darauf entsprechende Flags gesetzt.

Der Rand wird hier mit einer Projektion generiert weil seine direkte Erzeugung bei nicht achsenparallel voxelisierten Flächen kaum möglich ist. Außerdem ist es sinnvoll gerade über die Projektion in die Hauptrichtung $h_{\vec{n}}$ zu gehen, da der Rand auch genau in diese Richtung orientiert sein soll.

Zur Generierung der zweiten Randschicht in der U-V-Ebene wird das gerade beschriebene Verfahren unter Einbezug der zuvor bestimmten Randzellen erneut durchgeführt.

Die in der Projektionsebene erzeugte doppelte Umrandung wird über eine Art inverse Projektion auf die drei Voxelschichten im Raum übertragen (siehe Abbildung 4.13). Hierzu durchläuft der Algorithmus wiederum schrittweise die U-V-Ebene. Die darin enthaltenen Randvoxel werden im Raum entlang der Hauptrichtung neben die Voxelschichten platziert.

Für das Ergebnis dieses Prozesses betrachte man Abbildung 4.14 auf Seite 73. In dieser wird die Voxelschicht für die Dirichlet-Randbedingung zusammen mit der inneren Umrandung für alle drei Voxelschichten dargestellt. Wie zu erkennen ist, wird durch das gewählte Vorgehen ein in die Hauptrichtung orientierter Rand erzeugt.

Eine algorithmische Formulierung des Verfahrens zur Voxelisierung der beliebig geformten Einströmungsrandbedingung findet sich in Algorithmus 4.4.1 auf Seite 73.

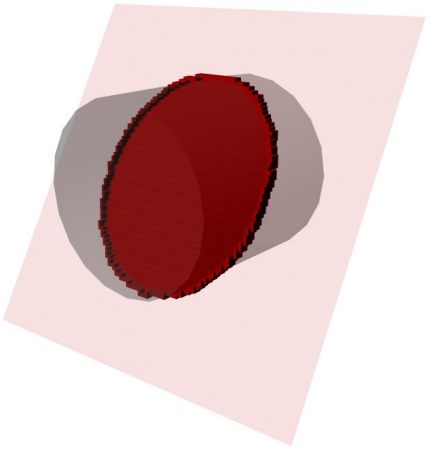


Abb. 4.14: Die Voxelisierung aus Abbildung 4.12 zusammen mit der inneren Umrandungs-Voxelschicht.

Algorithmus 4.4.1: VOXELIZECUSTOMINFLOWBOUNDARY($plane, mesh$)

$mainDirection = computeMainDirection(plane)$

$planeVoxelization = voxelizePlane(plane, mainDirection)$
$voxelization = voxelizeAlongVoxelization(mesh, planeVoxelization)$

$voxelization.setDirichletBoundaryFlag()$
$customInflowBoundaryVoxelization = copyWithOffset(voxelization, 0, mainDirection)$
$voxelization.setInflowFluidFlag()$
$customInflowBoundaryVoxelization = copyWithOffset(voxelization, 1, mainDirection)$
$customInflowBoundaryVoxelization = copyWithOffset(voxelization, 2, mainDirection)$

$UVPlane = project(customInflowBoundaryVoxelization, mainDirection)$
$setBorder(UVPlane, innerBorderFlag)$
$setBorder(UVPlane, outerBorderFlag)$
$inverseProjection(UVPlane, customInflowBoundaryVoxelization, mainDirection)$

return $(customInflowBoundaryVoxelization)$

Implementierung

Die Implementierung der Konfiguration des beliebig geformten Einströmungsrandes geschieht über die Methode `writeCustomInflowBoundary`. In dieser wird der im letzten Abschnitt beschriebene Algorithmus umgesetzt. Alle für den Algorithmus benötigten Informationen werden aus den der Methode übergebenen Knoten `SimulationCustomInflowBoundary` und `SimulationArea` ausgelesen. Zugriff auf das Mesh erhält die Funktion ebenfalls über den `SimulationCustomInflowBoundary`-Knoten. An diesen ist das polygonale Netz angekoppelt.

In der Implementierung lässt sich für den beliebig geformten Einströmungsbereich zusätzlich zur Hauptrichtung über das Attribute `direction` die Einströmrichtung des Fluids angeben. Dies hat, wie auch schon bei dem einfachen Einströmbereich, Einfluss auf die Reihenfolge der Voxelschichten.

Die Erzeugung der Parameter und Voxel-Dateien für den Strömungslöser kann nun auf Basis der gesetzten Flags sehr einfach geschehen. So müssen die einzelnen Voxelisierungen nur noch in die dazugehörigen Dateien geschrieben werden. Die genaue Position, an der die Voxel-Dateien eingefügt werden, lassen sich aus den vorhandenen Daten bestimmen. Sammelt die Methode `writeCustomInflowBoundary` analog zur Methode `writeInflowBoundary` alle Attribute aus dem Knoten für den Einströmrand, werden beispielsweise die nachfolgenden Parameter[10] in die Konfigurationsdatei des Strömungslösers geschrieben.

```
Voxel
{
      VoxelOffset <104,110,9>
      VoxelFile SimulationCustomInflowBoundary1.customInflow_outer_solid.voxel
      gas
}

Voxel
{
      VoxelOffset <104,110,9>
      VoxelFile SimulationCustomInflowBoundary1.customInflow_inner_solid.voxel
      liquid
}

Voxel
{
      VoxelOffset <104,110,9>
      VoxelFile SimulationCustomInflowBoundary1.customInflow_fluid.voxel
      fluid
      liquid
}

Voxel
{
```

[10] Das Beispiel entstammt dem in Abbildung 4.14 gezeigten beliebig geformten Einströmungsrand.

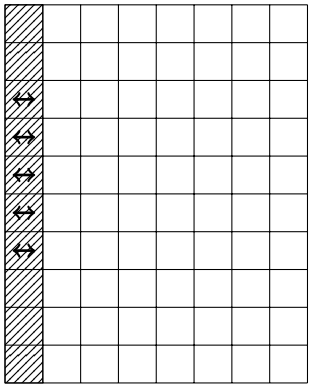


Abb. 4.15: Zu setzende Randbedingungen für einen Ausflussbereich.

```
        VoxelOffset <104,110,9>
        VoxelFile SimulationCustomInflowBoundary1.customInflow.voxel
        inflow <0.000000,0.000000,1.000000>
        liquid
}
```

4.4.5 Generierung von Ausflussbereichen

Die Erzeugung eines Ausflussbereiches bei einer Simulation ist unkomplizierter als die eines Einströmbereiches. Man betrachte hierzu Abbildung 4.15. In dieser werden Hinderniszellen auf dem Rand einer Simulationsumgebung gezeigt. Die Doppelpfeile markieren Zellen mit der *natürlichen Ausströmbedingung* (siehe Abschnitt 4.2). Tatsächlich reicht es im Fall der Ausflussbereiche, nur die entsprechende numerische Randbedingung festzulegen. Fallunterscheidungen und Sonderbehandlungen für mehrphasige Fluide werden nicht benötigt.

Eine Implementierung erfolgt analog zur Methode `writeSimulationInflowBoundary` in der Methode `writeSimulationInOutBoundary`. Diese voxelisiert die Zellschicht, auf der die Ausströmbedingung gesetzt wird. Zusätzlich wird deren Position im diskreten Voxelgitter bestimmt. Zusammen mit den Parametern für die Art des Ausströmbereiches und für dessen Geschwindigkeitskorrektur wird die Randbedingung in die an die Methode übergebene Konfigurationsdatei geschrieben. Ein so entstehender Eintrag könnte wie folgt aussehen:

```
Voxel
{
        VoxelOffset <0,41,0>
        VoxelFile SimulationInOutBoundary1.inOut.voxel
        inout 2,1
}
```

Hier wird ein Ausströmbereich des Typs 2 mit eingeschalteter Geschwindigkeitskorrektur an der Position `<0,41,0>` des Voxelgitters festgelegt. Die Größe der Randbedingung ergibt sich aus

der in der Datei `SimulationInOutBoundary1.inOut.voxel` übergebenen voxelisierten Geometrie.

Mit der Erläuterung der Erzeugung von Ausflussbereichen sind alle Methoden zur Ausgabe von Anfangs- und Randbedingungen beschrieben. Es stehen nun alle benötigten Mittel zur Generierung einer Szenenbeschreibung für den Strömungslöser zur Verfügung.

5 Extraktion der freien Oberfläche

Für eine Visualisierung der freien Oberfläche muss diese aus den Simulationsdaten gewonnen werden. Insbesondere soll aus den gegebenen Daten ein polygonales Netz, oder auch *Mesh*, erzeugt werden. Der ursprünglich hierfür im NaSt3DGPF-Löser verfolgte Ansatz ist ein *Post-Processing*–Schritt: Während der Simulationsrechnung wird für jeden zu visualisierenden Zeitschritt der gesamte für eine Rechnung benötigte Satz an Daten herausgeschrieben. Erst *danach* wird aus jedem Datensatz die freie Oberfläche extrahiert. In Anbetracht der großen Datenmengen, die bei einer dreidimensionalen Fluidsimulation entstehen, ist dieses Vorgehen für eine schnelle und vollautomatische Visualisierung der freien Oberfläche abzulehnen. Statt dessen empfielt es sich, direkt in den Löser ein Verfahren zur Oberflächenextraktion zu integrieren.

In diesem Kapitel wird der *Marching-Cubes*–Algorithmus zur Isoflächenextraktion vorgestellt. Er war in seiner ursprünglichen Form fehlerhaft, so dass zudem eine korrigierte Variante beschrieben wird. Einen wesentlichen Teil des Kapitels macht die Präsentation der Algorithmus-Implementierung aus. Das Verfahren wird in den Strömungslöser NaSt3DGPF eingebunden, so dass die Extraktion der Isofläche zum Level-Set–Wert Null während der Simulation erfolgt. Falls nur die Darstellung der freien Oberfläche von Interesse ist, kann so auf die Ausgabe der vollständigen Simulationsdaten verzichtet werden. Statt dessen werden nur polygonale Netze ausgegeben. Dies geht bei einer linearen Laufzeit in der Anzahl der Zellen im Diskretisierungsgitter des Navier-Stokes–Lösers sehr effizient.[1] Eine beschriebene Modifikation des Algorithmus ermöglicht seine Parallelisierung und führt so zu einer Beschleunigung.

5.1 Der Marching-Cubes–Algorithmus

Um die freie Oberfläche zu extrahieren, wird ein Verfahren zur Mesh-Generierung für *Isoflächen* benötigt. Angenommen es ist ein dreidimensionales, skalares Feld $f : \mathbb{R}^3 \to \mathbb{R}$ gegeben, dann ergibt sich die Isofläche S_α zum Isowert α durch

$$S_\alpha = \{\vec{x} \in \mathbb{R}^3 \mid f(\vec{x}) = \alpha\} \tag{5.1}$$

Das Standard-Verfahren für die Isoflächenextraktion ist der *Marching-Cubes–Algorithmus* von Lorensen und Cline [9].[2] Als ursprüngliche Anwendung wird in [9] die Visualisierung von medizinischen Daten aus der Computertomographie oder von Magnet-Resonanz-Scans vorgestellt. Da der Algorithmus universell anwendbar ist, bildet er heutzutage die Grundlage für die meisten Kontur-Darstellungen. In dieser Arbeit soll das Verfahren die Isofläche der Level-Set–Funktion

[1] Die in Abschnitt 8.3 vorgenommene Performance-Analyse des in dieser Arbeit implementierten Marching-Cubes–Algorithmus zeigt, dass der Linearzeit-Algorithmus tatsächlich auch schnell ist.

[2] Das *Dual-Contours*–Verfahren [44] kann für die hier angestrebte Anwendung nicht verwendet werden, da weder exakte Schnittpunkte der Level-Set–Funktion mit dem Diskretisierungsgitter bekannt sind, noch entsprechende Normalen.

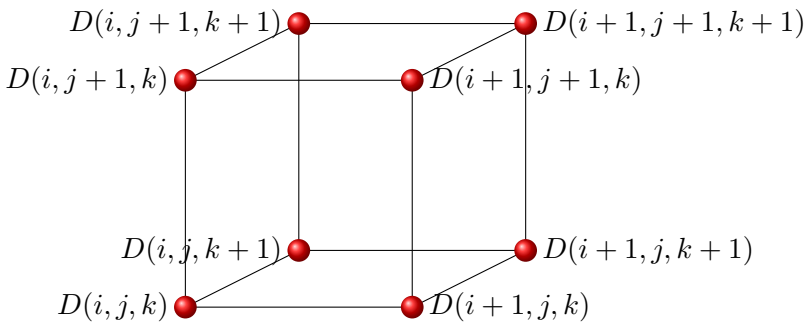


Abb. 5.1: Verwendete Werte in einem Einzelschritt.

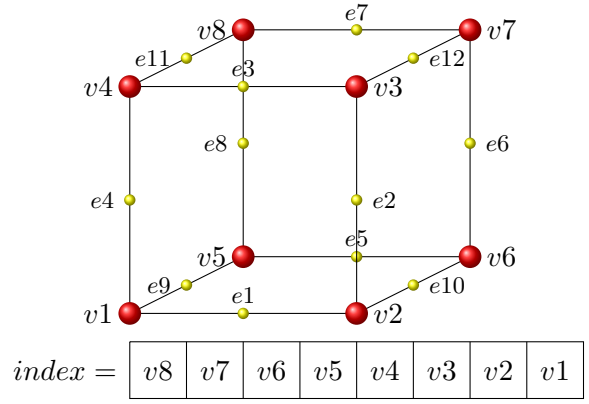


Abb. 5.2: Entstehung des Klassifikationsindexes.

ϕ zum Isowert Null ausgeben. Die genaue Funktionsweise des Marching-Cubes–Algorithmus wird im Folgenden vorgestellt.

Das Verfahren arbeitet auf einem diskreten, skalaren und dreidimensionalen Feld D. Dieses wird schrittweise durchlaufen. Dabei werden in einem Schritt immer acht Werte in D betrachtet, die zusammen einen Kubus ergeben. In Abbildung 5.1 kann man diese sehen.

Die Vorgehensweise ist die, dass in jedem Schritt in dem Kubus lokal eine Isofläche erzeugt wird. Dazu wird zunächst pro Schritt eine Klassifikation des gerade betrachteten Quaders durchgeführt: Für alle Eckpunkte wird entschieden, ob diese einen Wert kleiner oder gleich dem Isowert besitzen oder ob der Wert größer ist. Abhängig davon kann für jeden Punkt ein binärer Wert 1 oder 0 vergeben werden. Nummeriert man die Punkte in einer festen Reihenfolge durch, so kann ein eindeutiger Klassifikationsindex gefunden werden. Abbildung 5.2 deutet dies an.

Als nächstes muss festgelegt werden, welche Dreiecke erzeugt werden. Insbesondere bedeutet dies, dass die Anzahl der Eckpunkte der Dreiecke und deren Konnektivität gefunden werden

muss. Hier kommt nun die Klassifikation ins Spiel: Für jeden möglichen Index[3] können die benötigten Informationen im Vorhinein in eine Lookup-Tabelle gespeichert werden. Die darin enthaltenen Einträge können zur Laufzeit in konstanter Zeit ausgelesen werden.

Zur Erstellung der Einträge der Lookup-Tabelle erklären Lorensen und Cline, dass sich durch die Ausnutzung von Symmetrien die Anzahl der möglichen Klassifikationen auf 14 Grundtypen reduzieren lässt. Die Darstellung dieser Basisfälle ist in Abbildung 5.3 gegeben. Soll also mit einem Programm eine Tabelle mit den Konnektivitäten generiert werden, müssen nur 14 verschiedene Datensätze von Hand erzeugt werden. Alle anderen sind daraus ableitbar. Diese Vorgehensweise verringert die Gefahr von Fehlern bei der Erstellung der Einträge.

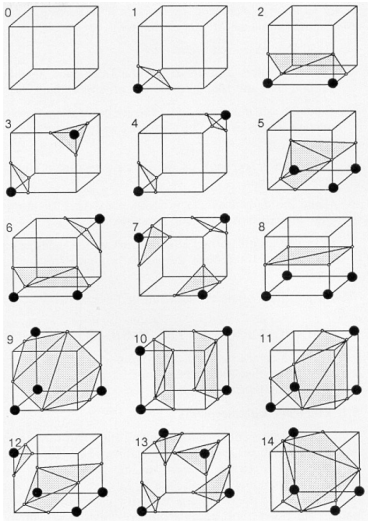


Abb. 5.3: Basisfälle der zu erstellenden Dreiecke für einen Kubus (Quelle: [9]).

Nach dem Nachschlagen der Eckpunkt-Konnektivität, muss die genaue Position eines jeden Dreieckspunktes im Mesh bestimmt werden. Dabei ist zu beachten, dass jeder der Punkte auf einer der Geraden zwischen den Ecken des Kubus liegt. In Abbildung 5.2 sind die Positionen der Eckpunkte durch die Punkte $e1$ bis $e12$ angedeutet. Die genaue Position wird nun über lineare Interpolation bestimmt. Angenommen es soll die Position $\vec{x}_{e1}$ von $e1$ zwischen den Eckpunkten $v1$ und $v2$ bezüglich des Isowertes α bestimmt werden. Dann geschieht dies wie folgt:

$$\vec{x}_{e1} = \left(1 - \frac{\alpha - D(i,j,k)}{D(i+1,j,k) - D(i,j,k)}\right) \vec{x}_{v1} + \frac{\alpha - D(i,j,k)}{D(i+1,j,k) - D(i,j,k)} \vec{x}_{v2} \tag{5.2}$$

In einem weiteren Schritt wird im Marching-Cubes–Algorithmus die Normale für jeden Eckpunkt[4] bestimmt. Dies geschieht über das Bilden der entsprechenden Gradienten im skalaren Feld D, lineare Interpolation und durch eine anschließende Normierung. Ist man nur an einzelnen Dreiecken interessiert, ist der Algorithmus nach diesem Schritt bereits fertig. Die sich daraus ergebende Vorgehensweise ist im nun folgenden Algorithmus 5.1 zusammengefasst.

[3] Es gibt insgesamt $2^8 = 256$ verschiedene Indices, da für jede Ecke ein binärer Wert vergeben wird.

[4] Eine Normale an einem Eckpunkt wird in der Computer Graphik für bestimmte Shading-Verfahren benötigt. In der vorliegenden Anwendung entspricht sie der Oberflächennormale der Isofläche an der Stelle des Eckpunktes.

Algorithmus 5.1.1: MARCHINGCUBES(D, α)

for each $(i, j, k) \in domain(D)$

do
$$\begin{cases} index = calculateIndex(cube(i,j,k)) \\ triangleList = lookupTriangles(index) \\ vertexList = interpolateVertexPositions(triangleList) \\ normalList = calculateNormals(vertexList) \\ output(vertexList) \\ output(normalList) \\ output(triangleList) \end{cases}$$

Der besondere Vorteil des Marching-Cubes–Algorithmus liegt in seiner Effizienz. Durch die fertige Lookup-Tabelle ist pro Gitterpunkt des diskreten Skalarfeldes D nur eine konstante Anzahl an Operationen nötig. Kann man, wie im Anwendungsfall der vorliegenden Arbeit, keine vereinfachenden Annahmen an die Topologie[5] der Isofläche machen und wird zudem stets nur eine[6] Isofläche pro Datensatz extrahiert, ist der Marching-Cubes–Algorithmus im komplexitätstheoretischen Sinne mit einer Laufzeit von $O(n)$ mit n der Anzahl der Werte im Feld D optimal.

5.2 Eine korrigierte Variante des Marching-Cubes–Algorithmus

Nach der Veröffentlichung des Marching-Cubes–Artikels [9] stellte sich heraus, dass der darin beschriebene Ansatz zur Isoflächenextraktion Inkonsistenzen bei der Erstellung der polygonalen Netze aufweist. Das in [9] beschriebenen Verfahren erzeugt Netze, die unerwünschte Löcher haben können. Der Grund für dieses Problem liegt in der Mehrdeutigkeit von bestimmten Klassifikationsergebnissen. Man betrachte hierzu Abbildung 5.4 auf Seite 81.

Diese Graphik zeigt zwei Kuben, bei denen die Eckpunkte der Frontflächen identisch klassifiziert werden. Bei beiden liegt der Wert der Punkte links unten und rechts oben oberhalb des Isowertes und der Wert der beiden anderen Punkte unterhalb. Da die restlichen Werte nicht alle übereinstimmen, wählt der Algorithmus verschiedene Indices für die beiden Situationen. Dies ist an sich kein Problem, sofern die Kanten der Dreiecke, die auf der Frontfläche liegen identisch wären. Hat man jedoch die Tabelle gemäß der 14 Basisfälle und deren Symmetrien aufgebaut, ist dies nicht der Fall. Grenzen nun zwei solche Kuben aneinander, kann es zu einem Loch kommen.

Die Mehrdeutigkeit entsteht dadurch, dass der Wert des skalaren Feldes zwischen den Eckpunkten nicht bekannt ist. Es kann also sein, dass zum Beispiel alle Werte im kontinuierlichen Skalarfeld auf der Diagonalen zwischen dem linken unteren und dem rechten oberen Punkt

[5] Falls genau eine geschlossene und zusammenhängende Isofläche im Feld D existiert, müssen nicht alle Werte des Feldes D betrachtet werden.

[6] Sollen Isoflächen zu mehr als einem Isowert extrahiert werden, kann ein Octree-basiertes Verfahren eine bessere Laufzeit haben.

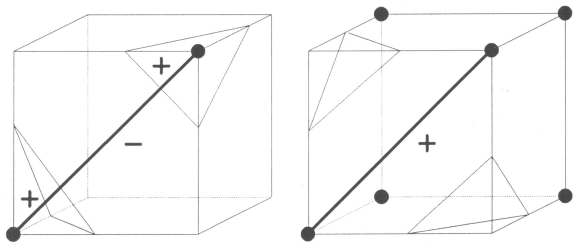


Abb. 5.4: Eine Inkonsistenz in der Lookup-Tabelle des Marching-Cubes–Algorithmus von Lorensen und Cline (Quelle: [55]).

oberhalb des Isowertes liegen. In diesem Fall ist die Wahl der Dreieckskanten wie im rechten Kubus korrekt. Ist zwischen den beiden Ecken an einer Stelle der Wert unterhalb des Isowertes, so müssen die Dreiecke Kanten wie bei dem linken Kubus enthalten.

Mehrere Veröffentlichungen machen Korrekturvorschläge für den Marching-Cubes–Algorithmus. In der vorliegenden Arbeit wird mit dem Lösungsansatz von Montani, Scateni und Scopigno [55] gearbeitet. Während andere Autoren eher heuristische Ansätze zur Umgehung der mehrdeutigen Basisfälle präsentieren, schlagen Montani, Scateni und Scopigno eine gegenüber dem ursprünglichen Algorithmus modifizierte Lookup-Tabelle vor: Lorensen und Cline gingen davon aus, dass eine komplementäre Klassifikation[7] nur das Umkehren der Dreiecksoberseiten zur Folge hat. Dahingegen schlagen Montani et al. bei 6 Basisfällen eine geänderte Geometrie für den komplementären Fall vor. Abbildung 5.5 auf Seite 82 zeigt die Korrekturen bei den Basisfällen.

In der linken Spalte finden sich die mehrdeutigen Basisfälle. Die mittlere Spalte zeigt den ursprünglichen Vorschlag von Lorensen und Cline für komplementäre Klassifikationen.[8] Bei den in der Abbildung angegebenen Fällen führt dies aber zu topologischen Inkonsistenzen. In der rechten Spalte stehen die von Montani et al. vorgeschlagenen Korrekturen für die komplementären Klassifikationen.

Erzeugt man mit diesen korrigierten Basisfällen eine Lookup-Tabelle, treten keine unerwünschten Löcher im Dreiecksnetz für die Isofläche auf. Die Autoren führen als einzigen Nachteil ihrer Methode an, dass nun pro Kubus maximal 5 statt 4 Dreiecke generiert werden können. An der Laufzeit-Komplexität des Algorithmus ändert dies nichts. Jedoch wird von den Autoren nicht ausgeführt, in welchem Maße sich diese Veränderung auf die tatsächliche Laufzeit auswirkt. Es wird nur erwähnt, dass das vorgestellte Korrekturverfahren in Bezug auf die Geschwindigkeit und Einfachheit „akzeptabler“ sei, als andere vorgeschlagene Korrekturen für den Marching-Cubes–Algorithmus.

[7] Bei dieser ist die für jeden Eckpunkt gefundene Angabe, ob der Wert unterhalb oder oberhalb des Isowertes liegt, genau entgegengesetzt. Entsprechend sind die Bits, die den Index bilden alle invertiert.

[8] Hier wird die Orientierung der Flächen durch die graue Einfärbung angedeutet.

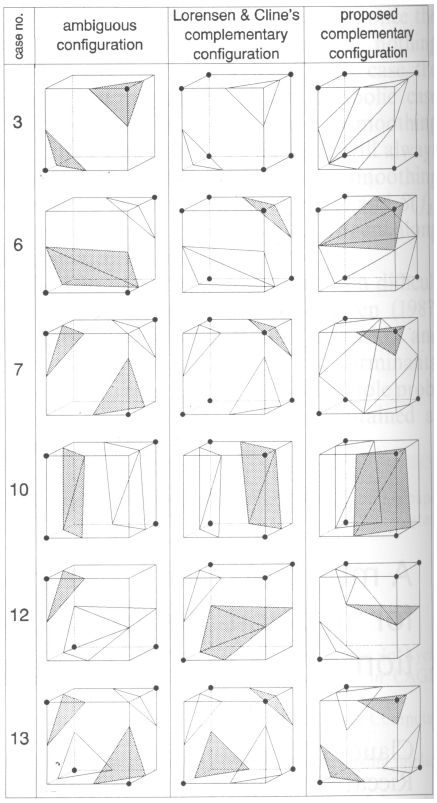


Abb. 5.5: Der Korrekturvorschlag von Montani et al. für die komplementären Klassifikationen von mehrdeutigen Basisfällen (Quelle: [55]).

5.3 Implementierung

Die Implementierung der Extraktion der freien Oberfläche arbeitet mit dem korrigierten Marching-Cubes–Algorithmus aus [55]. Neben dem Algorithmus fehlen zur praktischen Durchführung der Isoflächen-Generierung aber noch einige Details.

5.3.1 Dateiformat zur Mesh-Ausgabe

Zunächst ist ein geeignetes Dateiformat für die Ausgabe der polygonalen Netze zu finden. Die Wahl fällt hierbei auf das *Wavefront OBJ* - Dateiformat. Dieses ist ein ASCII-Format, das zeilenorientiert arbeitet. Es wurde von einer der Vorgängerfirmen von Autodesk entwickelt und ist heute ein freier und großflächig unterstützter Standard für polygonale Geometriedaten. Außerdem ist das Format sehr einfach strukturiert, so dass die Erzeugung entsprechender Dateien leicht fällt.

Der einzige Nachteil des Formates ist für die hier angestrebte Anwendung die Größe der erzeugten Dateien. Bedenkt man, dass große Dreiecksnetze leicht einige hunderttausend Eckpunkte haben können, ist die Verwendung eines ASCII-Formats nicht optimal. Dennoch erzeugt der Marching-Cubes–Algorithmus OBJ-Dateien, die je nach Simulationsaufgabe ungefähr ei-

ne Größenordnungen kleiner sind als die ursprünglichen Simulationsdaten (siehe Kapitel 8). Zudem lassen sich die erzeugten Dateien sehr leicht von aktueller Hardware verwalten. Weil außerdem nicht das Ziel der Echtzeit-Verarbeitung verfolgt wird, fällt die Wahl dennoch auf das OBJ-Format. Ausschlaggebend ist an dieser Stelle vor allem die Einfachheit und die große Verbreitung.

In dieser Arbeit ist ausschließlich die Erzeugung von polygonalen Netzen von Interesse. Für diese werden beim OBJ-Format nur zwei Sprachelemente benötigt: Zum einen lassen sich Eckpunkt-Koordinaten vorgeben und zum anderen können Dreiecksflächen definiert werden. Ein Eckpunkt an der Position $(1.0, 0.0, 1.0)$ wird durch eine Zeile der folgenden Form erzeugt:

```
v 1.0 0.0 1.0
```

Jeder erzeugte Eckpunkt erhält intern eine Referenznummer die bei 1 beginnend fortlaufend durchgezählt wird. Über diese lassen sich die Punkte ansprechen. Insbesondere kann dadurch eine polygonale Fläche beschrieben werden. Eine Fläche mit vier Eckpunkten kann beispielsweise so beschrieben werden:

```
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
v 0.0 1.0 0.0
f 1 2 3 4
```

Als alternative Zugriffsmöglichkeit auf einzelne Eckpunkte ist die Verwendung von negativen Referenznummern möglich. Mit dem Wert -1 referenziert man den letzten erzeugten Punkt, mit -2 den vorletzten, und so weiter. Das Viereck aus dem letzten Beispiel lässt sich damit alternativ wie folgt angeben:

```
v 0.0 0.0 0.0
v 1.0 0.0 0.0
v 1.0 1.0 0.0
v 0.0 1.0 0.0
f -4 -3 -2 -1
```

Der Funktionsumfang des OBJ-Dateiformats ist deutlich größer. Da aber in dieser Arbeit keine weiteren Funktionen benötigt werden, sei der interessierte Leser auf die Spezifikation [70] verwiesen.

5.3.2 Modifikationen des Marching-Cubes–Algorithmus zur verbesserten Laufzeit im Löser

In der vorliegenden Implementierung soll der Marching-Cubes–Algorithmus direkt in den Strömungslöser eingebaut werden. Das eingangs des Kapitels erwähnte Ziel dieser Integration ist die Vermeidung eines Post-Processings auf den vollen dreidimensionalen und zeitabhängigen Simulationsdaten. Falls nur die freie Oberfläche visualisiert werden soll, ist ein Abspeichern aller Simulationsdaten ineffizient. Sinnvoller ist hier ein in die Simulation integrierter Oberflächenextraktionsschritt. Dieser führt zu einer leichten, in der Praxis aber meist unerheblichen, Erhöhung der Laufzeit des Strömungslösers. Um die Laufzeiterhöhung während der Simulation

aber dennoch möglichst gering zu halten, sollen einige kleine Modifikationen am Marching-Cubes–Algorithmus durchgeführt werden. Diese führen dazu, dass ein 3D-Graphikprogramm, das die abgespeicherten Oberflächendaten lädt, noch ein minimales Post-Processing auf den polygonalen Netzen durchführen muss. Da die Nachbearbeitung aber nur für die Netze und nicht für die vollen Simulationsdaten erfolgt, ist die aufgewendete Zeit unerheblich.

Die erste Modifikation nutzt die Funktionalität des Wavefront OBJ - Formats aus, polygonale Flächen mit beliebig vielen Eckpunkten angeben zu können. Diese ermöglicht die Verringerung der Zahl der abzuspeichernden Flächen bei der Extraktion der freien Oberfläche. Im neunten Basisfall in Abbildung 5.3 (Seite 79) definiert man so beispielsweise nicht vier Dreiecksflächen sondern nur eine einzige Fläche, deren Umrandung über einen geschlossenen polygonalen Kantenzug mit sechs Eckpunkten gegeben ist. Das Programm, das die OBJ-Dateien lädt, rekonstruiert hieraus eine korrekte Triangulierung. Mit diesem stets vollautomatisch durchgeführten, schnellen Postprocessing-Schritt entfällt der zusätzliche Speicherplatz für die Flächen sowie die zusätzliche Laufzeit für die Ausgabe im Strömungslöser.

Die Erzeugung der korrekten Konnektivität zwischen den im Algorithmus generierten polygonalen Flächen ist ebenfalls ein Ansatzpunkt für Effizienzsteigerungen während der Simulation. Normalerweise müsste dafür Sorge getragen werden, dass Flächen, die Eckpunkte miteinander teilen, auch tatsächlich die selben Punkte in der OBJ-Datei verwenden. Auf die hierfür benötigten zusätzlichen Rechenschritte wird bei der vorliegenden Implementierung verzichtet. Die Wiederherstellung der Konnektivität erfolgt in der Zielapplikation Autodesk Maya. Diese bietet die Möglichkeit, Eckpunkte, die einen Abstand unter einem gewissen Schwellenwert haben, zu verschmelzen. Auch dies ist in der Regel außerordentlich schnell. Das Ergebnis eines solchen Verschmelzungsprozesses ist dann die korrekt hergestellte Geometrie.

Schließlich wird in dieser Arbeit auf die Berechnung der Eckpunkt-Normalen verzichtet. Maya kann diese vollautomatisch generieren. Damit entfällt nochmals ein erheblicher Rechenaufwand und die optischen Ergebnisse sind trotzdem stets sehr ansprechend.

Insgesamt werden bei der vorliegenden Implementierung tatsächlich nur die Rechenschritte während der Simulation durchgeführt, die unvermeidbar sind. Alle anderen Arbeiten erfolgen sehr effizient im Post-Processing der Oberflächendaten. Damit ist das gewünschte Ziel der Laufzeitreduktion während der Simulation erreicht.

5.3.3 Lookup-Tabelle

Das Herzstück des Marching-Cubes–Algorithmus ist die eingangs erwähnte Lookup-Tabelle. Über diese können Informationen über die Geometrie für jede mögliche Klassifikation eines Kubus gewonnen werden. In der hier vorgestellten Implementierung liegt die Lookup-Tabelle als ein Array `vType` von Structs des Typs `voxelType` vor. Tabelle 5.1 auf Seite 85 beschreibt die Elemente des `voxelType` - Structs.

Die Belegung des Arrays wird im Quelltext hart-kodiert. Die ersten beiden Einträge des Arrays lauten zum Beispiel:

```
vType[0] = {0, {}, ""}
vType[1] = {3, {1, 9, 4}, "f -1 -2 -3 \n"}
```

Wie man hier sehen kann entsprechen die Klassifikations-Indices der Kuben den Positionen der Einträge im Array. Bei einem Klassifikations-Index von eins wird zum Beispiel ein Dreieck mit

edgesCount	int	Anzahl der Kanten, auf denen ein Eckpunkt interpoliert werden muss (entspricht Anzahl der Eckpunkte im Kubus)
edges	int*	Indices der Kanten, auf denen ein Eckpunkt interpoliert werden muss
output	char*	Zeichenkette im Format der OBJ-Dateien zur Erzeugung der polygonalen Flächen

Tabelle 5.1: Die Elemente des Structs `voxelType`.

Eckpunkten auf den Kanten $e1$, $e9$ und $e4$ erzeugt.

Um Fehler bei der Erstellung der Lookup-Tabelle zu vermeiden, wird sie über ein eigenes Programm erstellt. Dieses beruht auf den modifizierten Basisfällen von Montani, Scateni und Scopigno. Aus den Basisfällen werden alle Einträge in geeigneter Weise abgeleitet. Das Generierungsprogramm wurde für diese Arbeit vollständig neu implementiert. Es liefert als Ausgabe die hart-kodierten Einträge des `vType`-Arrays als C++ - Quelltext. Der Quelltext wird in den implementierten Marching-Cubes–Algorithmus übernommen.

5.3.4 Umsetzung im Löser

Die Implementierung des Marching-Cubes–Algorithmus in den Strömungslöser NaSt3DGPF erfolgt über die Klasse `Visual`. Diese wird eigens für diese Arbeit neu eingeführt. `Visual` soll alle für die Visualisierung und Animation der Level-Set–Funktion benötigten Routinen enthalten.

Der Algorithmus wird über die Methode `MarchingCubes` ausgeführt. Er funktioniert genau wie in [9], [55] beschrieben: Es wird Schritt für Schritt über die einzelnen Kuben gelaufen und ein Klassifikationsindex bestimmt. Für diesen werden Informationen aus der Lookup-Tabelle ausgelesen. Mit den aus `vType` erhaltenen Informationen können auf sehr einfache Weise die entsprechenden Geometriedaten erzeugt und im OBJ-Format abgespeichert werden. Dazu läuft das Programm über die Elemente des `edges`-Arrays. Jede Zahl steht für eine der Kanten $e1$ bis $e12$ des Kubus, auf denen die Eckpunkte der Geometrie liegen (siehe Abbildung 5.2 auf Seite 78). Die Position der Eckpunkte wird aus den Koordinaten der Gitterpunkte der Level-Set–Funktion interpoliert und anschließend zeilenweise in die OBJ-Datei geschrieben. Eine Beschreibung für die polygonalen Flächen wird direkt aus der Variable `output` (siehe Tabelle 5.1) in die OBJ-Datei ausgegeben.

5.3.5 Erweiterung auf geschlossene polygonale Netze

Im Kontext von zweiphasigen Fluiden muss man zwischen offenen und geschlossenen Meshes bei der Isoflächenextraktion unterscheiden. Am besten wird dies an folgendem Beispiel klar: Wird ein Wasserbecken durch eine Simulationsumgebung modelliert, bei der der gesamte *Boden* mit Wasser bedeckt ist, zeigt eine normale Visualisierung der freien Oberfläche nur die Trennschicht zwischen Luft und Wasser (siehe Abbildung 5.6 auf Seite 86, links). Diese ist offen. Eine Visualisierung der freien Oberfläche mit eingefügten Flächen zwischen dem Wasser und

den Rändern des Simulationsgebietes (siehe Abbildung 5.6, rechts) führt zu einem geschlossenen Mesh. Je nach Anwendung kann eine Darstellung der freien Oberfläche wünschenswert sein. Deshalb wird im Rahmen der Implementierung der Oberflächenextraktion eine optionale Erweiterung zur Extraktion von geschlossenen polygonalen Netzen umgesetzt.

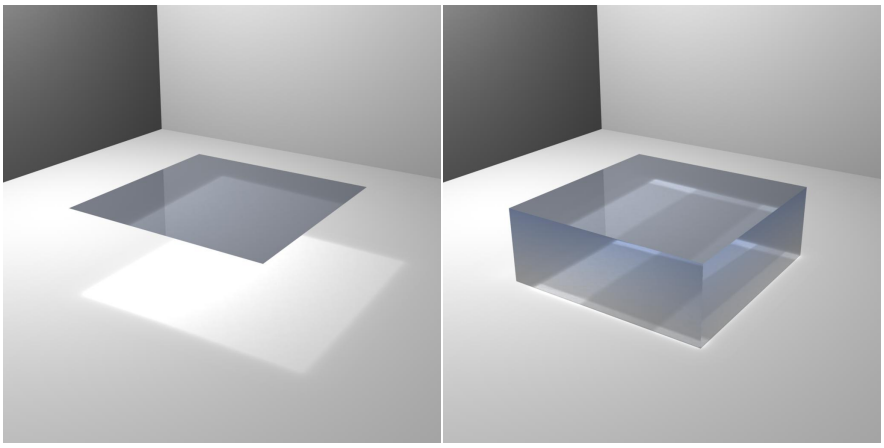


Abb. 5.6: Extrahierte Fluidoberfläche eines Wasserbeckens ohne (*links*) und mit (*rechts*) Abschluss an den Rändern.

Die Algorithmus-Idee zum Schließen der Meshes ist recht einfach. Zunächst wird an allen Rändern des Simulationsgebiets eine fiktive Schicht von Zellen mit der Dicke eins eingeführt. Diese trägt einen Level-Set–Wert, der in der Gasphase liegt und einen sehr großen Abstand zur Oberfläche suggeriert. Wendet man auf das so erweiterte Gebiet den Isoflächen-Extraktor an, so erzeugt dieser immer dann am Rand eine Fläche, wenn die fiktive Gaszelle zu einer Flüssigkeitszelle benachbart ist. Das sind genau die gewünschten Flächen.

Der Wert der Level-Set–Funktion in den fiktiven Zellen muss einen großen Abstand suggerieren, damit bei der Interpolation die Fläche genau auf den Rand des Simulationsgebiets gelegt wird. Andernfalls könnte sie ungewollt auf den Zellmittelpunkt gelegt werden, wodurch sich die Geometrie künstlich vergrößert.

Aus Effizienzgründen wird die fiktive Zellschicht bei der Implementierung nicht explizit konstruiert und abgespeichert. Statt dessen führen geeignete Fallunterscheidungen bei der Interpolation der Flächenpositionen zu dem selben Ergebnis bei einem geringeren Speicherverbrauch.

5.3.6 Parallelisierung

Bei der Verarbeitung von großen Datenmengen ist man immer an einer sinnvollen Verteilung der Aufgaben auf verschiedene Rechner interessiert. Hierzu wurde bereits in Abschnitt 2.1.3 die Idee für eine Parallelisierung des Navier-Stokes–Lösers vorgestellt: Die Gebietszerlegungsmethode sieht vor, die Datenfelder der Simulation geeignet zu unterteilen und die Lösungen auf entsprechend verteilten Gebieten zu berechnen. Aus diesen Teilergebnissen wird dann das Gesamtergebnis zusammengesetzt.

Sehr ähnlich soll hier bei der Parallelisierung des Marching-Cubes–Algorithmus vorgegangen werden. Anstatt diesen auf das gesamten Feld ϕ anzuwenden, wird in jedem einzelnen parallelen

Prozess das Verfahren auf den aus der Gebietszerlegung stammenden lokalen Gebieten Ω_h^p (siehe Abschnitt 2.1.3) ausgeführt. Damit erzeugt jeder Prozess ein Teilstück der Isofläche. Diese Teilstücke werden in getrennte Dateien geschrieben, so dass sogar eine voll parallele Datenhaltung der Meshes möglich ist.

Die parallele Implementierung der Isoflächenextraktion mit dem *Message Passing Interface* vereinfacht sich durch eine in Abschnitt 5.3.2 vorgestellte Optimierung. Vernachlässigt man die Konnektivität unter den einzelnen Geometrieflächen, muss keine Kommunikation zwischen den parallelen Prozessen stattfinden. Mit Hilfe der Kommunikation müssten die Prozesse normalerweise die aneinander stoßenden Flächen miteinander verbinden.

Da die in dieser Arbeit erstellte parallele Implementierung des Marching-Cubes–Verfahrens keinen Kommunikationsaufwand hat, ist die parallele Effizienz einzig von der Lastbalancierung im Strömungslöser NaSt3DGPF abhängig. Der Löser führt die Gebietszerlegung durch und erreicht damit bezüglich der Zahl der zu bearbeitenden Zellen eine Lastbalancierung nahe am Optimum. Insgesamt erfährt der implementierte Marching-Cubes–Algorithmus mit der Parallelisierung die erwünschte Beschleunigung (siehe Abschnitt 8.3).

5.3.7 Extraktionszeitpunkt

Die adaptive Schrittweitensteuerung des Navier-Stokes–Lösers (siehe Abschnitt 2.2.2) führt zu nicht äquidistanten Simulationszeitschritten. Für eine Visualisierung unter Einhaltung der Simulationszeit sollte deshalb die Isofläche zum Level-Set–Wert Null nicht in jedem Zeitschritt extrahiert und ausgegeben werden. Vielmehr ist eine Extraktion zu äquidistanten Zeitpunkten sinnvoll.

In der Implementierung der Oberflächenextraktion im NaSt3DGPF-Löser dient die Methode `WriteMesh` der korrekten zeitlichen Steuerung der Oberflächenextraktion. Über die Methode ist es möglich die Oberflächendaten sowohl zu jedem Zeitschritt, als auch in äquidistanten Zeitschritten auszugeben.

5.3.8 Konfiguration

Zur Kontrolle des Mesh-Generators werden im NaSt3DGPF-Löser einige neue Parameter für die Konfiguration aufgenommen. Diese beschreiben ein Zielverzeichnis, die Häufigkeit der Oberflächenextraktion und einiges mehr. Eine genaue Aufgliederung dieser und aller anderen im Rahmen dieser Arbeit neu in den Löser integrierten Konfigurationsparameter findet sich in Anhang A. Jeder dieser Parameter kann selbstverständlich auch über das Maya-Plugin zur Konfiguration kontrolliert werden.

6 Integration zeitabhängiger Meshes in Maya

Bei einer zweiphasigen Strömungssimulation können leicht mehrere tausend polygonale Netze für die Beschreibung der freien Oberfläche entstehen. Diese sollen während oder nach der Rechnung im Animationsprogramm Autodesk Maya geladen werden. Insbesondere muss es möglich sein, die Meshes zeitabhängig, das heißt zu einem über die *Timeline* von Maya vorgegebenen Zeitpunkt, anzuzeigen. Gleichzeitig sollen die Oberflächendaten in die Hindernisgeometrie der Simulation derart eingepasst werden, dass ein Rendern der gesamten Szene ohne Zusatzarbeiten direkt möglich ist.

In diesem Kapitel werden zunächst die Fähigkeiten von Maya diskutiert, die gerade beschriebenen Anforderungen zu realisieren. Hieran schließt sich die Konzeption eines neuen Plugins zur Integration zeitabhängiger Meshes an. Dessen Implementierung wird vorgestellt. Abgerundet wird das Kapitel durch einige Zusatzbemerkungen zur Weiterverarbeitung der geladenen polygonalen Netze.

6.1 Diskussion der von Maya bereitgestellten Funktionen

Auf den ersten Blick wirkt die Aufgabe, polygonale Netze im Wavefront OBJ - Datenformat in Maya zu laden sehr einfach, da deren Import per Knopfdruck möglich ist. Tatsächlich ist aber die große Anzahl der zu ladenden Oberflächen in Kombination mit deren Integration in das zeitabhängige Animations-Framework von Maya problematisch:

Durch die Anzahl der insgesamt zu ladenden polygonalen Netze wird es unmöglich, einzelne Ladeprozesse *manuell* zu starten. Hierfür muss eine automatisierte Skripting- oder Plugin-Lösung gefunden werden.

Eine weitere Schwierigkeit ergibt sich bei der Datenhaltung der Oberflächen. Versucht man diese alle gleichzeitig in eine 3D-Szene zu laden, erreicht man schnell die Kapazitätsgrenzen des Arbeitsspeichers. Eine durchaus übliche Zahl von über 100.000 Vertices pro polygonalem Netz und mehr als tausend Einzelnetzen zeigt, dass auch aktuelle Hardware hier nicht alle Oberflächen gleichzeitig verwalten kann.

Als Abhilfe bietet Maya einen Mechanismus an, um Teile von Szenen in eigene Dateien auszulagern. Über sogenannte *References* und *Proxies* soll zwischen den ausgelagerten Teilen beliebig hin- und hergeschaltet werden. Das Resultat ist ein deutlich verminderter Speicherbedarf, da jeweils nur eine Oberfläche im Speicher gehalten werden muss. Auch wenn dieser Ansatz vielversprechend zu sein scheint, scheitert eine Integration der zeitabhängigen Meshes in der Praxis an zwei Problemen:

Die Kopplung der darzustellenden Fluidoberfläche an die *Timeline* wird erschwert, da eine einfache Verknüpfung des *Proxy* mit der Animationszeit nicht möglich ist.

Das größere Problem liegt aber in der Konzeption des Proxy-basierten Auslagerungsmechanismus. Jedes einzelne ausgelagerte Szenenelement wird von Maya als ein eigenes Objekt

verwaltet. Dies bedeutet, dass sich das Setzen eines Materials oder einer anderen Eigenschaft für ein ausgelagertes Mesh auch nur auf genau dieses Mesh auswirkt. Schaltet man auf eine andere Oberfläche um, hat sie unabhängig davon ihre eigenen Eigenschaften. Das Festlegen eines Materials oder einer anderen Eigenschaft für die Fluidoberfläche muss somit für jeden Zeitschritt manuell oder über ein Skript erfolgen. Ein solches Vorgehen ist einerseits sehr unkomfortabel und andererseits auch extrem zeitaufwändig. Folglich soll hier für die Integration von zeitabhängigen Meshes von der Verwendung der Maya-eigenen Technik zum Auslagern von Szenenteilen abgesehen werden.

Aufgrund der Erweiterbarkeit durch Skripte und Plugins ist für Maya eine große Anzahl an freien und auch kommerziellen Zusatzpaketen verfügbar. Dem Autor dieser Arbeit war es aber trotz intensiver Bemühungen nicht möglich, eine Erweiterung zur Integration von zeitabhängigen Meshes zu finden. Deshalb muss an dieser Stelle davon ausgegangen werden, dass die im Folgenden vorgestellte Implementierung eines Plugins zur Integration zeitabhängiger Meshes in Maya das erste offiziell verfügbare Plugin dieser Art ist.

6.2 Konzeption

6.2.1 Anforderungen an das Plugin

In der Einleitung und der Diskussion wurden indirekt alle Forderungen an das Plugin zur Integration zeitabhängiger Meshes in Maya beschrieben: Als Datenbasis dienen die vom Strömungssimulator erzeugten Wavefront OBJ-Dateien, die die freie Oberfläche zu verschiedenen Zeitpunkten beschreiben. Diese sollen in Abhängigkeit von der im Animationsprogramm gerade gewählten Zeit geladen werden und unter Maya als Geometrie zur Verfügung stehen. Ihre Positionierung muss so erfolgen, dass sie mit der im Vorfeld der Simulation vorgegebenen Geometrie zusammenpassen. Gleichzeitig muss das Plugin so gestaltet sein, dass die eingelesenen freien Oberflächen im Dependency Graph nach außen hin durch *einen* Knoten repräsentiert werden. Nur dadurch ist es in einfacher Weise möglich, für alle Netze die selben Eigenschaften vorzugeben.

6.2.2 Darstellung polygonaler Netze in Maya

Um ein geeignetes Plugin für die gerade formulierten Anforderungen zu erstellen, bedarf es zunächst eines gewissen Hintergrundwissens zur Darstellung von polygonalen Netzen in Maya. Der einfachste Weg zum Verständnis der Mechanismen zur Anzeige der Meshes besteht in der Betrachtung der Knoten im Dependency Graph, die bei der Erzeugung eines der polygonalen Standardobjekte (wie Sphäre, Kubus, Zylinder,...) entstehen. Man betrachte hierzu Abbildung 6.1 auf Seite 91.

In dieser werden die Knoten gezeigt, die für die Darstellung einer Sphäre benötigt werden. `pSphere1` ist ein `transform`-Knoten, über den die Position der Geometrie festgelegt wird. Diesem untergeordnet ist ein `mesh`-Knoten (`pSphereShape1`). Bei diesem Knotentyp läuft die Verwaltung der Darstellung eines Meshes zusammen. Über eine entsprechende Verbindung wird dieser Knoten einem Shader zugeordnet. Im vorliegenden Beispiel wurde noch kein spezieller Shader für die Sphäre gewählt, so dass der allgemeine initiale Shader `initialShadingGroup` verwendet wird. Die eigentlichen Geometriedaten werden nicht im `mesh`-Knoten gespeichert.

Abb. 6.1: Dependency-Graph–Knoten, die bei der Erzeugung einer polygonalen Sphären-Geometrie entstehen.

Statt dessen werden diese über den Knoten `polySphere1` vom Typ `polySphere` erzeugt und über eine Verbindung zum `mesh`-Knoten weitergeleitet.

Betrachtet man die gerade beschriebene Struktur für den speziellen Fall der polygonalen Sphäre, fällt Folgendes auf: Der Knoten, der die Geometrie generiert, ist der `polySphere1`-Knoten. Ersetzt man diesen zum Beispiel durch einen Knoten des Typs `polyCube`, wird eine andere Geometrie (in diesem Fall die eines Würfels) dargestellt. Die Grundidee zur Bereitstellung einer durch ein Plugin benutzerdefinierten Geometrie besteht folglich in der Entwicklung eines Knotens, der sich nach außen hin wie die Knotentypen `polySphere` oder `polyCube` verhält, aber statt dessen individuell vorgegebene Meshes erzeugt.

Sowohl der `polySphere`- als auch der `polyCube`-Knoten haben ein Attribut vom Typ `kMesh`, das ein polygonales Netz beschreibt. Dieses füllen sie mit einer Sphären- beziehungsweise Kubus-Geometrie. Analog dazu muss also ein neuer Knoten entwickelt werden, der auch ein `kMesh`-Attribut enthält und es mit geeigneten Geometrie-Daten füllt. Verbindet man das Attribut mit einem `mesh`-Knoten und diesen wiederum mit einem Shader, können über den neuen Knoten individuelle Geometrien generiert werden. Um die korrekte Darstellung kümmert sich Maya vollautomatisch. Da die Geometrie somit variabel wird, der Mesh-Knoten aber immer der selbe bleibt, ist auch schon eine der anfangs gestellten Forderungen erfüllt: Die sich zeitabhängig verändernde Fluidoberfläche wird nur durch einen Knoten im Dependency Graph repräsentiert.

6.3 Implementierung

Das Plugin zur Integration zeitabhängiger Meshes soll im Folgenden den Namen `SimulationMeshNode` tragen. Es besteht aus zwei Komponenten. Eine Komponente ist der im vergangenen Abschnitt angedeutete neue Knoten vom gleichnamigen Typ `SimulationMeshNode`. Dieser dient der Einbindung der Geometriedaten. Die zweite Komponente ist ein Kommando `createMeshNode`, das der Erzeugung des `SimulationMeshNode` und aller zusätzlich benötigten Knoten und Verbindungen zwischen den Attributen dient. Das Zusammenspiel der einzelnen Knoten soll im Folgenden beschrieben werden.

6.3.1 Vernetzung der Knoten

Als Voraussetzung für die Nutzung des `SimulationMeshNode` wird die Existenz eines `SimulationArea`-Knotens angenommen. Durch die Ausführung des Kommandos `createMeshNode` sollen die in Abbildung 6.2 auf Seite 92 gezeigten Knoten `SimulationMeshNode1` und `polySurfaceShape1` erstellt werden.

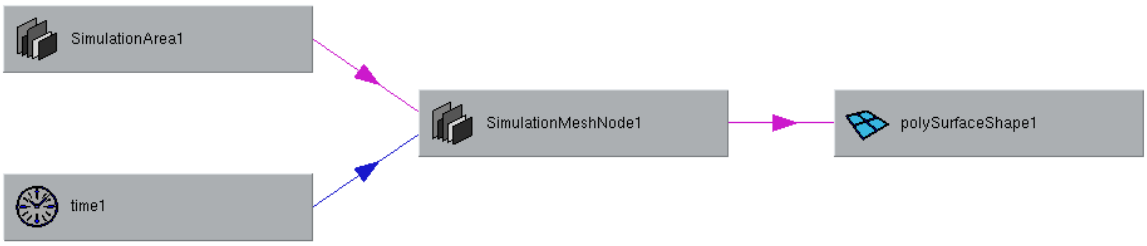


Abb. 6.2: Konstellation der Knoten nach der Ausführung des `createMeshNode`-Kommandos.

`SimulationMeshNode1` ist der zu implementierende Knoten. Dieser erhält über eine Verbindung mit dem `SimulationArea1`-Knoten das Verzeichnis der polygonalen Netze, die geladen werden sollen. Der `time1`-Knoten enthält die aktuelle Animationszeit von der Timeline und gibt diese an den zu implementierenden Knoten weiter. Aus der Zeit und dem Pfad bestimmt der Knoten vom Typ `SimulationMeshNode` die zu ladenden Dateien mit der freien Oberfläche. Ein Parser interpretiert das OBJ-Format und erzeugt daraus die Geometrie-Daten für das `kMesh`-Attribut des Knotens. Durch die Verbindung mit dem `mesh`-Knoten `polySurfaceShape1` wird die freie Oberfläche dargestellt.

Mit der hier beschriebenen Vernetzung der einzelnen Knoten lässt sich die gewünschte Funktionalität erreichen: Die Anbindung an den `time1`-Knoten sorgt für die Zeitabhängigkeit. Über den `polySurfaceShape1`-Knoten werden zum Beispiel Shader oder Glätter für die Oberflächen auf alle Meshes angewandt, ohne dass die Zeitabhängigkeit beachtet werden muss. Es fehlt nur noch die korrekte Positionierung der anzuzeigenden Oberflächen, um den Anforderungen aus Abschnitt 6.2.1 zu genügen. Dies geschieht durch die Unterordnung des `SimulationMeshNode1` unter den `transform`-Knoten der `SimulationArea1`. Damit werden die polygonalen Netze genau an der Position der `SimulationArea1` dargestellt. Hieraus ergibt sich die richtige Lage der Oberflächen bezüglich der Hindernisgeometrien.

Nachdem nun das Zusammenspiel der einzelnen Knoten beschrieben wurde, muss noch die Implementierung des `SimulationMeshNode`-Knotens und des dazugehörigen Befehls zur Erstellung des Knotens detaillierter vorgestellt werden.

6.3.2 Ein Knoten zur Behandlung zeitabhängiger Meshes

Der `SimulationMeshNode`-Knoten dient dem zeitabhängigen Laden der freien Oberflächen. Er ist ein Standardknoten, der von der Klasse `MPxNode` abgeleitet wird. In Tabelle 6.1 auf Seite 93 sind die Attribute des `SimulationMeshNode` aufgeführt und beschrieben. Diese decken sich genau mit den bisherigen Ansprüchen an den Knoten.

Die Attribute sind derart miteinander verknüpft, dass der Wert des `outMesh`-Attributs neu *berechnet* wird, wenn sich der Wert eines der anderen Attribute verändert. Zur Berechnung wird die Methode `compute` überschrieben. Sie liest die Werte aus den Attributen `time` und `meshPath` und parst die OBJ-Dateien mit der freien Oberfläche.

Zunächst bestimmt der Parser, der durch die Methode `loadGeometryData` implementiert wird, welche Meshdaten eingelesen werden müssen. Da die Timeline in Maya beliebig vergrößert

Attribut	Typ	Beschreibung
`time`	kTime	aktuelle Zeit in der Szene
`meshPath`	kString	Pfad zu den vom Simulator generierten OBJ-Dateien
`outMesh`	kMesh	eingelesene Mesh-Daten für den Zeitpunkt `time`

Tabelle 6.1: Beschreibung der Attribute des `SimulationMeshNode`.

Quell-Knoten	Quell-Attribut	Ziel-Knoten	Ziel-Attribut
SimulationArea1	meshOutputDirectory	SimulationMeshNode1	meshPath
time1	outTime	SimulationMeshNode1	time
SimulationMeshNode1	outMesh	polySurfaceShape1	inMesh

Tabelle 6.2: Verbindungen zwischen den Knoten, die bei der Ausführung von `createMeshNode` hergestellt werden.

werden kann, können nicht alle Frames in Maya eins zu eins ein Mesh zugeordnet bekommen. Das lässt sich umgehen, indem immer das letzte verfügbare polygonale Netz eingeladen wird, das zu einem Frame gehört. Angenommen es wurden nur hundert Meshes generiert und es soll das 120. Frame dargestellt werden, dann werden die OBJ-Dateien zum hundertsten Netz geladen.[1] Ein solcher Mechanismus ist vor allem dann sinnvoll, wenn Oberflächen für einzelne Frames fehlen: Hier wird dennoch immer ein Mesh angezeigt.

Die Hauptarbeit der Methode `loadGeometryData` besteht im Einlesen der von den parallelen Prozessen erzeugten OBJ-Dateien. Sie werden zeilenweise abgearbeitet. Alle dabei eingeladenen Geometrieinformationen über die Oberfläche werden in entsprechende Datenstrukturen abgespeichert. Um die Implementierung einfach zu halten, werden nur die Sprachelemente des Wavefront OBJ - Formats unterstützt, die auch bei der Generierung der Dateien im Strömungssimulator eingesetzt werden.

Aus den über den Parser gewonnenen Daten wird zum Abschluss der `compute`-Methode mit dem *Function Set* `MFnMesh` für polygonale Oberflächen ein Mesh in den für Maya üblichen Datenstrukturen erzeugt und an das Attribut `outMesh` übergeben.

6.3.3 Ein Kommando zur Erzeugung des Knotens

Das `createMeshNode`-Kommando übernimmt die Erstellung des `SimulationMeshNode` und der übrigen benötigten Knoten. Es wird von der Klasse `MPxCommand` abgeleitet.

Aufgerufen wird `createMeshNode` wahlweise mit einem oder keinem Argument. Wird ein Argument angegeben, muss dies der Name eines `SimulationArea`-Knotens sein. Falls kein Parameter angegeben wird, erwartet der Befehl einen ausgewählten `SimulationArea`-Knoten.

Die Funktionalität des `createMeshNode`-Kommandos wird über die Methode `doIt` implementiert. Nach den gerade beschriebenen Parameterabfragen wird die gewählte `SimulationArea` als Basis für die Erstellung des `mesh`-Knotens genutzt. Insbesondere wird also der Mesh-Knoten

[1] In diesem Beispiel wird von einer Eins-zu-Eins - Relation zwischen der Frame-Nummer und der Nummer des jeweils angezeigten polygonalen Netzes ausgegangen.

erzeugt und dem `transform`-Knoten der `SimulationArea` untergeordnet. Zusätzlich wird der neue `SimulationMeshNode`-Knoten generiert. Die von `createMeshNode` zu erstellenden Verbindungen zwischen den Attributen der einzelnen Knoten werden in Tabelle 6.2 auf Seite 93 beschrieben. Alle dort angegebenen Knotennamen entsprechen denen aus Abbildung 6.2 (auf Seite 92). Für ein korrektes Shading wird schließlich der `mesh`-Knoten der `initialShadingGroup` hinzugefügt.

Nach dem Aufruf des Kommandos `createMeshNode` stehen die durch den Strömungssimulator erstellten Meshes der freien Oberfläche in Maya zur Verfügung.

6.4 Weiterverarbeitung und Shading

In Abschnitt 5.3.2 wurde erwähnt, dass aus Performance-Gründen auf die Erzeugung der korrekten Konnektivität zwischen den polygonalen Flächen verzichtet wird. Um diese herzustellen wendet man auf den mit `createMeshNode` erstellten `mesh`-Knoten den Menü-Befehl `Edit Mesh→Merge` an. Dieser fasst Vertices eines Meshes zusammen, die einen Abstand unter einem Schwellenwert[2] haben. Hierdurch entsteht implizit die Konnektivität.

Die Festlegung des Shaders beziehungsweise Materials für die freien Oberfläche ist durch die Konzeption des Plugins sehr einfach, da sich die zeitabhängige Oberfläche wie ein gewöhnliches polygonales Netz verhält. Von besonderem Interesse für die Darstellung von Wasser ist der *Dielectric Shader*, der von dem mit Maya mitgelieferten Renderer *Mental Ray* angeboten wird. Dieser Shader simuliert physikalisch-basiert die Oberflächeneigenschaften von *dielektrischen* Materialien, wie Glas oder Wasser. Für eine umfangreiche Beschreibung seiner Eigenschaften sei auf ein Grundlagenwerk zu Mental Ray [50] verwiesen. Alle im Rahmen dieser Arbeit generierten Visualisierungen und Animationen der freien Oberfläche verwenden diesen Shader.

Wenn alle Materialien, Beleuchtungselemente und Kameras für eine Visualisierung festgelegt sind, bleibt als letzter Schritt in Maya das Rendering. Hierfür wird der gerade erwähnte Renderer verwendet.

[2] Der Schwellenwert muss geringer als die Hälfte der Länge der kürzesten Polygonkante sein.

7 Zwei-Wege-Kopplung zwischen der Level-Set–Methode und Partikeln

Die Animation von zweiphasigen Fluiden ist ein essentieller Bestandteil aktueller Forschung im Bereich der Computer Graphik. Als wesentliche Herausforderung wird in diesem Bereich der visuelle Realismus von animiertem Wasser[1] verstanden. Häufig werden für die Animation stark vereinfachte Modelle für Fluide verwendet. Diese genügen meist nicht dem Anspruch der physikalischen Korrektheit.

In Rahmen dieser Arbeit wurde bereits ein vollständiges System zum physikalisch korrekten Simulieren und Visualisieren von zweiphasigen Fluiden in Autodesk Maya vorgestellt und implementiert. Unter Einsatz von moderner Hochleistungshardware können so beliebig genaue und realistische Simulationen und Visualisierungen von Strömungen erstellt werden. Ein Ziel der Computer Graphik ist es, darüber hinaus auch äußerst kleinskalige Phänomene wie Wasserspritzer, zu animieren. Hierdurch wird der visuelle Realismus nochmals gesteigert. Doch selbst die eben erwähnte Hochleistungshardware ist bis heute nicht im Stande bei extrem großskaligen Simulationen, wie Überflutungen, winzigste Details realistisch darzustellen.

An dieser Stelle setzt diese Arbeit an. Es wird im Folgenden ein Verfahren zur Zwei-Wege-Kopplung der Level-Set–Methode mit einem Partikelsystem vorgestellt. Durch das Partikelsystem werden kleinskalige Phänomene, wie Spritzer, aufgelöst. Gleichzeitig wird die Implementierung und Ankopplung des Partikelsystems an den Navier-Stokes–Löser NaSt3DGPF beschrieben. Äußerst wichtig ist auch die Parallelisierung des Systems. Diese wird erörtert. Zur hochqualitativen Darstellung der Ergebnisse wird erneut Maya herangezogen. Dabei wird das Verfahren in das bisher vorgestellte System zur Fluid-Simulation integriert.

7.1 Das Partikelsystem

7.1.1 Modell

Zur Darstellung kleinskaliger Fluidphänomene soll, wie erwähnt, ein Partikelsystem zum Einsatz kommen. Häufig kommen im Umfeld der Visualisierung *masselose* Partikel zum Einsatz. Diese werden erzeugt und dann auf Basis eines Geschwindigkeitsfeldes fortbewegt. Ein Einsatz von masselosen Partikeln ist jedoch für die hier angestrebte Anwendung falsch, da dadurch die Schwerkraft außer Acht gelassen wird. Benötigt wird also ein Modell, das das Verhalten von *massebehafteten* Fluid-Partikeln, also Wassertropfen, beschreibt.

Ein solches Modell wird im Paper von Thürey et al. [61] vorgestellt. Es soll für diese Arbeit übernommen werden. In der Publikation wird ein Wassertropfen durch seine Position $\vec{x}_P$, seine

[1] In diesem Kapitel wird stets Wasser als Beispiel für Flüssigkeiten mit Spritzern oder Gischt verwendet. Tatsächlich arbeitet das hier vorgestellte Verfahren aber für beliebige Flüssigkeiten.

Geschwindigkeit $\vec{u}_P$ und seinen Radius r_P charakterisiert.[2] Die Dichte von Wasser ρ_W und Luft ρ_L wird in dieser Arbeit gemäß der entsprechenden Dichte ρ_1 und ρ_2 in den zweiphasigen Navier-Stokes–Gleichungen gesetzt (siehe Abschnitt 2.2.1). Durch diese Wahlen ergibt sich die Masse m_P eines Partikels mit dem Radius r_P wie folgt[3]:

$$m_P = \rho_W \frac{4}{3} \pi \, {r_P}^3 \tag{7.1}$$

Die eigentliche Modellierung des Verhaltens eines Tropfens geschieht über die Newtonschen Gesetze. Danach ergibt sich die Geschwindigkeit aus

$$\vec{u} = \frac{\partial \vec{x}}{\partial t}. \tag{7.2}$$

Für die Beschleunigung $\vec{a}$ gilt die bekannte Relation zur Geschwindigkeit $\vec{u}$:

$$\vec{a} = \frac{\partial \vec{u}}{\partial t} \tag{7.3}$$

Eine Kraft $\vec{F}$ ist nach Newton durch die Masse m und die Beschleunigung $\vec{a}$ gegeben:

$$\vec{F} = m\vec{a} \tag{7.4}$$

Schließlich ergibt sich die Kraft $\vec{F}_P$, die auf ein einzelnes Partikel wirkt, aus der Summe aller wirkenden Kräfte $\vec{F}_i$:

$$\vec{F}_P = \sum_i \vec{F}_i \tag{7.5}$$

Thürey et al. modellieren nun die auf den Tropfen wirkende Kraft, indem alle Kräfte bis auf die Gravitationskraft $\vec{F}_G$ und die Luftwiderstands-Kraft $\vec{F}_D$ vernachlässigt werden. Damit ergibt sich aus den Gleichungen (7.3), (7.4) und (7.5) der folgende Ansatz:

$$m_P \frac{\partial \vec{u}_P}{\partial t} = \vec{F}_G + \vec{F}_D \tag{7.6}$$

Unter Verwendung der Gravitation $\vec{g}$ erhält man für $\vec{F}_G$

$$\vec{F}_G = m_P \vec{g}. \tag{7.7}$$

Die Luftwiderstands-Kraft des Tropfens wird über ein Standardmodell ([69], Seite 123f) für den Luftwiderstand bei turbulenten Strömungen berechnet:

$$\vec{F}_D = \frac{1}{2} C_D \, \rho_L \, A \, \vec{u}_{rel} \, |\vec{u}_{rel}| \tag{7.8}$$

A steht dabei für die Querschnitts-Fläche, die ein umströmter Gegenstand hat. Bezüglich dieser Fläche wird der Luftwiderstand berechnet. Da in diesem Fall ein Tropfen, also idealisiert eine

[2] Die hier verwendete Notation weicht von der im Originalpaper ab, damit eine einheitliche Lesart in dieser Arbeit ermöglicht wird.

[3] Der Tropfen wird hier als sphärisch angenommen.

Kugel, umströmt wird, ergibt sich die Querschnittsfläche aus der Fläche einer Kreisscheibe zu

$$A = \pi r^2. \tag{7.9}$$

Mit $\vec{u}_{rel}$ wird in Gleichung (7.8) die Relativgeschwindigkeit der Luft bezüglich der Geschwindigkeit des Partikels bezeichnet. Es gilt also

$$\vec{u}_{rel} = \vec{u} - \vec{u}_P, \tag{7.10}$$

wobei $\vec{u}$ für die Geschwindigkeit der Luft steht. Im Paper von Thürey wird für $\vec{u}$ überall konstant der Wert $(0, 0, 0)^t$ angenommen. Es wird also weder eine Luftströmung simuliert, noch eine konstante Luftgeschwindigkeit verwendet. In der vorliegenden Arbeit soll dem gegenüber das physikalisch korrekt simulierte Geschwindigkeitsfeld der Luft verwendet werden. Damit ist $\vec{u}$ also die Geschwindigkeit der Luft an der Stelle $\vec{x}_P$ im Geschwindigkeitsfeld der Navier-Stokes–Gleichungen. Das Resultat dieser Wahl ist ein wesentlich realistischeres Verhalten der Tropfen. Auf der einen Seite werden Effekte wie die Verdrängung der Luft durch das Wasser und der damit verbundene Transport der Partikel mit einbezogen. Auf der anderen Seite kann Wind durch geeignete Einströmungsbedingungen in der Gas-Phase simuliert werden.

Der Faktor C_D wird als *Widerstandsbeiwert* bezeichnet. Er hängt von Einflüssen wie der Oberflächen-Form und -Rauigkeit und der Stärke der Turbulenz in der Luft ab. Da eine numerische Simulation zur Gewinnung eines hochgenauen Wertes für C_D zu teuer ist, wird C_D in [61] durch mehrere Annahmen für einen bestimmten Parameterbereich approximiert. Man beruft sich dabei auf Ergebnisse der Meteorologie, wonach bestimmte Tropfengrößen zu bestimmten finalen Tropfengeschwindigkeiten führen und schränkt die Größe der Tropfen künstlich auf zwischen $r_1 = 0.005m$ und $r_2 = 0.0005m$ ein. Zwischen diesen beiden Werten und den aus anderen Annahmen resultierenden Werten für C_D wird dann mittels bilinearer Interpolation folgender Wert für C_D berechnet:

$$C_D = \frac{|\vec{u}_{rel}|}{u_{t2} + (u_{t1} - u_{t2})u_r}(\frac{1}{2} + \frac{1}{2}u_r) \tag{7.11}$$

Die Werte $u_{t1} = 9\frac{m}{s}$ und $u_{t2} = 2\frac{m}{s}$ sind Grenzgeschwindigkeiten, die für die Tropfen der Größen r_1 respektive r_2 angenommen werden. u_r wird via

$$u_r = \frac{r_1 - r_P}{r_1 - r_2} \tag{7.12}$$

gewonnen.

Durch die vorgestellten Gleichungen ist das Verhalten eines massebehafteten Teilchens in einem Luft-Geschwindigkeitsfeld vollständig charakterisiert. Der nächste Schritt besteht in der zeitlichen Diskretisierung der Gleichungen (7.2) und (7.6).

7.1.2 Diskretisierung

Zur Diskretisierung der Gleichungen (7.2) und (7.6) wird in beiden Fällen in [61] eine Euler-Zeitdiskretisierung[4] verwendet. Es wird von einem Zeitschritt der Größe Δt ausgegangen, wobei dieser im Rahmen der vorliegenden Arbeit jeweils mit der aktuellen Zeitschrittweite zum Lösen

[4] Die Genauigkeit des Eulerschrittes reicht für visuelle Effekte aus.

Variablen	Typ	Einheit	Beschreibung
`x`, `y`, `z`	`double`	$[m]$	Position des Tropfens im Simulationsgebiet
`u`, `v`, `w`	`double`	$[\frac{m}{s}]$	Geschwindigkeitskomponenten des Tropfens
`r`	`double`	$[m]$	Tropfenradius
`m`	`double`	$[kg]$	Masse des Tropfens (wird automatisch berechnet)
`injection_time`	`double`	$[s]$	Zeitpunkt, zu dem der Tropfen startet

Tabelle 7.1: Beschreibung der wesentlichen Variablen der Klasse `Drop`.

der Navier-Stokes–Gleichungen gleichgesetzt wird. Damit wird Gleichung (7.2) zum Neuberechnungsschritt

$$\vec{x}_P(t + \Delta t) = \vec{x}_P(t) + \Delta t\, \vec{u}_P(t) \tag{7.13}$$

für die Position eines Wassertropfens. In der gegenüber dem Paper fehlerkorrigierten Fassung lautet die Diskretisierung von Gleichung (7.6)

$$\vec{u}_P(t + \Delta t) = \vec{u}_P(t) + (\vec{F}_G + \vec{F}_D)\Delta t\, {m_P}^{-1}. \tag{7.14}$$

Mit dieser Diskretisierung sind alle benötigten Mittel vorhanden, um die Bewegung eines massebehafteten Partikels iterativ zu berechnen. Vorgegeben werden muss nur eine initiale Geschwindigkeit und Position, sowie der Radius des Tropfens.

7.1.3 Implementierung

Die Implementierung des Partikelsystems erfolgt durch die Erweiterung des NaSt3DGPF-Lösers. Das hier vorgestellte Partikelsystem fußt in der Implementierung auf einem vorhandenen parallelisierten Partikelsystem für masselose Teilchen, das vollständig umgeschrieben wird. Intern werden die Wassertropfen bzw. massebehafteten Partikel über Instanzen einer Klasse `Drop` dargestellt. Die wesentlichen Variablen der Klasse finden sich in Tabelle 7.1. Bis auf `injection_time` sind alle dort angegebenen Variablen die trivialen Überführungen aus dem Modell. `injection_time` beschreibt den Zeitpunkt in Simulationszeit zu dem ein Wassertropfen mit der initial vorgegebenen Position, Geschwindigkeit und Größe gestartet werden soll. Die Methode `AdvanceDrop` der Klasse berechnet für den jeweiligen Tropfen einen Zeitschritt der Zeitdiskretisierung aus dem letzten Abschnitt.

Insgesamt wird das Partikelsystem über eine Klasse `DropSystem` verwaltet. Mit entsprechenden Methoden kann man Tropfen in das System einfügen. Dazu wird die Startposition, die Startgeschwindigkeit, die Größe und der Startzeitpunkt angegeben. Solange der Startzeitpunkt während einer Simulation noch nicht eingetreten ist, wird ein Partikel über die `injection_list` verwaltet. Gestartete Partikel sind in einer Liste mit dem Namen `drop_list`. Für alle in dieser Liste enthaltenen Partikel wird in jedem Zeitschritt die `AdvanceDrop`-Methode aufgerufen, so dass deren Position jeweils neu berechnet wird. Ebenso wird in jedem Zeitschritt überprüft, ob ein Partikel auf Grund seiner `injection_time` von der `injection_list` in die `drop_list` übernommen werden muss und somit gestartet werden soll. Generell werden Partikel gelöscht, sobald sie durch eine Positionsaktualisierung in Zellen kommen, die als *Solid* geflaggt sind. Übertragen bedeutet dies, dass Wassertropfen in soliden Objekten entfernt werden.

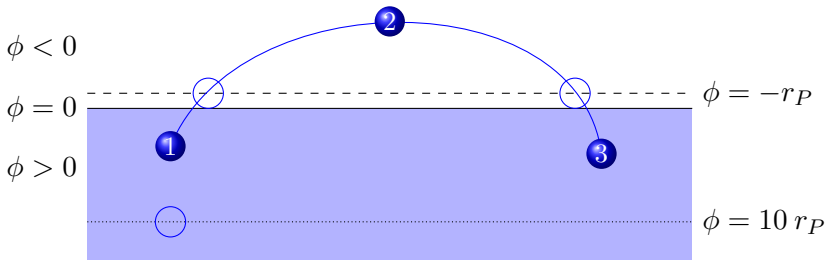


Abb. 7.1: Die drei Zustände eines Tropfenfluges.

Selbstverständlich muss auch der Eintritt des Tropfens ins Wasser behandelt werden. Unabhängig von der Rückkopplung der massebehafteten Partikel mit der Fluidoberfläche, die in Abschnitt 7.2.2 erläutert wird, muss vom Partikelsystem entschieden werden, wann ein Tropfen in der Flüssigkeit zu entfernen ist. Dazu werden bei der Bewegung des Partikels im System drei wesentliche Zustände unterschieden. Abbildung 7.1 zeigt diese.

Im ersten Zustand befindet sich ein Wassertropfen mindestens teilweise im Wasser. Dies ist dann der Fall, wenn der Wert ϕ der Level-Set–Funktion an der Position des Partikels bei $\phi \geq -r_P$ liegt. Der zweite Zustand ist der Flug-Zustand des Partikels. Entsprechend ist der Wert ϕ in diesem Zustand stets bei $\phi < -r_P$. Ab dem Zeitpunkt des Eintauchens, also für $\phi \geq -r_P$ befindet sich ein Tropfen im dritten Zustand. Es ist zu beachten, dass ein Partikel sich nur dann im ersten Zustand befinden kann, wenn er auch zumindest zum Teil in der Flüssigkeit gestartet wird. Andernfalls ist er direkt im zweiten Zustand.

Mit den gerade definierten Zuständen kann das Partikelsystem eindeutig entscheiden, wann ein Tropfen ins Wasser eingetaucht ist und somit gelöscht werden muss. Dies ist stets erst in Zustand drei der Fall. Insbesondere soll ein Tropfen in diesem Zustand gelöscht werden, wenn der Wert der Level-Set–Funktion ϕ an seinem Mittelpunkt bei $\phi \geq r_P$ liegt. Das ist genau dann, wenn der Tropfen vollständig eingetaucht ist.

Die Praxis zeigt, dass die Löschbedingung von $\phi \geq r_P$ aufgrund von Diskretisierungsungenauigkeiten etwas aufgeweicht werden muss. So kann es bei nicht fein genug aufgelösten Simulationen dazu kommen, dass der Tropfen aus visuellen Gesichtspunkten zu früh gelöscht wird. Bei der Implementierung des Partikelsystems wird deshalb ein Sicherheitsfaktor $dropRemovalRadiusFactor$ eingeführt, mit dem die Löschbedingung $\phi \geq r_P \cdot dropRemovalRadiusFactor$ lautet. Ein Wert von 4 für diesen Faktor leistet in den meisten Fällen das Erwünschte.

Schaut man sich die Definition der in Abbildung 7.1 gezeigten Zustände noch einmal genauer an, erkennt man, dass diese noch nicht alle möglichen Eventualitäten berücksichtigt. So kann es vorkommen, dass ein Tropfen den ersten Zustand nie verlässt. Das ist genau dann der Fall, wenn der Tropfen im Wasser gestartet wird, aber seine Geschwindigkeit nicht ausreicht um in den zweiten Zustand zu kommen. Typischerweise verliert ein solcher Tropfen aber aufgrund der Gravitation an Höhe und *fällt* somit im Wasser immer tiefer. Für diesen Fall ist bei der Implementierung des Partikelsystems eine zusätzliche Löschebene einzuplanen: In dieser Arbeit wird jedes Partikel gelöscht, an dessen Position die Level-Set–Funktion ϕ einen Wert von $\phi \geq 10\, r_P$ hat.

Parallelisierung

Da der NaSt3DGPF-Löser mittels der Gebietszerlegungsmethode vollständig parallelisiert ist, wird das Partikelsystem in dieser Arbeit ebenfalls parallelisiert. Dies ist unumgänglich, weil bei einer parallelen Navier-Stokes–Rechnung jeder beteiligte Prozessor ein Teilgebiet des Simulationsgebiets zugewiesen bekommt. Entsprechend liegt lokal auch nur ein Teil des Geschwindigkeitsfeldes vor (siehe Abschnitt 2.1.3). Weil das Geschwindigkeitsfeld aber für die Berechnung der Partikelpositionen gebraucht wird, muss das Simulationsgebiet für das Partikelsystem analog zu den Navier-Stokes–Gleichungen via Gebietszerlegung aufgeteilt werden. Ein Prozessor verwaltet dann jeweils die in seinem Teilgebiet vorkommenden Partikel.

Die Schwierigkeit bei diesem Ansatz besteht in der Verknüpfung der einzelnen Rechenprozesse. Einerseits muss dafür gesorgt werden, dass Partikel, die in einem Prozess in die `injection_list` eingefügt werden im Bedarfsfall geeignet an andere Prozesse weitergegeben werden. Andererseits müssen die Partikel in der `drop_list`, die durch Anwendung der `AdvanceDrop`-Methode weiterbewegt werden, dabei aber das Teilgebiet des Prozesses verlassen, ebenso an den entsprechenden Prozess weitergegeben werden.

Beide Probleme werden unter der Verwendung der Methode `ShouldCommunicate` der Klasse `Drop` gelöst. Diese bestimmt, ob ein Tropfen mit seiner absoluten Position $\vec{x}_P$ nicht mehr auf dem Diskretisierungsgitter des aktuellen Prozesses liegt. Ist das der Fall, muss eine Kommunikation zwischen den parallelen Prozessen stattfinden. Dabei werden sowohl bei der `injection_list` als auch bei der `drop_list` die Eigenschaften der Partikel, die das lokale Simulationsgebiet verlassen haben, an alle anderen Rechenprozesse verschickt. Die Prozesse, die die Partikel weiter bearbeiten fügen sie in ihre Listen ein. Alle anderen Prozesse verwerfen die empfangenen Tropfen. Dies geschieht ebenfalls über die `ShouldCommunicate`-Routine.

Die Kommunikation sowie die Verwaltung der parallelen Prozesse im NaSt3DGPF-Löser erfolgt über das *Message Passing Interface* (MPI). Dementsprechend werden für die Kommunikationsschritte bei der Parallelisierungen des Partikelsystems auch Methoden des MPI verwendet.

Algorithmus

Die in diesem Teilabschnitt beschriebene Vorgehensweise zum Implementieren und Parallelisieren des Partikelsystems wird im Algorithmus 7.1.1 (Seite 101) zur zeitlichen Evolvierung des Partikelsystems zusammengefasst.

7.2 Die Kopplung

In diesem Abschnitt soll die Kopplung des im letzten Abschnitt beschriebenen Partikelmodells an die Level-Set–Methode bzw. die Navier-Stokes–Gleichungen beschrieben werden. Die Kopplung besteht aus zwei Teilen. Der erste Teil ist die Generierung der Wassertropfen. Als Rückkopplung wird im Folgenden die Modellierung eines Tropfeneinschlags auf die *Wasseroberfläche*, also die Modellierung einer Impulsübertragung von den Partikeln auf die Level-Set–Nullkontur beim Auftreffen eines Tropfens, verstanden.

Algorithmus 7.1.1: EVOLVEDROPSYSTEM(Δt)

```
for each process
do { for each d ∈ drop_list
     do { d.AdvanceDrop(Δt)
          if insideSolid(d)
            then drop_list.delete(d)
            else if d.ShouldCommunicate()
                 then { communicate(d)
                        drop_list.delete(d)
     for each d ∈ injection_list
     do { if d.ShouldCommunicate()
          then { communicate(d)
                 injection_list.delete(d)
          else if (d.injection_time >= currentTime)
               then { drop_list.add(d)
                      injection_list.delete(d)
```

7.2.1 Tropfengenerierung

Idee

Das Partikelsystem soll kleinskalige Features, wie Wassertropfen darstellen. Es muss also eine Heuristik gefunden werden, die in dem Moment Tropfen generiert, wenn in der Natur auch Tropfen entstehen würden. In [61] wird dafür der Spannungstensor aus einem Turbulenz-Modell ausgewertet. Auch wenn der NaSt3DGPF-Löser bei Bedarf ein Turbulenzmodell mitberechnet, soll in dieser Arbeit als Grundlage für eine Heuristik die *Krümmung* der Wasseroberfläche anstatt der Turbulenz verwendet werden.

Die Idee hinter dem Einsatz der Krümmung besteht darin, dass immer dann Wassertropfen generiert werden, wenn die Wasseroberfläche stark gekrümmt ist. Starke Krümmungen entstehen gerade bei Wellenkämmen. Da bei diesen häufig Gischt und Spritzer auftreten, ist die Krümmung ein ideales Merkmal für die Tropfengenerierung.

Neben diesem Argument für die Wahl der Krümmung als Ausgangspunkt für eine Heuristik gibt es noch zwei weitere gute Gründe von der Verwendung eines Turbulenz-Modells abzusehen. Zum einen erfordert die Berechnung der Turbulenz eine erhöhte Rechenzeit. Auf der anderen Seite ist gerade die Level-Set–Methode für eine einfache Berechnung der Krümmung prädestiniert. Die mittlere Krümmung κ der Level-Set–Funktion ergibt sich direkt aus der Divergenz ihres Normalenvektors ([13], Seite 51):

$$\kappa = \nabla \cdot \vec{n}(\vec{x},t) = \nabla \cdot \left(\frac{\nabla \phi(\vec{x},t)}{|\nabla \phi(\vec{x},t)|} \right) \tag{7.15}$$

Zusätzlich zur Krümmung soll auch der Druck an der Tropfenerzeugungs-Position mit in die

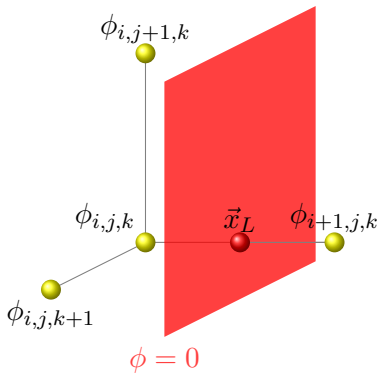


Abb. 7.2: Zellen, die vom Tropfengenerierungs-Verfahren gelesen werden.

Heuristik einbezogen werden. Als natürliche Annahme ergibt sich hier, dass an der Stelle mindestens der Normaldruck von $P = 1000\ Pa$ herrschen soll.

Implementierung

Eine Implementierung der Tropfengenerierung kann in das vorhandene Framework zur Visualisierung der freien Oberfläche integriert werden. Dazu wird die Klasse `Visual` um eine Methode `injectDrops` erweitert. In dieser sucht ein Algorithmus zunächst alle Punkte auf der freien Oberfläche, an denen gemäß der zuvor skizzierten Heuristik Wassertropfen generiert werden sollen.

Um dies zu erreichen werden alle Gitterzellen der diskreten Level-Set–Funktion $\phi_{i,j,k}$ durchlaufen. Für jede Zelle (i, j, k) überprüft der Algorithmus, ob zwischen ihr und einer der in Abbildung 7.2 gezeigten benachbarten Gitterzellen die Level-Set–Isofläche zum Wert Null verläuft. Dies geschieht mit einem Vorzeichenvergleich zwischen den einzelnen Level-Set–Werten. Wird ein Vorzeichenwechsel detektiert, muss die Position $\vec{x}_L$ des Schnittpunktes aus der Level-Set–Isofläche und der entsprechenden Gitterlinie berechnet werden. Hierfür kommt eine lineare Interpolation auf Basis der Level-Set–Werte auf dem Gitter und der Position der Gitterpunkte zum Einsatz (siehe Abschnitt 5.1). Abbildung 7.2 deutet die Position $\vec{x}_L$ des Schnittpunktes an. Die Krümmung κ an dieser Stelle wird aus den Punkten, zwischen denen die Isofläche liegt, linear interpoliert. Da die Krümmung bereits im NaSt3DGPF-Löser berechnet wird, kann hier auf eine existierende Methode `LocalCurvature(i,j,k)` zurückgegriffen werden. Falls die interpolierte Krümmung einen Schwellenwert κ_{thresh} unterschreitet[5], ist die erste Bedingung für die Tropfengenerierung erfüllt.

Für die Überprüfung der Druck-Bedingung, wird auch P an der Position $\vec{x}_L$ linear interpoliert. Ist der Druck mindestens 1000 Pa groß und κ_{thresh} unterschritten, wird ein massebehaftetes Partikel mit einer vorgegebenen Größe erzeugt. Dieses bekommt als initiale Geschwindigkeit

[5] Durch die Wahl der Vorzeichen bei der Level-Set–Funktion haben die Wellenkämme von Wasserwellen eine *negative* Krümmung. Somit muss man hier tatsächlich von einem *Unterschreiten* des Schwellenwertes sprechen.

den Wert im Geschwindigkeitsfeld der Navier-Stokes–Gleichungen an der Stelle $\vec{x}_L$.[6]

Statt der Position $\vec{x}_L$ wird für den Tropfen eine in der Ebene der Level-Set–Isofläche leicht variierte Start-Position verwendet. Wird also zwischen den Werten $\phi_{i,j,k}$ und $\phi_{i+1,j,k}$ ein Vorzeichenwechsel gefunden, verändert man die Startposition in der Y- und Z-Richtung. Die Variation geschieht randomisiert und liegt in der Größenordnung der Maschenweite der Diskretisierung. Als Ergebnis erhält man einen leicht *gejitterten* Anfangspunkt für einen Tropfenflug. Im Mittel führt das zu einer gleichmäßigen Verteilung der zu startenden Tropfen unabhängig von der gerade gewählten Maschenweite.

Bei der praktischen Erprobung des gerade beschriebenen Verfahrens fiel auf, dass die Rate der ausgestoßenen Partikel über die Zeit hinweg unterschiedlich war. Ein solches Verhalten lässt sich durch die adaptive Zeitschrittweiten-Steuerung (siehe Abschnitt 2.2.2) des NaSt3DGPF-Lösers erklären: Ruft man die `injectDrops`-Methode in jedem Zeitschritt auf, werden bei einer adaptiven Reduktion der Zeitschrittweite entsprechend mehr Tropfen pro Zeit erzeugt.

Eine Lösung für diese Schwierigkeit ergibt sich durch den Einsatz einer von der Zeitschrittweite abhängigen Wahrscheinlichkeitsverteilung für den Tropfenstart. Da die Zeitschrittweite Δt für die im Umfeld dieser Arbeit gemachten Simulationen kleiner als $1s$ ist,[7] kann man die Schrittweite als eine Wahrscheinlichkeit interpretieren. Durch die Erzeugung einer gleichverteilten Zufallszahl $randVal \in [0, 1]$ und deren Vergleich mit dem Wert Δt kann in folgender Weise eine Entscheidung für den Tropfenstart getroffen werden:

$$\begin{aligned} randVal \leq \Delta t &\Rightarrow \text{starte Tropfen} \\ randVal > \Delta t &\Rightarrow \text{kein Tropfenstart} \end{aligned}$$

Die Wahrscheinlichkeit für einen Tropfenstart entspricht mit dieser Wahl genau der Zeitschrittweite. Das führt über die Zeit hinweg zu einer im Mittel konstanten Tropfenausstoßrate.

Zusätzliche Flexibilität bringt bei diesem Mechanismus die Modifikation der Wahrscheinlichkeit $prob = \Delta t$. So kann es sinnvoll sein, diese um einen Faktor $injectionProbabilityFactor$ auf $prob = \Delta t \cdot injectionProbabilityFactor$ zu erhöhen. Dadurch wird im Mittel die Anzahl der erzeugten Tropfen größer. Dies kann gerade bei großskaligen Simulationen den visuellen Realismus erhöhen.

Parallelisierung

Die Parallelisierung der Tropfengenerierung gliedert sich in das gewohnte Parallelisierungsschema der Gebietszerlegung ein. Da die Lösung der Navier-Stokes–Gleichungen jeweils auf Teilregionen des Simulationsgebiets erfolgt, wird die Tropfenerzeugung prozessweise auf diesen Regionen durchgeführt. Eine parallele Weiterverarbeitung erfolgt, wie schon beschrieben, über die Mechanismen, die das Partikelsystem anbietet.

[6] Die Geschwindigkeit wird hier ebenfalls interpoliert.
[7] Dies kann auch bei der Konfiguration des Navier-Stokes–Lösers vorgegeben werden.

Algorithmus

Nimmt man die Beschreibung der Implementierung und Parallelisierung zusammen erhält man den Algorithmus 7.2.1 zur vollautomatischen Erzeugung von Tropfen, der auf Wunsch in jedem Zeitschritt im Navier-Stokes–Löser aufgerufen werden kann.

Algorithmus 7.2.1: INJECTDROPS( )

```
for each process
do { for each φ_{i,j,k} ∈ localProcessDomain
     do { prob = Δt · injectionProbabilityFactor
          for each neighbour ∈ {(i+1,j,k),(i,j+1,k),(i,j,k+1)}
            do if signChanged(φ,(i,j,k),neighbour)
               then { calculate x⃗_L
                      κ = interpolateLocalCurvature(x⃗_L)
                      P = interpolatePressure(x⃗_L)
                      if ((κ < κ_thresh) and (P ≥ 1000))
                        then { randVal = uniformDistributedValue_between(0,1)
                               if (randVal < prob)
                                 then { x⃗_start = jitter(x⃗_L)
                                        if
                                          then { u⃗_start = interpolateVelocity(x⃗_L)
                                                 startDrop(x⃗_start, u⃗_start, globalDropRadius) } } } } }
```

7.2.2 Tropfeneinschlag

Die Rückkopplung bei der hier vorgestellten Zwei-Wege-Kopplung zwischen der Level-Set–Methode und dem Partikelsystem besteht in der Simulation des Tropfeneinschlages bei Partikeln, die auf die freie Oberfläche treffen. Im Folgenden soll ein einfaches und sehr effizientes Modell zur Nachbildung eines realen Tropfeneinschlags entwickelt werden.

Modell

Der Tropfen-Einschlag soll durch eine Kraftübertragung des Tropfens auf die Nullkontur der Level-Set–Funktion modelliert werden. Man betrachte hierzu noch einmal die Navier-Stokes–Gleichungen:

$$\frac{\partial}{\partial t}\vec{u} + (\vec{u}\cdot\nabla)\vec{u} + \frac{1}{\varrho_\infty}\nabla p = \frac{\mu}{\varrho_\infty}\Delta\vec{u} + \vec{g} \tag{7.16}$$

$$\nabla\cdot\vec{u} = 0 \tag{7.17}$$

Auf der rechten Seite der Impulsgleichung kommt als sogenannte *Volumenkraft* die Schwerkraft $\vec{g}$ vor. Mit Volumenkraft wird dabei eine Kraft bezeichnet, die in jedem Punkt der Simulationsumgebung wirkt und bei der es sich formal um eine *Kraftdichte* handelt. Bei Kraftdichten

wird eine Kraft pro Masse des Einheitsvolumens betrachtet ([45], Seite 12). Folglich ist die Einheit von $\vec{g}$ nicht Newton ($[N]$), sondern $[\frac{N}{kg}] = [\frac{m}{s^2}]$ und somit die Einheit der Beschleunigung. Dementsprechend ist es auch ganz natürlich die Erd*beschleunigung* in $\vec{g}$ einzusetzen. Möchte man andere Kräfte als die Erdbeschleunigung wirken lassen, müssen diese zunächst formal durch die Masse des Einheitsvolumens, also durch ein Kilogramm geteilt werden.[8]

Man kann die Impulsgleichung verallgemeinern:

$$\frac{\partial}{\partial t}\vec{u} + (\vec{u} \cdot \nabla)\vec{u} + \frac{1}{\varrho_\infty}\nabla p = \frac{\mu}{\varrho_\infty}\Delta\vec{u} + \vec{a} \tag{7.18}$$

In dieser Darstellung kommt nun eine allgemeine Volumenkraft $\vec{a}$ vor. Es ist also möglich, dass zusätzliche Kräfte, die auf das Volumen wirken sollen, durch eine geeignete Modifikation der Kraft $\vec{a}$ in die Simulation eingebracht werden.

Genau dies ist der Ansatz, über den der Tropfeneinschlag in die Level-Set–Nullebene modelliert wird. Dazu definiert man ein $\vec{a}$ wie folgt:

$$\vec{a}(\vec{x}, t) := \vec{g} + \Delta\vec{a}(\vec{x}, t) \tag{7.19}$$

Bei dieser Definition fällt auf, dass $\vec{a}$ nun zeit- und ortsabhängig ist. Insgesamt setzt sich damit die Volumenkraft aus der konstanten Schwerkraft und einem variablen Kraftterm $\Delta\vec{a}$ zusammen. In diesen können Zusatzkräfte einfließen. Bei dem speziellen Fall der Kraft eines Tropfeneinschlages wirkt die Zusatzkraft $\Delta\vec{a}$ lokal an der Position des Tropfens.

Der Tropfeneinschlag wird nun als eine Art deformierender Aufprall modelliert. Die Idee besteht darin, dass der Tropfen die Wasseroberfläche beim Auftreffen verformt und dabei auf die Geschwindigkeit Null abgebremst wird. Bei diesem Abbremsprozess entsteht eine Kraft $\vec{F}_{hit}$, die 1:1 auf die Oberfläche übertragen wird.

Durch die Level-Set–Methode wird die Wasseroberfläche als implizite Fläche in einem Volumen definiert. Soll eine Kraft auf die Fläche übertragen werden geschieht dies durch eine Volumenkraft, die indirekt die Fläche transportiert. Hier kommt der Kraftterm $\Delta\vec{a}$ ins Spiel. Über diesen wird bei einem Aufprall lokal die umgekehrte Abbremskraft in die Volumenkraft mit eingebracht. Als Resultat verformt sich die Nullkontur der Level-Set–Funktion an der Aufprallstelle in der gewünschten Weise.

Eine Berechnung der Abbremskraft $\vec{F}_{hit}$ ist außerordentlich schwierig. Das Studium von Literatur zum Thema Tropfenaufprall [32] zeigt, dass das Phänomen eines Tropfenaufpralls außerordentlich komplex ist und von sehr vielen Faktoren abhängt. Ein Weg, dies in physikalisch einwandfreier Art zu simulieren läge in der Verwendung der Navier-Stokes–Gleichungen. Jeder Tropfen müsste über die Level-Set–Funktion modelliert und dann entsprechend gerechnet werden. Gerade dies soll aber bei der Kopplung mit Partikeln verhindert werden, da bei der vollständigen Simulation ein zu hoher Rechenaufwand entsteht.

Als einziger Ausweg bleibt eine grobe Modellierung der Abbremskraft: Nach Newton ergibt sich eine Kraft $\vec{F}$ aus der Masse m und der Beschleunigung $\vec{a}$, während die Beschleunigung die

[8] Diese Division führt nur zu einer Veränderung der Einheit, weshalb im Folgenden zur einfacheren Sprechweise stets von einer Kraft und nicht von der formal korrekten Kraftdichte gesprochen wird.

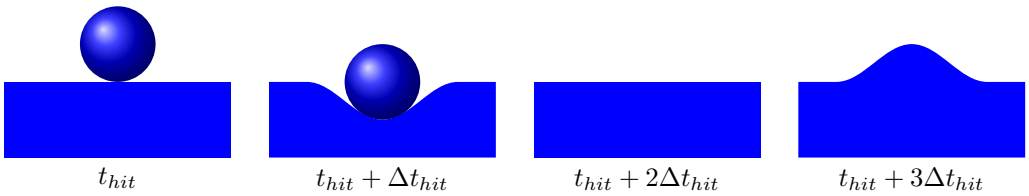


Abb. 7.3: Darstellung des Modells zur Berechnung der Eintauchzeit.

Ableitung der Geschwindigkeit ist (siehe Abschnitt 7.1.1). Damit ergibt sich

$$\vec{F}(t) = m\,\vec{a}(t) = m\,\frac{\partial \vec{u}}{\partial t}(t)\,. \tag{7.20}$$

Da es hier um die Partikel aus den vorangegangenen Abschnitten geht, soll natürlich deren Geschwindigkeit und Masse betrachtet werden,

$$\vec{F}(t) = m_P\,\vec{a}_P(t) = m_P\,\frac{\partial \vec{u}_P}{\partial t}(t)\,. \tag{7.21}$$

Eine einfache Diskretisierung mit einer Vorwärtsdifferenz liefert

$$\vec{F}(t) \approx m_P\,\frac{\vec{u}_P(t+\Delta t) - \vec{u}_P(t)}{\Delta t}\,. \tag{7.22}$$

Gesucht ist nun die Kraft $\vec{F}_{hit}$ die beim Abbremsen zum Zeitpunkt t_{hit} des Aufpralls auftritt,

$$\vec{F}_{hit} = \vec{F}(t_{hit}) \approx m_P\,\frac{\vec{u}_P(t_{hit}+\Delta t) - \vec{u}_P(t_{hit})}{\Delta t}\,. \tag{7.23}$$

Der Zeitpunkt t_{hit} kann dabei als bekannt vorausgesetzt werden und $\vec{u}_P(t_{hit})$ lässt sich bestimmen. Angenommen, man wüsste wie lange das Eintauchen eines Tropfens braucht, dann könnte man die Kraft berechnen. Sei dazu die Eintauchdauer Δt_{hit} gegeben. Dann ist gemäß der Modellannahme die Geschwindigkeit des Tropfens bei Null, wenn dieser vollständig eingetaucht ist. Es gilt also

$$\vec{u}_P(t_{hit} + \Delta t_{hit}) = 0. \tag{7.24}$$

Damit erhält man als Näherung für $\vec{F}_{hit}$

$$\vec{F}_{hit} \approx m_P \frac{-\vec{u}_P(t_{hit})}{\Delta t_{hit}}. \tag{7.25}$$

Als Letztes muss Δt_{hit} modelliert werden. Man betrachte Abbildung 7.3. Diese stellt den Eintauchprozess eines Tropfens dar. Es wird angenommen, dass durch das Eintauchen eine Welle erzeugt wird. Zum Zeitpunkt $t_{hit} + \Delta t_{hit}$ hat die Welle ihre maximale negative Auslenkung. Bei $t = t_{hit} + 2\Delta t_{hit}$ ist die Welle wieder ausgeglichen und bei $t = t_{hit} + 3\Delta t_{hit}$ erhält die Welle ihre maximale positive Auslenkung. Die so erzeugte Welle breitet sich im Raum aus. Für die Periodendauer T dieser Welle gilt offensichtlich $T = 4\Delta t_{hit}$.

Ganz allgemein gilt bei Wellen, dass sich deren *Geschwindigkeit* v aus der *Frequenz* f und der *Wellenlänge* λ zusammensetzt,

$$v = \lambda f \,. \tag{7.26}$$

Die Frequenz ist die Inverse der Periodendauer. Also gilt in diesem Fall

$$f = \frac{1}{4\,\Delta t_{hit}}. \tag{7.27}$$

In dem hier vorgestellten Modell wird für die Wellenlänge der Durchmesser des Wassertropfens angenommen,

$$\lambda = 2\, r_P \,. \tag{7.28}$$

Dies entspricht in etwa dem Durchmesser der Deformation der Wasseroberfläche. In [64] wird eine Approximation v_t für die Wellengeschwindigkeit v für sogenannte *Tiefwasserwellen* angegeben. Tiefwasserwellen sind demnach Wellen auf Gewässern, deren Tiefe mindestens bei der halben Wellenlänge liegt. Übertragen auf dieses Modell bedeutet das, dass ein zu simulierendes Wasser mindestens den Tropfenradius als Tiefe haben muss. Eine solche Forderung ist sicherlich immer erfüllt. Die nach [64] angenäherte Geschwindigkeit v_t lautet

$$v_t \approx \sqrt{\frac{g\,\lambda}{2\,\pi}} \,. \tag{7.29}$$

g ist wie immer die Erdbeschleunigung. Mit v_t und Gleichung (7.26) lässt sich eine Näherung für die Periodendauer T angeben,

$$T = 4\Delta t_{hit} \approx \sqrt{\frac{2\,\pi\,\lambda}{g}} \,. \tag{7.30}$$

Das Einsetzen der Wellenlänge und der Erdbeschleunigung und ein Umstellen von Gleichung (7.30) liefert schließlich eine Näherung für die Eintauchdauer,

$$\Delta t_{hit} \approx \frac{1}{2}\sqrt{\frac{r_P\,\pi}{9.81}} \,. \tag{7.31}$$

Hiermit ist die gesuchte Kraft $\vec{F}_{hit}$ gefunden,

$$\vec{F}_{hit} = -2\, m_P\, \vec{u}_P(t_{hit}) \sqrt{\frac{9.81}{r_P\,\pi}} \,. \tag{7.32}$$

Diese wirkt ab dem Zeitpunkt t_{hit} für eine Dauer von Δt_{hit}. Sie fließt über den Zusatz-Kraftterm $\Delta\vec{a}$ in die Navier-Stokes–Gleichungen ein und soll nur lokal im Umfeld des Tropfeneinschlags vorhanden sein. Näheres hierzu folgt in der Beschreibung der Implementierung.

Implementierung

Zur Verwaltung der Tropfeneinschläge wird der Navier-Stokes–Löser und insbesondere das Partikelsystem `DropSystem` um eine Liste `impact_list` erweitert. Falls ein Tropfen vom zweiten

Variablen	Typ	Einheit	Beschreibung
`pos`	`double[3]`	$[m]$	Position des Tropfeneinschlags
`force`	`double[3]`	$[N]$	auftretende Kraft
`time`	`double`	$[s]$	Zeitpunkt des Tropfeneinschlags
`radius`	`double`	$[m]$	Tropfenradius

Tabelle 7.2: Beschreibung der wesentlichen Variablen eines Eintrages der `impact_list`.

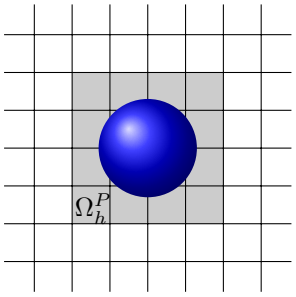


Abb. 7.4: Darstellung des Gebietes Ω_h^P, das den Tropfen beim Einschlag umhüllt.

zum dritten Zustand (siehe Abbildung 7.1, Seite 99) über geht, also in die Wasseroberfläche eintritt, wird ein Eintrag mit den in Tabelle 7.2 angegebenen Daten in die Liste eingefügt.

Die auftretende Kraft bestimmt das Programm im Moment des Einschlags auf Basis der aktuellen Geschwindigkeit des Tropfens. Sie wird über die neue Methode `GetForce` der Klasse `Drop` berechnet. Dabei kommt das Modell für die Kraft aus dem letzten Abschnitt zum Einsatz.

Um den Krafteintrag $\Delta\vec{a}$ in die Rechnung des NaSt3DGPF-Lösers zu integrieren, wird eine Methode `ModifyFGHOnImpact` entwickelt. Diese durchläuft die Liste der Tropfeneinschläge. Für jeden Eintrag in der Liste, dessen Tropfeneinschlagszeitpunkt nicht länger her ist, als das jeweilige Δt_{hit}, modifiziert sie das globale Volumenkräfte-Feld $\vec{a}$. Da das Feld in jedem Zeitschritt komplett neu aufgebaut wird, muss nur ein entsprechender Kraft-Wert additiv hinzugefügt werden.

Im vorangegangenen Abschnitt wurde noch nicht der genaue Wirkungsbereich der Kraft $\vec{F}_{hit}$ angegeben. Dies soll hier nachgeholt werden. Der Grundgedanke liegt darin, die Kraft genau in den Zellen aufzuaddieren, die beim Tropfeneinschlag vom Tropfen berührt werden. Als erste grobe Näherung kann man mit einem Vielfachen des Radius des Tropfens ein würfelförmiges, diskretes Gebiet Ω_h^P bestimmen, in dem sich potentiell zu modifizierende Zellen befinden. Eine zweidimensionale Darstellung des Gebietes findet sich in Abbildung 7.4.

Durch die Vorauswahl dieses Gebietes wird der nachfolgende Schritt zur Berechnung der Kräfte für die einzelnen Zellen optimiert. Bei diesem wird für jede Zelle im Gebiet Ω_h^P die Entfernung zum Tropfenmittelpunkt $\vec{x}_{hit}$ berechnet. Auf Basis dieses Abstands d wird dann über die Funktion $w(d)$ ein zum Rand des Tropfens hin geglätteter Krafteintrag für jede Zelle

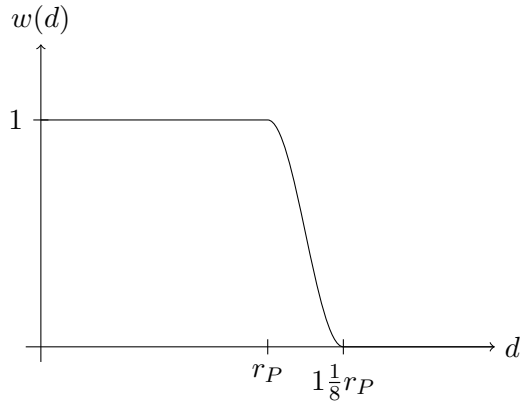


Abb. 7.5: Plot der Gewichtsfunktion $w(d)$.

gefunden. Die Funktion $w(d)$ ist wie folgt definiert:

$$w(d) = \begin{cases} 1 & 0 \leq d \leq r_P \\ \frac{1}{2}\cos(8\pi \frac{d-r_P}{r_P}) + \frac{1}{2} & r_P < d \leq \frac{9}{8} r_P \\ 0 & sonst \end{cases} \tag{7.33}$$

Ein Plot dieser Funktion (in Abbildung 7.5) gibt Aufschluss über ihre Beschaffenheit. Wie man sieht ist die Gewichtsfunktion $w(d)$ auf dem Intervall $[0, r_P]$ konstant auf dem Wert Eins. Der Wert r_P ist der Radius des gerade einschlagenden Tropfens. Ein geglätteter Abfall auf Null findet im Intervall $[r_P, \frac{9}{8} r_P]$ statt. Der Krafteintrag $\vec{F}_{hit}(\vec{x}_c)$ für eine Zelle c in Ω_h^P mit der Weltkoordinate $\vec{x}_c$ berechnet sich damit bezüglich $\vec{x}_{hit}$ zu

$$\vec{F}_{hit}(\vec{x}_c) = w(|\vec{x}_c - \vec{x}_{hit}|)\, \vec{F}_{hit}. \tag{7.34}$$

Die Idee hinter dem geglätteten Abfall der Gewichtsfunktion besteht darin, dass nicht nur die Zellen, die im Tropfen enthalten sind eine Kraft erhalten, sondern diese Kraft auch ein wenig in die Umgebung abstrahlt. Das führt zu optisch und numerisch besseren Ergebnissen. Gleichung (7.34) muss man so verstehen, dass der Wert $\vec{F}_{hit}$ auf der rechten Seite die bei dem Aufprall berechnete Kraft aus dem Modell im letzten Abschnitt ist.

Da die Kraft $\vec{F}_{hit}$ eigentlich eine Bremskraft ist und damit in die zum Tropfenfall entgegengesetzte Richtung wirken müsste, sollte man annehmen, dass an dieser Stelle das Vorzeichen umgedreht werden muss. Damit würde die Kraft zu einer Schubkraft für die freie Oberfläche. Durch die typische Wahl im NaSt3DGPF-Löser, die Gravitation in negativer Richtung verlaufen zu lassen, ist aber die bereits berechnete Wirkrichtung der Kraft $\vec{F}_{hit}$ korrekt und führt zum erwünschten Schub.

In der Praxis hat sich gezeigt, dass die Gleichung wie folgt modifiziert werden sollte:

$$\vec{F}_{hit}(\vec{x}_c) = w(|\vec{x}_c - \vec{x}_{hit}|)\, w_{grid}\, w_{opt}\, \vec{F}_{hit}. \tag{7.35}$$

Der Faktor w_{grid} dient der korrekten Behandlung von Tropfeneinschlägen in großskaligen Si-

mulationen. Ist ein Tropfen kleiner als eine Diskretisierungszelle, sollte die Kraft auch nur entsprechend geringer wirken. Deshalb enthält w_{grid} den Anteil des Volumens eines Tropfens an dem Gesamtvolumen der Zelle, in die der Tropfen mit dem Mittelpunkt einschlägt. Ist das Volumen des Tropfens größer als das der Zelle, wird der Wert auf Eins normalisiert. w_{opt} dient der zusätzlichen manuellen Kontrolle der Kraft beim Tropfeneinschlag und kann je nach Bedarf aus visuellen Überlegungen heraus mit einem Wert ≥ 1 belegt werden.

Das gesamte Verfahren der Tropfeneinschlags-Berechnung lässt sich im Löser ein- und abschalten. Somit kann je nach Bedarf die Rückkopplung des Partikelsystems zur Level-Set–Funktion erfolgen.

Parallelisierung

Es ist zu beachten, dass auch der Tropfeneinschlag parallelisiert werden muss um den gesamten Löser für die Navier-Stokes–Gleichungen parallel zu erhalten. Wichtig ist in diesem Zusammenhang, dass die Liste der Tropfeneinschläge (`impact_list`) an alle Prozesse verschickt wird. Nur damit wird die Modifikation des Kräftefeldes für die Volumenkräfte auch über die Gebietsgrenzen der Gebietszerlegung hinweg möglich. Die Kommunikation der Daten erfolgt in gewohnter Weise.

Algorithmus

Aus den obigen Festlegungen ergibt sich der in Pseudocode formulierte Algorithmus 7.2.2 für die Modifikation des Kräftefeldes F zum Zeitpunkt t.

Algorithmus 7.2.2: ModifyFGHOnImpact(F, t)

for each *process*
do
  for each $I \in impact\_list$
  do
    if $(t \leq I.time + \Delta t_{hit}(I))$
    then
      calculate Ω_h^P
      calculate w_{grid}
      for each $\vec{x}_c \in \Omega_h^P$
      do if $\vec{x}_c \in localProcessDomain$
      then
        $\vec{F}_{hit}(\vec{x}_c) = w(|\vec{x}_c - I.pos|)\, w_{grid}\, w_{opt}\, I.force$
        $\vec{F}(\vec{x}_c) = \vec{F}(\vec{x}_c) + \vec{F}_{hit}(\vec{x}_c)$

7.3 Anbindung an Maya

Für die Animation der Wassertropfen soll erneut die Animationssoftware Autodesk Maya herangezogen werden. Wie bei der Visualisierung der freien Oberfläche muss auch für die massebehafteten Partikel ein Plugin implementiert werden, das automatisiert Partikelinformationen

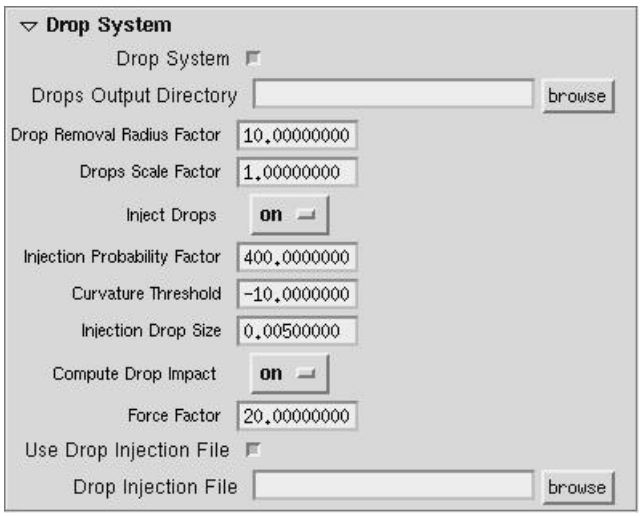


Abb. 7.6: Screenshot der Parameter zum Partikelsystem im Attribute Editor der `SimulationArea`.

lädt und diese in Geometrien in Maya umwandelt. Die Details dieser Implementierung sollen in diesem Abschnitt vorgestellt werden.

Vor der Berechnung und Animation der Wassertropfen steht allerdings noch die Konfiguration des NaSt3DGPF-Lösers mittels Maya: Auch für die Wassertropfen soll es möglich sein, alle wesentlichen Daten über das `SimulationArea`-Plugin vorzugeben. Aus diesen Daten wird dann vollautomatisch eine Konfigurationsdatei für den Strömungslöser erstellt. Die benötigten Erweiterungen für die Konfiguration sollen hier beschrieben werden.

7.3.1 Konfiguration

Zur Konfiguration des Partikelsystems und der Zwei-Wege-Kopplung wird der `SimulationArea`-Knoten um einige Attribute erweitert. Auf eine genaue Vorstellung aller Attribute mit entsprechenden Standardwerten und Typen soll hier verzichtet werden. Statt dessen findet sich in Abbildung 7.6 ein Screenshot des Attribute Editors zum `SimulationArea`-Plugin. Um genau zu sein wird in dem Bild nur der Teil des Editors gezeigt, der die neuen Attribute für die Tropfendarstellung enthält. Wie man sehen kann, kann sowohl die Tropfengenerierung (*Inject Drops*) als auch die Rückkopplung (*Compute Drop Impact*) beliebig an- und abgeschaltet werden. Ebenso lässt sich ein Zielverzeichnis für die Datenausgabe des Partikelsystems vorgeben. *Curvature Threshold* steht hier für den Wert κ_{thresh}. Über den Parameter *Drop Injection File* kann eine Datei vorgegeben werden, die Startdaten für einzelne Tropfen enthält.

Neben der Aufnahme der neuen Attribute in den `locator`-Knoten `SimulationArea` und in dessen Attribute Editor, muss auch das Kommando `createNavFile` erweitert werden. Dieses wurde bereits im Kapitel zur Erstellung der Szenenbeschreibung für den Strömungslöser (Kapitel 4) beschrieben. Die Erweiterung besteht darin, die in Abbildung 7.6 gezeigten Attribute als Parameter in die Konfigurationsdatei zu schreiben. Für eine detaillierte Auflistung aller im Rahmen dieser Arbeit eingeführten neuen Parameter für die Konfiguration des Strömungslö-

sers sei an dieser Stelle erneut auf Anhang A verwiesen. Da die Umsetzung der Erweiterung des `createNavFile`-Kommandos um weitere Parameter offensichtlich ist, soll an dieser Stelle darauf nicht weiter eingegangen werden.

7.3.2 Datenaustausch

Die Übergabe der berechneten Partikel vom NaSt3DGPF-Löser an Maya erfolgt, wie auch bei der freien Oberfläche, durch Dateien. Hierzu wird die Klasse `DropSystem` zur Verwaltung des Partikelsystems um eine Methode `WriteToFile` erweitert. Diese durchläuft die `drop_list` und schreibt sämtliche Tropfen-Positionen und -Radien in eine Datei. Aufgerufen wird `WriteToFile` aus der Methode `WriteMesh` der Klasse `NavierCalc`. `WriteMesh` implementiert bereits das Herausschreiben der freien Oberfläche (siehe Abschnitt 5.3.7). Entsprechend wird bei Bedarf pro geschriebener Oberfläche im selben Zeitschritt auch eine Datei mit allen im Partikelsystem bekannten Tropfen generiert.

Eine Parallelisierung kann, wie auch bei den freien Oberflächen ganz natürlich geschehen. Jeder Prozess erzeugt seine eigene Datei, die die Tropfen des jeweiligen Gebiets der Gebietszerlegung enthält. Durch diese Vorgehensweise wird jegliche zeitaufwendige Kommunikation für die Generierung einer gemeinsamen Datei für alle Gebiete vermieden. Das kommt der Laufzeit zugute.

7.3.3 Darstellung der Wassertropfen

Zur Visualisierung der Tropfen wird das in Maya mitgelieferte *nParticle*-Partikelsystem verwendet. Grundsätzlich sollen mit diesem System ganz eigenständig diverse physikalische Effekte wie zum Beispiel das Verhalten von Stoffen simuliert werden. Da die Wasserspritzer aber vollständig durch den Strömungslöser berechnet werden, sollen für ihre Darstellung nur die *Rendering*-Fähigkeiten des *nParticle*-Systems verwendet werden. Die Grundidee besteht also darin, die Wassertropfen als Partikel in das Partikelsystem zu emittieren und dann mit den sehr guten Rendering-Fähigkeiten von Maya darzustellen. Eine Vorgehensweise hierfür soll im Folgenden erläutert werden.

Zur Übernahme der Partikel in Maya wird wie bei dem Import der freien Oberflächen ein neuer *Dependency Node* implementiert und als Plugin eingebunden. Der Knoten soll in diesem Fall `SimulationWaterDropsNode` heißen. Das Plugin für Maya trägt den selben Namen. Der zu implementierende Knoten ähnelt dem Knoten `SimulationMeshNode`. Die Attribute des neuen Knotens werden in Tabelle 7.3 auf Seite 113 beschrieben.

Um die genaue Funktion des `SimulationWaterDropsNode` zu verstehen, sollte zunächst erklärt werden, wie dieser mit anderen Knoten zusammen in den Dependency Graph integriert wird: Sei dazu zunächst ein Knoten vom Typ `SimulationArea` gegeben. Wenn der `SimulationWaterDropsNode` mit Hilfe des Kommandos `createWaterDropsNode`[9] erzeugt wird, wird gleichzeitig ein *nParticle*-System erstellt. Eine Darstellung der zum System gehörenden Knoten findet sich in Abbildung 7.7 auf Seite 113. Von den dort gezeigten Knoten ist vor allem der des Typs `nParticle` (mit dem Namen `nParticleShape1`) von Interesse, da dieser auch die Visualisierung der Partikel übernimmt.

[9] Dieses Kommando wird ebenfalls in das Plugin integriert.

Attribut	Typ	Beschreibung
`time`	kTime	aktuelle Zeit in der Szene
`dropsPath`	kString	Pfad zu den Ausgabedateien der Wassertropfen
`nParticleShapeName`	kString	Name des nParticleShape-Knotens
`startFrame`	kInt	Nummer des ersten Frames, für das das Partikelsystem arbeitet

Tabelle 7.3: Beschreibung der Attribute des `SimulationWaterDropsNode`.

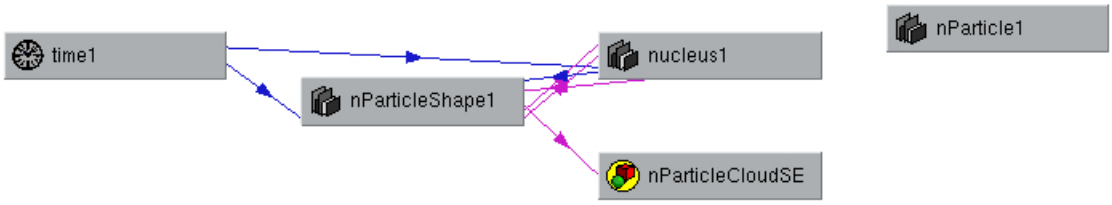


Abb. 7.7: Darstellung aller mit dem *nParticle*-System zusammenhängenden Knoten.

Zur Anbindung des `SimulationWaterDropsNode` an `nParticleShape1` wird im `createWaterDropsNode`-Kommando eine Verknüpfung zwischen den `startFrame`-Attributen der beiden Knoten hergestellt. Außerdem verbindet man das Attribut `outTime` des `time`-Knotens mit dem `time`-Attribut des `SimulationWaterDropsNode`. Schließlich wird das `dropsOutputDirectory`-Attribut der `SimulationArea` an das `dropsPath`-Attribut des `SimulationWaterDropsNode` angeschlossen. Führt man alle diese Verknüpfungen aus, so erhält man die in Abbildung 7.8 dargestellten Verbindungen im Dependency Graph.

Im Befehl `createWaterDropsNode` werden auch noch weitere Einstellungen an den einzelnen Knoten vorgenommen. Auf diese soll aber nicht weiter eingegangen werden, da sie für das grundlegende Verständnis der Visualisierung der Tropfen nicht wichtig sind. Vielmehr kann man nun unter Berücksichtigung der gerade vorgestellten Verknüpfungen die Funktionsweise des Knotens zum Einbinden der Wassertropfen (`SimulationWaterDropsNode`) beschreiben.

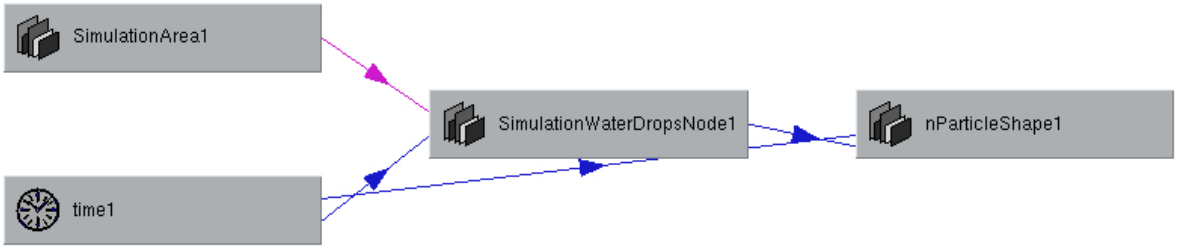


Abb. 7.8: Die Verknüpfungen des `SimulationWaterDropsNode` nach Ausführung des `createWaterDropsNode`-Kommandos.

Die Funktionalität des Knotens besteht darin, dass für ein sich änderndes `time`-Attribut Dateien mit den entsprechenden Partikelpositionen geladen werden. Als Pfad für die Dateien wird der Inhalt des `dropsPath`-Attributs genommen. Das korrekte Mapping von Zeit zu Dateinamen und die Weiterverarbeitung der parallel herausgeschriebenen Dateien wurde schon beim `SimulationMeshNode` erklärt, weshalb dies hier nicht wiederholt wird. Eingelesene Tropfenpositionen liegen während der Verarbeitung als `MPointArray` vor. Zur Darstellung werden die Tropfen in das Partikelsystem emittiert. Das geschieht über die `emit`-Methode des zum Partikelsystem zugehörigen Function Set.

Normalerweise würden sich die ausgesendeten Partikel gemäß einer von Maya vorgegebenen Physik in den Frames nach der Emission weiterbewegen. Da aber im `createWaterDropsNode`-Kommando das Partikelsystem dahingehend konfiguriert wird, dass die Tropfen nur in dem Frame existieren, in dem sie erzeugt werden, sind zu jedem Zeitpunkt im System nur die gerade eingelesenen Partikel vorhanden. Durch diese Vorgehensweise lässt sich eine direkte Visualisierung der Wassertropfen ermöglichen, ohne dass die normalerweise verwendete Physik-Engine von Maya darauf Einfluss hat.

Ist einmal die Verknüpfung der Tropfendaten mit dem Partikelsystem durchgeführt, können die Standard-Renderingverfahren für Partikel in Maya verwendet werden. Zwei mögliche Darstellungsverfahren sind das Rendern über eine Geometrie und das Rendern der Partikel als *Cloud*. Beim Rendern mittels Geometrie werden automatisiert an den Partikelpositionen Sphären aus Polygonen erzeugt. Diese können zum Beispiel mit einem *Dielectric Shader* versehen werden, der dann ein wasserartiges Material mit Brechungen an der Oberfläche und leichten volumetrischen Effekten erzeugt. Beim Rendering-Typ *Cloud* erinnert das gerenderte Partikel an eine kleine runde Wolke.

Auf die genaue Vorgehensweise zum Rendern von Partikeln soll hier nicht weiter eingegangen werden, da sie entweder dem mit Autodesk Maya vertrauten Leser bereits bekannt ist oder leicht in der entsprechenden Literatur [52] nachgelesen werden kann.

Setzt man das implementierte Plugin `SimulationWaterDropsNode` in der Praxis ein, lässt es sich sehr einfach bedienen. Ist schon eine `SimulationArea` vorhanden, muss nur noch über den Menüpunkt *Simulation* → *Create Water Drops Node* das entsprechend umgesetzte Kommando aufgerufen werden. Das Laden und Visualisieren der Partikel geschieht dann vollautomatisch.

8 Ergebnisse

Im nun folgenden Kapitel werden die Ergebnisse von verschiedenen Simulationsaufgaben und Testproblemen vorgestellt. Mit Hilfe der Testprobleme sollen ganz speziell einige Verfahren, die im Rahmen dieser Arbeit implementiert werden auf ihre Performance und die visuellen Resultate hin analysiert werden. Zudem sollen einige eher allgemein gehaltene Simulationsaufgaben die Praxistauglichkeit des implementierten Gesamtsystems zeigen.

Bei den Performance-Analysen kommen verschiedene Rechnersysteme zum Einsatz, die im ersten Abschnitt dieses Kapitels beschrieben werden. Alle angegebenen Laufzeiten sind *Wall Clock*-Zeiten. Das bedeutet, dass die Zeitangaben der tatsächlich vergangenen Zeit vom Anfang bis zum Ende eines jeweiligen Testlaufes entsprechen.

Zur Visualisierung beziehungsweise Animation der Simulationsresultate sind der elektronischen Ausgabe der vorliegenden Arbeit gerenderte Videos beigefügt. Werden diese im Fließtext erwähnt befindet sich stets das Symbol  am Rand des Textblocks. Hierüber können die Videos per Klick gestartet werden.[1] In den Beschriftungen zu Abbildungen, für die Videos verfügbar sind, findet sich ebenfalls das Verknüpfungssymbol.

Nach der Übersicht über die für die Simulation und das Rendering eingesetzten Rechnersysteme folgt im zweiten Abschnitt des vorliegenden Kapitels die Vorstellung einer Fluidsimulation in einem Canyon. Mit dieser soll gezeigt werden, dass das implementierte Konfigurationssystem gut funktioniert. Außerdem werden Performance-Tests bezüglich der Voxelisierungsalgorithmen durchgeführt. Im Anschluss daran wird der implementierte Marching-Cubes–Algorithmus mit einem Testproblem auf sein Laufzeitverhalten hin analysiert. Die Abschnitte 8.4 und 8.5 präsentieren Testsimulationen für die in dieser Arbeit implementierte Zwei-Wege-Kopplung der Level-Set–Funktion mit einem Partikelsystem.

Zwei weitere Simulationen dienen der Demonstration der besonderen Anwendungsmöglichkeiten eines zweiphasigen Navier-Stokes–Lösers. Es wird ein Tropfen simuliert, der in ein Wasserbecken einschlägt und einen Rayleigh-Jet erzeugt. Die zweite Simulation zeigt eine große Luftblase, die in einem Wasserbecken aufsteigt.

An die beiden Simulationen schließen sich drei ingenieurswissenschaftlich motivierte Testfälle an. Es werden ein fahrendes Boot, ein Staudamm aus dem Wasser abgelassen wird und ein Fluss in einem steinigen Flussbett simuliert.

Abgeschlossen wird das Kapitel durch drei Simulationen, die auf die Computeranimation ausgerichtet sind. Die erste zeigt Wasser, das turbulent in einer Gasse strömt. Als nächstes wird in einer sehr großskaligen Simulation ein Fluss nachgebildet, der unter der Londoner Tower Bridge her fließt. Schließlich wird eine Welle simuliert, die sich an einem Leuchtturm bricht und dabei Gischt erzeugt.

[1] Sofern die benötigten Codecs installiert sind, sollte dies sowohl unter Windows als auch unter Linux über den Adobe Reader funktionieren.

8.1 Test-Systeme

Für die Berechnungen und Visualisierungen, die in diesem Kapitel beschrieben werden, kommt zum einen ein Cluster-System und zum anderen eine Graphik-Workstation zum Einsatz. Beide sollen zusammen mit der jeweiligen Software-Ausstattung nachfolgend vorgestellt werden.

8.1.1 Die Graphik-Workstation

Als Graphik-Workstation wird eine Dell Precision 690n verwendet. Diese ist mit zwei Quad Core Intel Xeon E5345 Prozessoren mit 2,33 GHz pro Kern ausgestattet. Zudem sind 16 GB Arbeitsspeicher und eine nVidia Quadro FX3500 - Graphikkarte mit 256 MB Speicher verbaut. Zur Datenspeicherung kommt eine SATA 2–Festplatte mit 7.200 Umdrehungen pro Minute zum Einsatz.

Das Betriebssystem ist Linux. Es wird die Ubuntu-Distribution in der Version 8.04 als 64-Bit-Installation verwendet. Auf der Workstation ist eine 64-Bit - Version von Autodesk Maya 2009 installiert. Für die Übersetzung der Plugins wird der GNU C-Compiler in der Version 4.1.2 benutzt.

Alle Arbeiten in Maya sowie alle Renderings aus diesem Kapitel werden auf der gerade beschriebenen Workstation durchgeführt.

8.1.2 Das Cluster-System

Die Simulationen werden auf dem Cluster-System *Himalaya* des Instituts für Numerische Simulation der Universität Bonn durchgeführt. In seiner ersten Ausbaustufe war der Cluster im Juni 2005 unter den Top 500 der weltweit schnellsten Rechnersysteme [68]. Zum Zeitpunkt der Entstehung dieser Arbeit besteht das System [42] aus 128 Rechnern (bzw. *Knoten*) des Typs Dell PowerEdge 1850. Diese sind jeweils mit zwei Intel Xeon EM 64 T Prozessoren mit je 3.2 GHz ausgestattet. Auf jedem Knoten sind zwischen 4 und 6 GB Arbeitsspeicher verbaut, so dass eine verteilte Arbeitsspeicherkapazität von 640 GB zur Verfügung steht. Zur Kommunikation wird ein Myrinet XP–Netz eingesetzt.

Angesteuert werden die Knoten über ein Front-End–System [41]. Dieses wird durch eine Dell PowerEdge 2900 III realisiert. Das System stellt dem gesamten Cluster ungefähr 27 Terrabyte an Plattenspeicher über ein NFS-Dateisystem zur Verfügung.

Die Knoten und auch das Front-End–System werden mit einem Ubuntu-Linux in der Version 8.04 (64 Bit) betrieben. Softwareseitig findet die parallele Kommunikation über das *Message Passing Interface* (MPI) statt. Insbesondere kommt hier MPICH-MX zum Einsatz. Der Navier-Stokes–Löser NaSt3DGPF wird mit dem GNU C-Compiler in der Version 4.2.3 und dem Optimierungsflag `-O3` übersetzt. Gleichzeitig wird die gerade erwähnte MPI-Bibliothek hinzu gelinkt um ein paralleles Arbeiten zu ermöglichen.

8.2 Der Canyon - Eine Simulation mit komplexer Geometrie

Ein besonderer Vorzug des in dieser Arbeit implementierten Maya-Plugins zur Konfiguration von Strömungssimulationen ist die Möglichkeit, schnell und einfach Simulationen in komplexen Geometrien vornehmen zu können. Dies wird möglich, da das Plugin die Festlegung von Hindernissen in Fluidsimulationen über beliebige geschlossene polygonale Netze erlaubt. Das im Folgenden vorgestellte Beispielproblem dient der Demonstration der gerade beschriebenen Fähigkeit und wird vollständig mit dem interaktiven System zur Simulationskonfiguration in Maya zusammengestellt. Als Hindernisgeometrie kommt ein Canyon zum Einsatz. Er wurde von [1] bezogen und wird in Abbildung 8.1 gezeigt.

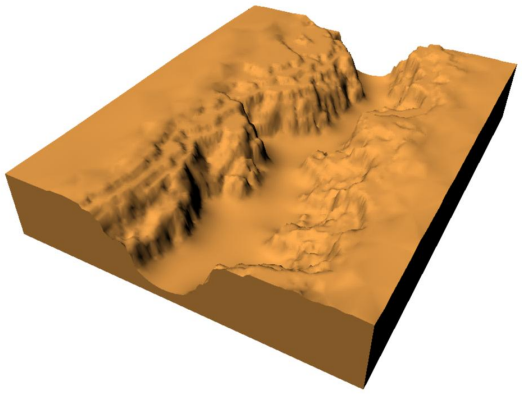


Abb. 8.1: Gerenderte Canyon-Geometrie, die mit Wasser geflutet wird.

Die Simulationsaufgabe besteht nun darin, den Canyon von vorne mit Wasser zu fluten. Für die Generierung eines Einströmungsbereiches mit der Form des Canyon-Profils kommt der `SimulationCustomInflowBoundary` (siehe Abschnitt 3.3.4 und 4.4.4) zum Einsatz. Dieser wird mit einem polygonalen Netz geschnitten, das genau die Form des invertierten Querschnitts des Canyons hat. Um ein solches Mesh zu generieren wird über eine boolesche Differenzbildung das Canyon-Profil aus einem Quader herausgeschnitten. Abbildung 8.2 auf Seite 118 zeigt das Resultat dieser Operation.

Es werden Ausströmungsrandbedingungen sowohl auf der vollen Fläche oberhalb des Canyons als auch auf der dem Einströmungsrand gegenüberliegenden Seite definiert. Hierdurch kann die durch das einfließende Wasser verdrängte Luft abfließen. Alle weiteren wichtigen Simulationsparameter befinden sich in Tabelle 8.1 auf Seite 118.

An Hand der Canyon-Simulation sollen nun zwei Aspekte untersucht werden. Zum einen werden Performance-Messungen für die Algorithmen zur Konfiguration einer Simulation und insbesondere für die Voxelisierung von polygonalen Netzen in Maya durchgeführt. Zum anderen zeigt die Simulation, dass der Einsatz von beliebig komplexen Geometrien in einer Strömungssimulation gelingt.

Bei der Erzeugung der Szenenbeschreibung für den Strömungslöser müssen Voxelisierungen für die Ausflussränder, für den beliebig geformten Einflussrand und für den Canyon, also die

Größe des Simulationsgebietes:	$150\,m \times 50\,m \times 190\,m$
Auflösung:	$225 \times 75 \times 285$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des einströmenden Wasserspiegels:	$20\,m$
Einströmgeschwindigkeit:	$15\,m/s$

Tabelle 8.1: Parameter der Canyon-Simulation.

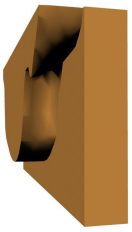


Abb. 8.2: Seitliche Ansicht auf das polygonale Netz mit dem der Einströmungsrand definiert wird.

Hindernisgeometrie, generiert werden. Eine Messung der Laufzeit dieses Umwandlungsprozesses wird auf der eingangs vorgestellten Graphik-Workstation bei vier verschiedenen Auflösungen durchgeführt. Die Ergebnisse sind in Tabelle 8.2 auf Seite 119 eingetragen.

Abbildung 8.3 auf Seite 119 zeigt ein Diagramm der Laufzeit bezüglich der Anzahl der Simulationszellen. Sehr schön lässt sich erkennen, dass die Zeit linear in der Zahl der Zellen ist. Die in dieser Arbeit eingesetzten Voxelisierungsverfahren haben keine optimale Komplexität. Begründet wurde dies schon in Abschnitt 4.3. Dementsprechend kann die Laufzeit nicht mit aktuellen Voxelisierungsansätzen verglichen werden. Dennoch sind die in Tabelle 8.2 angegebenen Zeiten akzeptabel, wenn man bedenkt, dass die eigentliche Simulation in Bezug auf die jeweilige Auflösung ein Vielfaches der Zeit benötigt.

Die Simulation des vorliegenden Testbeispiels wird auf dem Cluster *Himalaya* mit 64 Prozessen durchgeführt. Für 3.85 Sekunden simulierte Zeit benötigt das System eine Laufzeit von 6 Stunden und 32 Minuten. In Anbetracht dieser außerordentlich großskaligen Simulation ist die Laufzeit gut. Bei der Simulation werden 205 Megabyte an Oberflächendaten erzeugt. Die vollen Simulationsdaten erfordern demgegenüber 20 Gigabyte Speicherplatz. Wie erwünscht führt die Beschränkung auf das Abspeichern der Oberfläche als polygonales Netz zu einer deutlichen Datenreduktion.[2]

Ein gerendertes Video des Canyons in Echtzeit liegt der Arbeit bei. Das Rendering des Videos benötigt eine Laufzeit von ca. 43 Minuten. Alternativ zeigt Abbildung 8.4 auf Seite 120 den Fortschritt der Simulation zu verschiedenen Zeitpunkten in Einzelbildern. Wie man in der

[2]Durch den Einsatz eines Binärformates zum Speichern der polygonalen Netze anstatt des Wavefront OBJ - Formates können zusätzliche Einsparungen beim Speicherverbrauch erreicht werden (siehe Abschnitt 5.3.1).

Abbildung erkennen kann, fließt das Wasser sehr realistisch durch die sehr komplexe Geometrie. Es zeigt sich also, dass der Strömungssimulator selbst mit kompliziertesten Hindernissen zurechtkommt.

Auflösung	Zellen-Anzahl	Kantenlänge einer Zelle	Laufzeit
$75 \times 25 \times 95$	178125	$2.00\,m$	$97\,s$
$150 \times 50 \times 190$	1425000	$1.00\,m$	$801\,s$
$225 \times 75 \times 285$	4809375	$0.67\,m$	$2719\,s$
$300 \times 100 \times 380$	11400000	$0.50\,m$	$6471\,s$

Tabelle 8.2: Performance-Ergebnisse bei der Voxelisierung des Canyons und seines Einströmungsbereiches.

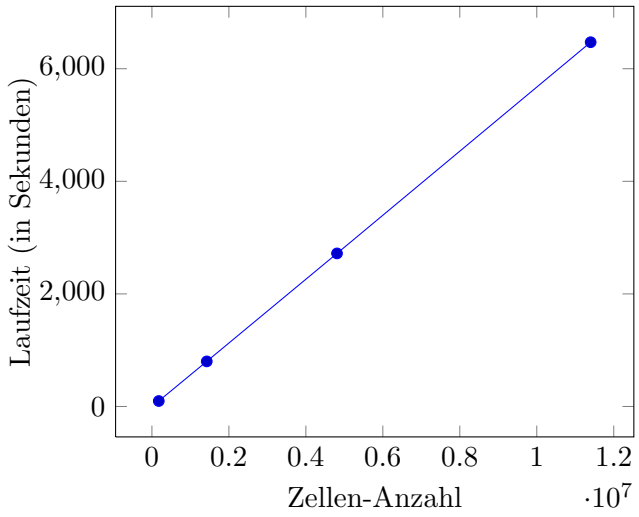


Abb. 8.3: Die Laufzeit der Voxelisierungsalgorithmen für das Canyon-Beispiel in Abhängigkeit von der Anzahl der zu voxelisierenden Zellen.

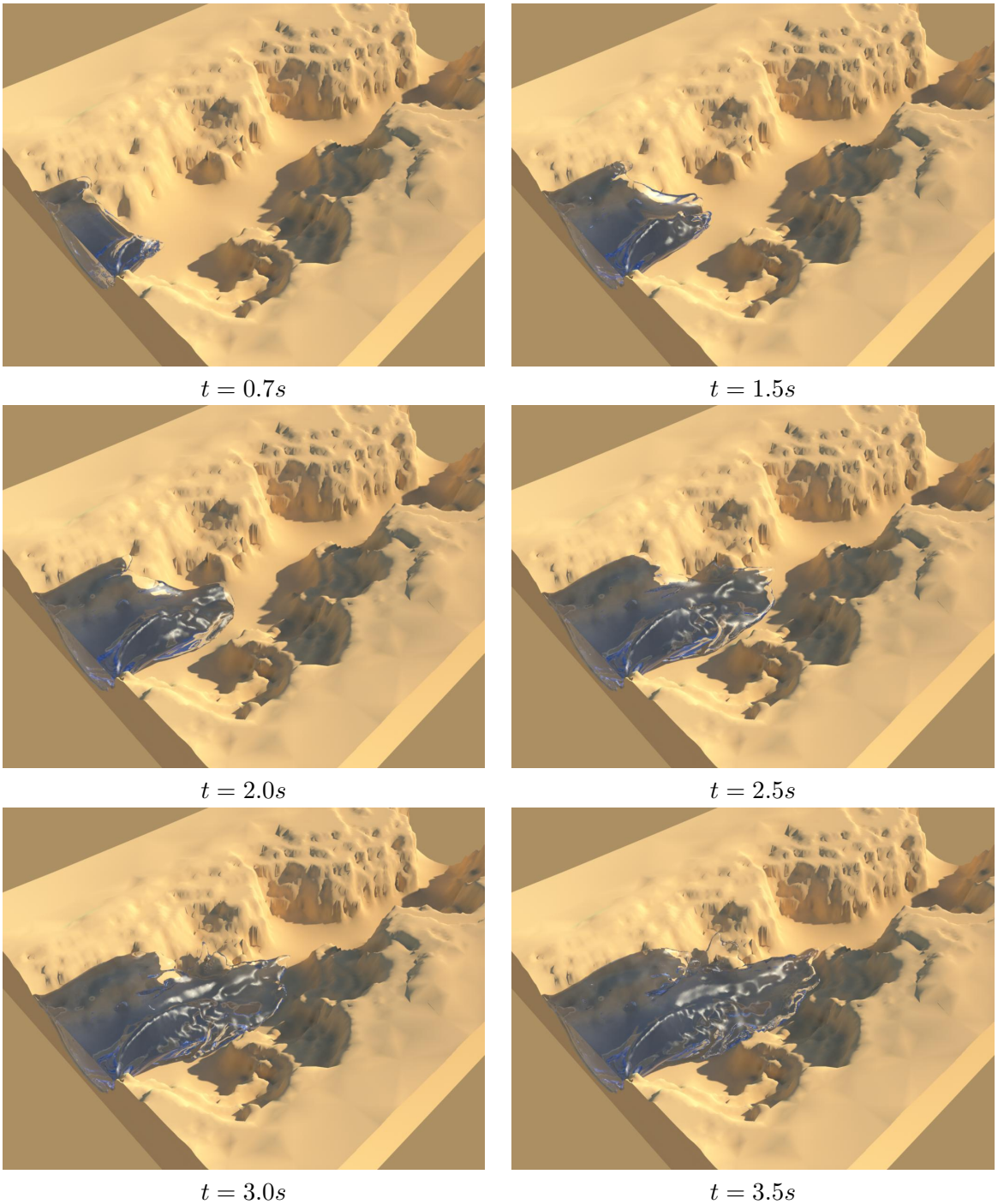


Abb. 8.4: Das Wasser fließt bei einer Geschwindigkeit von 15 m/s bzw. 54 km/h in die komplexe Geometrie des Canyons.

8.3 Performance-Analyse des Marching-Cubes–Algorithmus

Als wesentlicher Bestandteil des Visualisierungsprozesses wird in Kapitel 5 der Marching-Cubes–Algorithmus zur Extraktion von Isoflächen vorgestellt. Unter Einsatz dieses Verfahrens erzeugt der Navier-Stokes–Löser polygonale Netze zur Repräsentation der freien Oberfläche zwischen den Fluidphasen.

In diesem Abschnitt soll die in Abschnitt 5.3 beschriebene Implementierung des Marching-Cubes–Algorithmus auf ihre Performance hin analysiert werden. Dazu wird mit dem Verfahren die freie Oberfläche aus einer speziellen Simulationsumgebung extrahiert. Für den Performance-Test wird eine Umgebung gewählt, die grundsätzlich mit einem Gas gefüllt ist. Am *Boden* befindet sich ein Wasserbecken, das bis zu den Rändern des Simulationsbereiches reicht. Außerdem enthält die Umgebung eine spärenförmige Flüssigkeitsregion. Abbildung 8.5 zeigt eine photorealistische Visualisierung der freien Oberfläche des Testbeispiels mit und ohne zusätzlich eingefügten Randflächen.

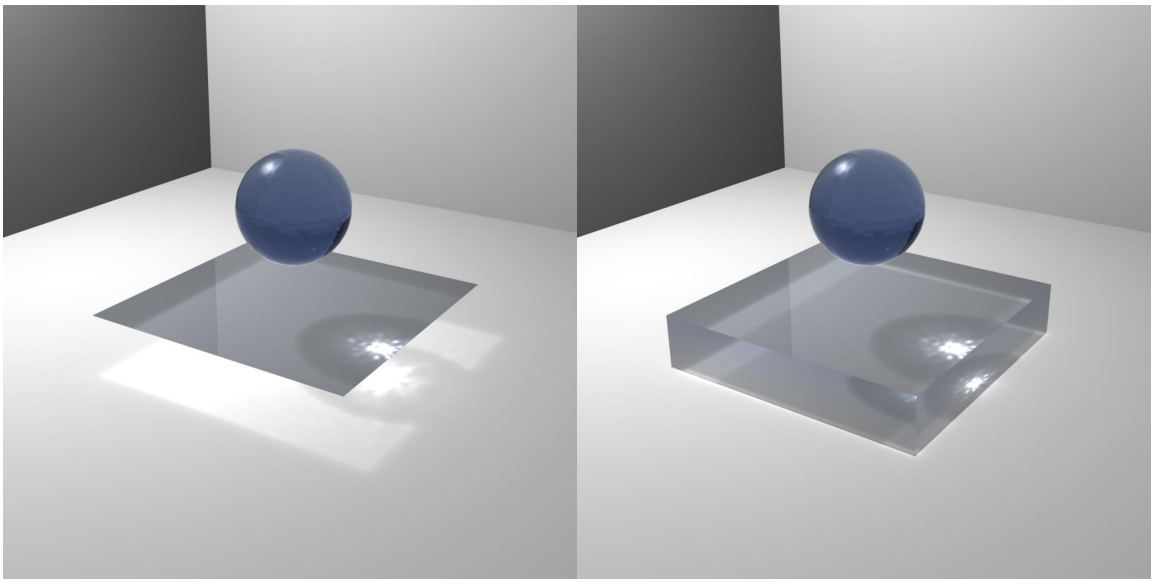


Abb. 8.5: Visualisierung des Testproblems ohne *(links)* und mit *(rechts)* Abschluss der freien Oberfläche am Rand des Berechnungsgebietes bei einer Auflösung von $200 \times 200 \times 200$.

Mit der sphärenförmigen Flüssigkeitsregion soll die Fähigkeit des Algorithmus überprüft werden, komplexe, nicht planare Oberflächen zu extrahieren. Im Gegensatz dazu dient der Flüssigkeitsbereich am Boden der Simulationsumgebung der Geschwindigkeitsanalyse des Verfahrens zum Schließen der freien Oberfläche.

Die künstliche Wahl des vorliegenden Testproblems ist für die Performance-Analyse unausweichlich, da die Extraktion der freien Oberfläche für unterschiedliche Auflösungen überprüft werden soll. Bei einem weniger synthetischen Simulationsbeispiel würden für voneinander abweichende Auflösungen unterschiedliche Geometrien berechnet werden. Man bekäme nicht miteinander vergleichbare Ergebnisse.

Für die Geschwindigkeitsmessungen kommt das Cluster-System zum Einsatz. Auf diesem

werden Simulationen mit der zuvor beschriebenen Konfiguration durchgeführt. Dabei wird stets nur die freie Oberfläche für den ersten Zeitschritt der Berechnungen extrahiert. Um die gemessenen Zeiten von der Auslastung des clustereigenen Netzwerkdateisystems unabhängig zu machen, werden die extrahierten polygonalen Netze auf die lokale Festplatte eines jeden Knotens geschrieben.

Gemessen wird die Laufzeit für folgende Auflösungen des Simulationsgebietes: $50 \times 50 \times 50$, $100 \times 100 \times 100$, $200 \times 200 \times 200$, $400 \times 400 \times 400$ und $500 \times 500 \times 500$. Dabei wird die Oberflächenextraktion mit 64, 32, 16, 8, 4, 2 und einem (parallelen) Prozess durchgeführt. Außerdem werden die polygonalen Netze sowohl mit als auch ohne Abschluss am Rand erzeugt.

Tabelle 8.3 zeigt die Ergebnisse der Performance-Tests für den reinen Marching-Cubes–Algorithmus. Für die Ergebnisse des Verfahrens mit einer zusätzlichen Generierung einer abschließenden Randgeometrie sei auf Tabelle 8.4 verwiesen. Die Laufzeiten werden in beiden Tabellen in Sekunden angegeben. Sie ergeben sich aus der Mittelung von vier Messungen und geben die Zeit an, die der langsamste parallele Prozess für die Extraktion der Oberfläche benötigt hat. In einigen Fällen kann keine Zeit ermittelt werden, da das zu berechnende Problem zu groß für den Arbeitsspeicher der beteiligten Rechner ist.

Prozessoren	Auflösung				
	50^3	100^3	200^3	400^3	500^3
1	0.12196	0.84781	6.29526	-	-
2	0.08211	0.42475	3.14102	-	-
4	0.06532	0.32325	2.18385	16.79130	-
8	0.02615	0.15614	1.09801	8.87747	16.53250
16	0.02461	0.09219	0.57945	4.85934	8.26779
32	0.01063	0.06017	0.46799	3.34169	6.81967
64	0.00711	0.03157	0.21986	1.68164	3.29093

Tabelle 8.3: Die Laufzeiten des implementierten Marching-Cubes–Algorithmus ohne Abschluss am Rand (Angaben in Sekunden).

Prozessoren	Auflösung				
	50^3	100^3	200^3	400^3	500^3
1	0.17415	1.04407	7.11975	-	-
2	0.09957	0.52986	3.69462	-	-
4	0.07679	0.40514	2.61589	18.66062	-
8	0.04882	0.20487	1.32409	9.37901	17.69246
16	0.03889	0.10392	0.66128	5.08305	9.04915
32	0.01574	0.09050	0.57291	4.14843	8.03593
64	0.00743	0.04952	0.32455	1.99461	3.72143

Tabelle 8.4: Die Laufzeiten des implementierten Marching-Cubes–Algorithmus mit eingefügten Randflächen (Angaben in Sekunden).

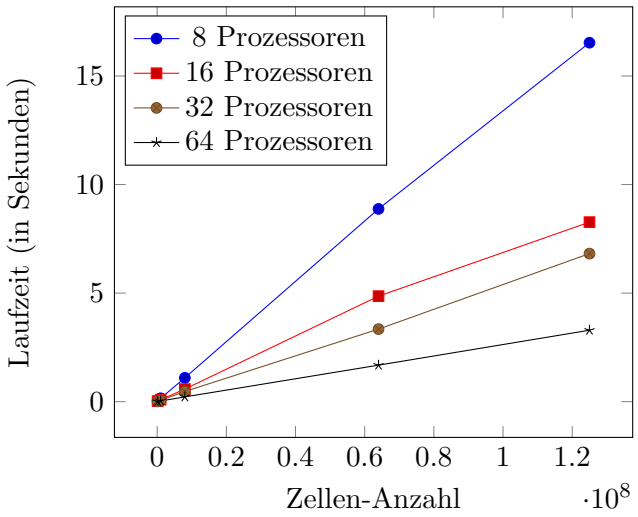


Abb. 8.6: Laufzeit des Marching-Cubes–Algorithmus in Abhängigkeit von der Anzahl der Zellen im Diskretisierungsgitter für verschiedene Parallelisierungsstufen (ohne Randflächen-Extraktion).

In Abbildung 8.6 wird die Laufzeit des Marching-Cubes–Algorithmus (aus Tabelle 8.3) bezüglich der Anzahl der Zellen des Diskretisierungsgitters gezeigt. Für alle dargestellten Stufen der Parallelisierung lässt sich die erwartete lineare Laufzeit erkennen.

Abbildung 8.7 auf Seite 124 zeigt die Geschwindigkeitsverbesserung durch Parallelisierung bei verschiedenen Auflösungsstufen (bzgl. Tabelle 8.3). Auch hier verhält sich die Implementierung genau wie gewünscht: Mit linear steigender Prozessorzahl fällt die Rechenzeit exponentiell ab.

Schließlich vergleicht Abbildung 8.8 auf Seite 124 die Geschwindigkeiten der Marching-Cubes–Varianten ohne und mit der Generierung von Randflächen. Man kann dem Balkendiagramm entnehmen dass die zusätzliche Erzeugung der Randflächen die Laufzeit gering erhöht. Dies ist verständlich, da mehr Polygone erzeugt werden müssen. Insgesamt kann man aber den sehr geringen zusätzlichen Zeitaufwand tolerieren, da durch ihn in einigen Fällen eine optisch ansprechendere Visualisierung möglich wird.

Um die zuvor vorgestellten Ergebnisse einordnen zu können, muss man beachten, dass eine Navier-Stokes–Simulation mit einer Auflösung von 200^3 Zellen bereits die in dieser Arbeit verwendete Hardware stark auslastet. Üblicherweise werden deshalb kaum höhere Auflösungen gewählt. Die Auflösungen 400^3 und 500^3 dienen vor allem der Demonstration der guten Skalierbarkeit der Parallelisierung.

Als weiterer Anhaltspunkt für die Einordnung der Performancemessungen ist ein Vergleich der Laufzeit des Marching-Cubes–Algorithmus mit der Gesamtlaufzeit eines einzelnen Zeitschrittes des Navier-Stokes–Lösers sinnvoll: Lässt man das Testproblem als Simulation bei einer Auflösung von 200^3 Zellen auf 32 Prozessoren mit je 300 Iterationen des impliziten Posissonlösers zur Druckkorrektur laufen, benötigt ein einzelner Zeitschritt mindestens 42 Sekunden. Demgegenüber braucht der Marching-Cubes–Algorithmus unter diesen Bedingungen 0.46799 Sekunden. Die Gesamtzeit für einen Zeitschritt ist also etwa 90 Mal so groß wie die Laufzeit der Oberflächenextraktion. Somit wird mit der Implementierung genau die im Abschnitt 5.3.2 erwartete zu vernachlässigende kleine Laufzeit des Extraktionsverfahrens erreicht.

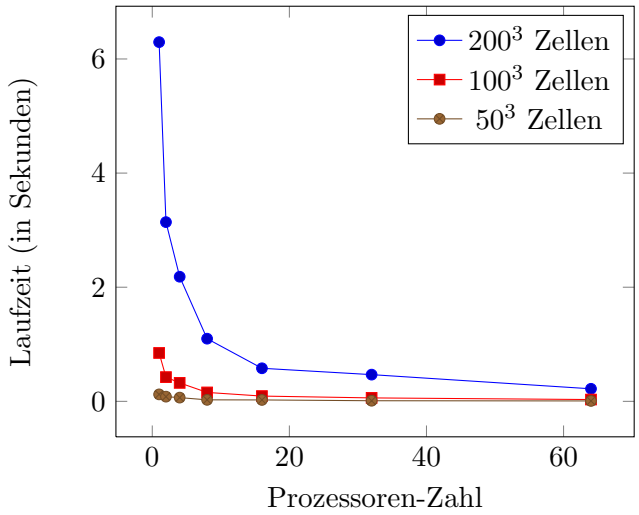


Abb. 8.7: Laufzeit des parallelen Marching-Cubes–Algorithmus in Abhängigkeit von der Anzahl der verwendeten Prozessoren für verschiedene Auflösungen des Diskretisierungsgitters (ohne Randflächen-Extraktion).

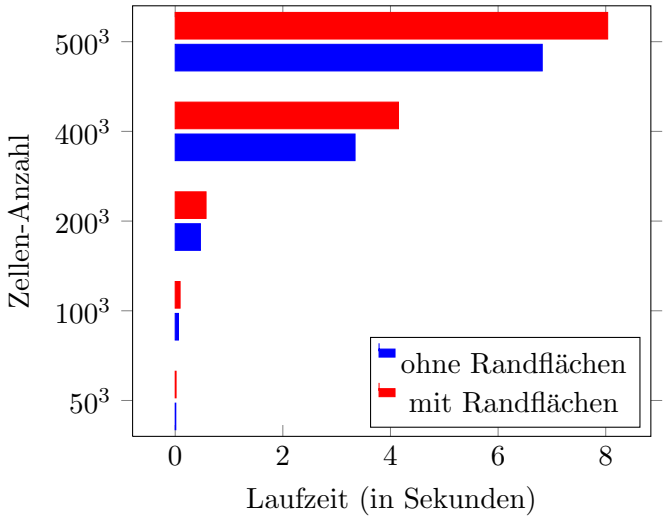


Abb. 8.8: Vergleich der Laufzeit des Marching-Cubes–Algorithmus mit und ohne zusätzlicher Erzeugung von Randflächen (beim Einsatz von 32 Prozessoren).

8.4 Der Wasserstrahl - Ein Testbeispiel für die Tropfengenerierung

In diesem Abschnitt soll gezeigt werden, dass durch die Anbindung des Partikelsystems an die Level-Set–Funktion und durch die Heuristik zur Tropfenemission aus Kapitel 7 die visuelle Qualität von Fluidanimationen erhöht werden kann. Als Testfall soll ein Wasserstrahl simuliert werden, der in ein Wasserbecken fließt. Abbildung 8.9 ist eine Skizze der initialen Simulationskonfiguration.

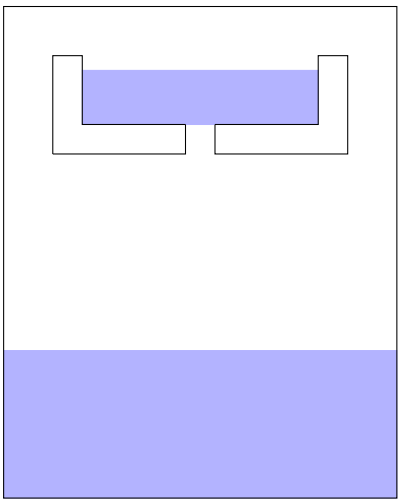


Abb. 8.9: Skizze der initialen Konfiguration der Wasserstrahl-Simulation.

In der Abbildung ist neben dem Wasserbecken am *Boden* ein weiteres erhöhtes Becken mit Wasser dargestellt. Aus dem kreisrunden Loch des erhöhten Beckens entweicht über die Zeit hinweg ein Wasserstrahl. Die allgemeinen Konfigurationsparameter der Simulation werden in Tabelle 8.5 auf Seite 126 zusammengefasst.

Es ist nun zu erwarten, dass beim Auftreffen des Wasserstrahles auf das untere Wasserbecken starke Turbulenzen auftreten. Diese führen unter anderem zu Spritzern. Gleichzeitig ist aber die Simulation nur gering aufgelöst, so dass die Spritzer auf Grund ihrer Größe nicht richtig dargestellt werden können. Hier sollen nun das Partikelsystem mit massebehafteten Partikeln und die Tropfenemissionsheuristik helfen. Durch diese werden selbst kleinste Fluidtröpfchen aufgelöst, so dass der optische Eindruck verbessert wird.

Alle für die Simulation benötigten Konfigurationseinstellungen und Geometrien werden über das in dieser Arbeit implementierte Konfigurationsplugin interaktiv zusammengestellt. Die Berechnung erfolgt mit 16 parallelen Prozessen. Für eine simulierte Zeit von 1.95 Sekunden benötigt das Clustersystem 44 Minuten. Bei der Simulation entstehen 481 Megabyte an Oberflächendaten. Durch das Abspeichern der vollen Simulationsdaten würden Daten im Umfang von 2.5 Gigabyte anfallen. Das Ergebnis wird als Video in vierfacher Zeitlupe ohne und mit den Tröpfchen aus dem Partikelsystem visualisiert. Die Renderingzeit für die Szene ohne Tröpfchen beträgt 43 Minuten. Dahingegen braucht die Generierung der photorealistischen Bilder mit Tropfen 3 Stunden und 58 Minuten. Ein Grund für die erhebliche Zeitdifferenz ist, dass die Tropfen als polygonale Sphären mit dem *Dielectric Shader* gerendert werden. Für ein korrektes

Größe des Simulationsgebietes:	$4.0\,m \times 5.0\,m \times 4.0\,m$
Auflösung:	$60 \times 75 \times 60$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels im unteren Becken:	$1.5\,m$
Höhe des Wasserspiegels im oberen Becken:	$0.7\,m$
Radius der emittierten Tropfen:	$0.5\,cm$
`DropRemovalRadiusFactor`	3.0
`InjectionProbabilityFactor`	1000
`CurvatureThreshold`	-3.5

Tabelle 8.5: Simulationsparameter der Wasserstrahl-Simulation.

Rendering mit Lichtbrechungen und Reflektionen in den Tropfen müssen die Qualitätseinstellungen beim Raytracer *Mental Ray* erhöht werden.

Einzelbilder der Simulationsergebnisse können in den Abbildungen 8.10, 8.11 und 8.12 auf den Seiten 127, 128 und 129 betrachtet werden. Um den Unterschied zwischen den Visualisierungen mit und ohne die Partikel besser erkennen zu können werden die beiden Ergebnisse nebeneinander gezeigt. Auf der linken Seite finden sich die Bilder ohne und rechts die Bilder mit Partikeln zu den selben Zeitpunkten.

Sehr schön ist zu erkennen, wie bereits kurz nach dem Auftreffen des Wasserstrahls ($t = 0.66s$) die ersten Partikel entstehen. Sie bilden die Spritzer nach, die hier auf Grund der geringen Auflösung nicht entstehen können. Im darauffolgenden Bild ($t = 0.84s$) kann man erkennen, wie sich die Partikel durch das physikalisch motivierte Transportmodell und wegen ihrer initialen Geschwindigkeit weiterbewegen. Zum Zeitpunkt $t = 1.1s$ sind bereits fast alle Tropfen auf Grund der Gravitation in das Wasser eingetaucht. Gleichzeitig führen die Druckverhältnisse sowie die Krümmungswerte um den Eintauchpunkt des Wasserstrahles dazu, dass neue Tropfen erzeugt werden. Das nächste Bild ohne Tropfen ($t = 1.32s$) zeigt starke Deformationen der Level-Set–Funktion auf Grund der turbulenten Verhältnisse um die Eintauchposition des Strahls. Tatsächlich kann aber erst durch die Partikel im rechten Bild die volle Dynamik nachgebildet werden. Zum Zeitpunkt $t = 1.48s$ löst sich ein kleiner, durch die Level-Set–Funktion modellierter Tropfen ab. Im nächsten dargestellten Bild ($t = 1.78s$) ist dieser verschwunden. Dieses Diskretisierungsartefakt ist für eine optisch ansprechende Animation fatal. Betrachtet man dagegen für die beiden Zeitpunkte die Einzelbilder mit den emittierten Tröpfchen, so sorgen diese für den benötigten Detailreichtum, um die Szene realistischer wirken zu lassen.

Fasst man nun das Ergebnis der Wasserstrahl-Simulation zusammen, so kann man feststellen, dass das eingesetzte Partikelsystem sowie das Verfahren zur Tropfenemission genau die an sie gestellten Erwartungen erfüllen. Durch das physikalisch motivierte Modell für den Tropfentransport werden die Bewegungen der Partikel ausreichend genau dargestellt. Die Heuristik zur Tropfenemission erzeugt stets dann Tropfen, wenn sie zur genaueren Auflösung der Simulation benötigt werden. Insbesondere fallen die Diskretisierungsartefakte der grob aufgelösten Simulation nicht mehr ins Gewicht, da die als Tropfen gerenderten Partikel die visuell wichtigen Details nachbilden.

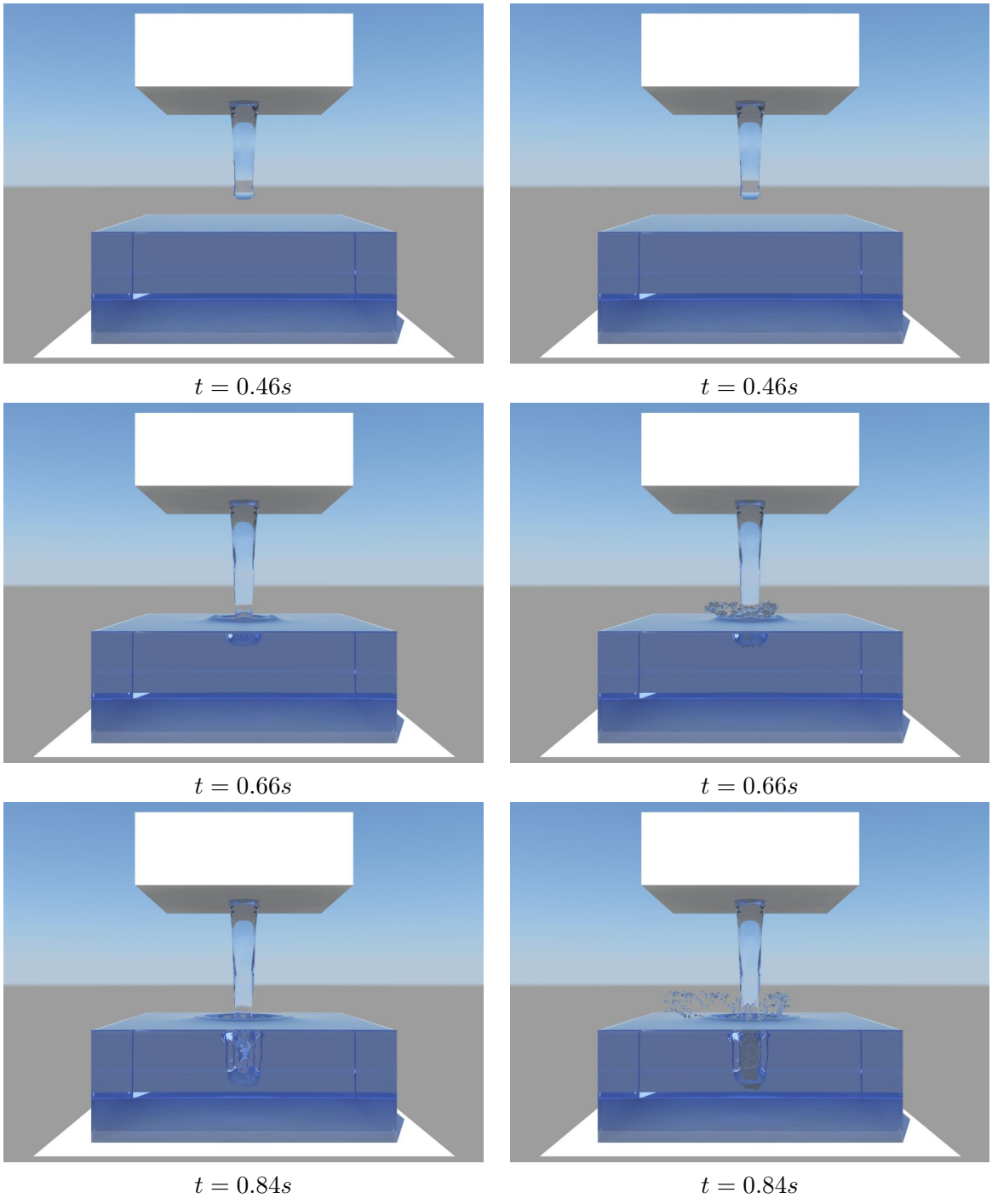


Abb. 8.10: Ein Wasserstrahl taucht in ein Wasserbecken ein. Visualisierung ohne (links) und mit (rechts) den Einsatz des Partikelsystems zur Darstellung von Wassertröpfchen.

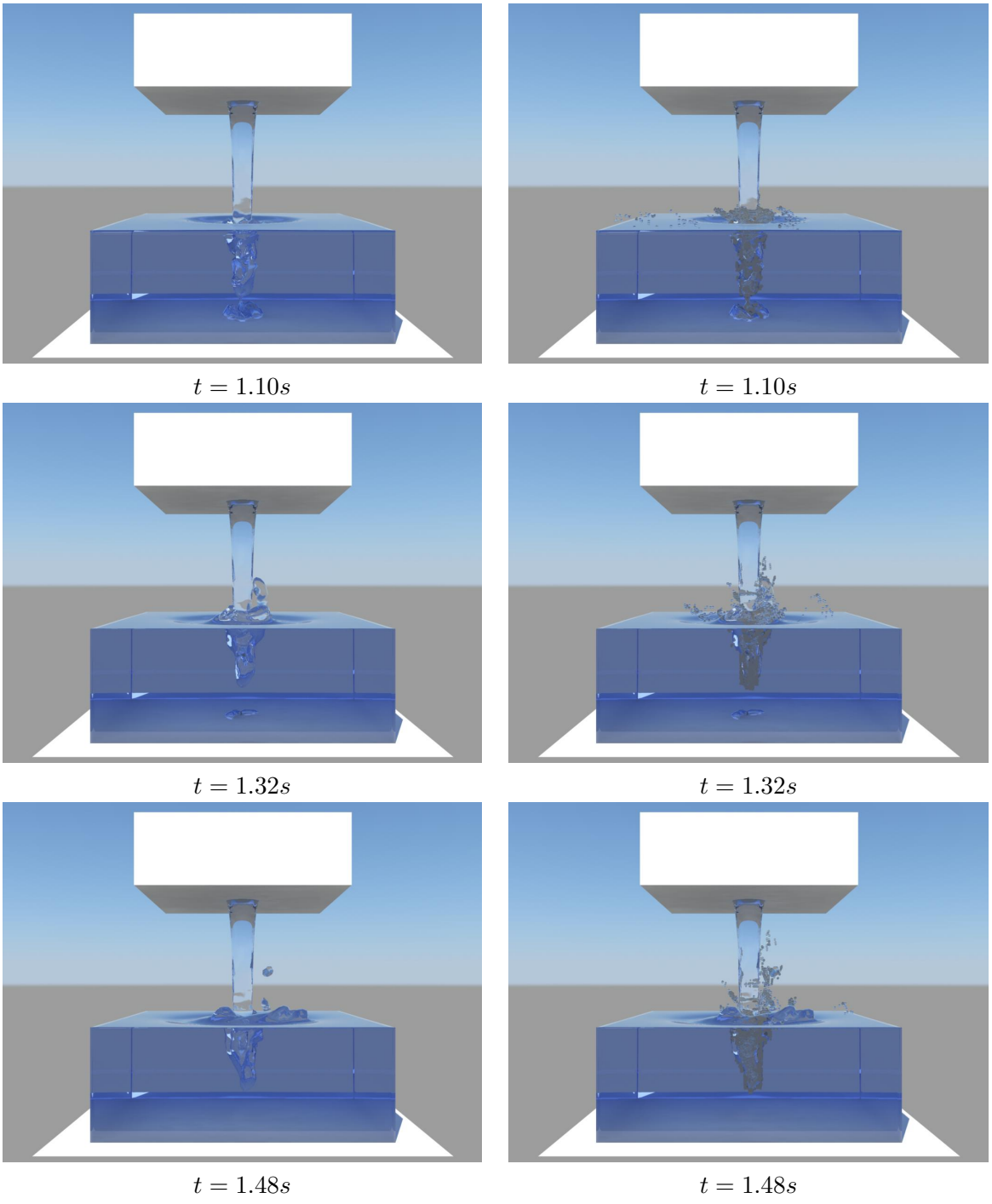
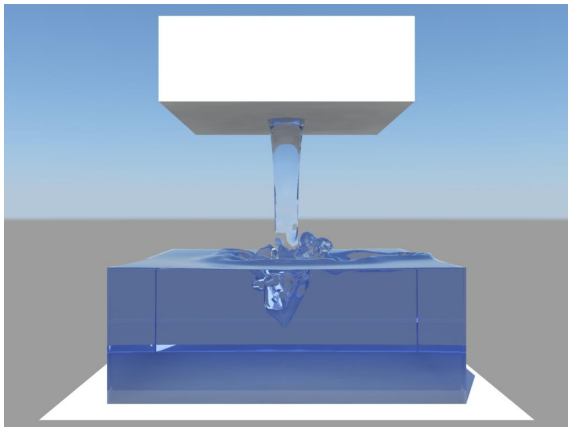
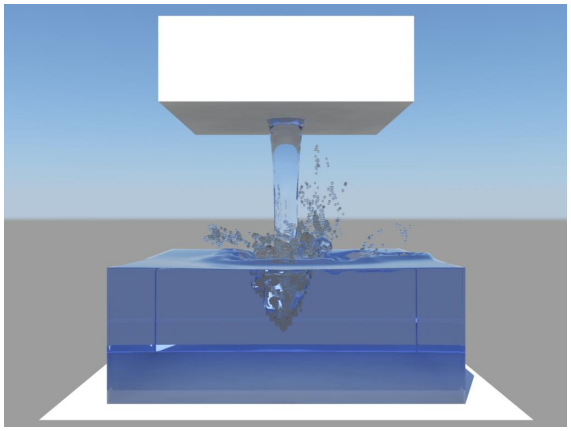


Abb. 8.11: Die Tropfen-Partikel (rechts) verbessern den visuellen Eindruck des turbulenten Wassers. Der alleinige Einsatz der Level-Set–Funktion (links) reicht hier nicht aus.



$t = 1.78s$



$t = 1.78s$

Abb. 8.12: Links: Darstellung des einschlagenden Wasserstrahls ohne Partikel. Rechts: Visualisierung mit zusätzlichen Tropfenpartikeln.

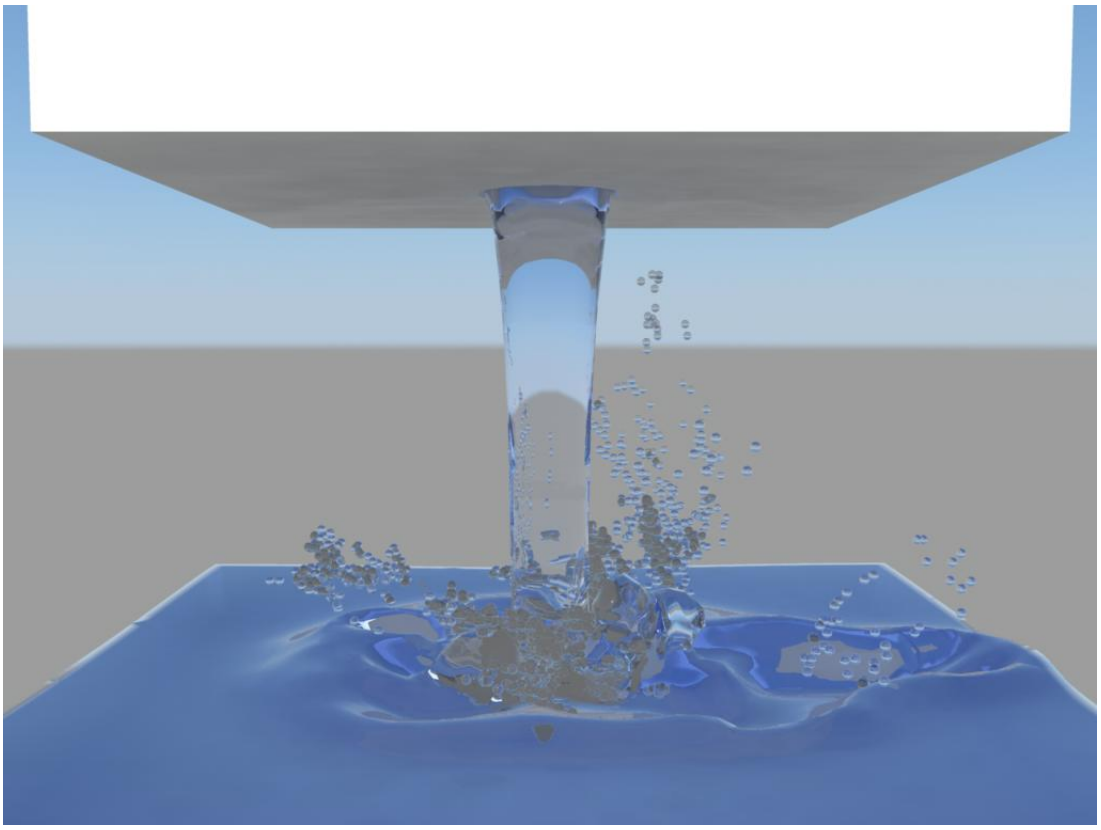


Abb. 8.13: Vergrößerte Darstellung des turbulenten Bereiches mit Partikeln zum Zeitpunkt $t = 1.78s$

8.5 Simulationen von Tropfeneinschlägen mit Partikeln

Nach der expliziten Betrachtung der Tropfengenerierung für das Partikelsystem im vorangegangenen Abschnitt sollen in diesem Abschnitt Testprobleme zur Simulation von Tropfeneinschlägen vorgestellt werden. Damit wird der zweite Teil der Zwei-Wege-Kopplung zwischen der Level-Set–Methode und Partikeln aus Kapitel 7 auf seine Qualität und Geschwindigkeit hin analysiert. Insgesamt werden drei Simulationen beschrieben, die jeweils auf verschiedene Aspekte des Tropfeneinschlags eingehen.

8.5.1 Einschlag eines einzelnen Tropfens

Im ersten Beispiel wird der Einschlag eines einzelnen Wassertropfens in ein Becken mit Wasser betrachtet. Der Tropfen wird mit der in Abschnitt 7.3.1 erwähnten Funktion, einzelne Partikel über eine Datei vorzugeben, in die Simulationsumgebung emittiert. Alle in diesem Fall wichtigen Parameter der Simulation sind in Tabelle 8.6 nachzulesen. Um einen möglichst genauen Eindruck des Tropfeneinschlags zu erhalten, wird bei dem vorliegenden Testfall eine relativ kleinskalige Simulation im Zentimeter-Bereich durchgeführt.

Größe des Simulationsgebietes:	$20\,cm \times 20\,cm \times 20\,cm$
Auflösung:	$80 \times 80 \times 80$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$7.5\,cm$
Radius des emittierten Tropfens:	$0.5\,cm$
Initiale Geschwindigkeit des Tropfens:	$0\,m/s$
Fallhöhe (vom Tropfenmittelpunkt aus):	$10.5\,cm$
`DropRemovalRadiusFactor`	3.0
`ForceFactor`	1000.0

Tabelle 8.6: Parameter für das Testproblem eines einzelnen Tropfeneinschlages.

Die Simulation wird auf dem Cluster *Himalaya* mit acht Knoten bzw. 16 parallelen Prozessen durchgeführt. Für eine Berechnung von zwei Sekunden Simulationszeit braucht das Testsystem 26 Minuten und 30 Sekunden. Während der Simulation werden 578 Megabyte an polygonalen Netzen erzeugt. Würde man die vollen Simulationsdaten abspeichern, entstünden 1.2 Gigabyte Daten. Die Abbildungen 8.14 und 8.15 auf den Seiten 131 und 132 zeigen das visualisierte Ergebnis der Simulation als Bildfolge. Alternativ ist der elektronischen Ausgabe der vorliegenden Arbeit ein gerendertes Video beigelegt. Das Video zeigt die Sequenz in einer Zeitlupe mit einem Viertel der Normalgeschwindigkeit, da der Tropfeneinschlag sonst zu schnell wäre. Es kann in 43 Minuten auf der Graphik-Workstation erzeugt werden.

Zum Zeitpunkt $t = 0.18s$ (in Abbildung 8.14) befindet sich der Tropfen noch über der Wasseroberfläche. Das darauffolgende Bild zeigt den gerade eingetauchten Tropfen, der aber noch nicht entfernt wurde. Gleichzeitig beginnt die Einschlagskraft zu wirken. Um dies besser

sehen zu können zeigt die Abbildung 8.16 auf Seite 133 zusätzlich eine vergrößerte orthogonale Ansicht des Tropfeneinschlagspunktes.

Es ist gut zu erkennen, wie sich kurz nach dem Einschlag die volle Kraft entfaltet und eine kreisrunde *Delle* in der Oberfläche erzeugt. Die in Abschnitt 7.2.2 eingeführte *geglättete* Gewichtsfunktion zur Verteilung der Kraft auf die Zellen sorgt dabei für eine Deformation, die frei von Diskretisierungsartefakten ist.

Ab dem Zeitpunkt $t = 0.22s$ wirkt die Einschlagskraft nicht mehr. Nun lassen die im Fluid vorhandenen Kräfte die Wasseroberfläche nach oben schwingen. Den vollen Ausschlag nach oben hat sie zum Zeitpunkt $t = 0.28s$. Damit stellt sich der erwünschte Anschwingprozess ein.

Ganz wie im Modell in Abschnitt 7.2.2 gefordert, ergibt sich in der Nachfolge des Tropfeneinschlags eine Schwingung auf der Oberfläche, die sich in alle Richtungen ausbreitet: Die typischen Ringe eines Tropfeneinschlags entstehen. Bei der Betrachtung des beigefügten Videos kann man schließlich noch beobachten, wie die angeregte Wasseroberfläche, ganz wie in der Realität, nach einer gewissen Zeit wieder ruhig wird.

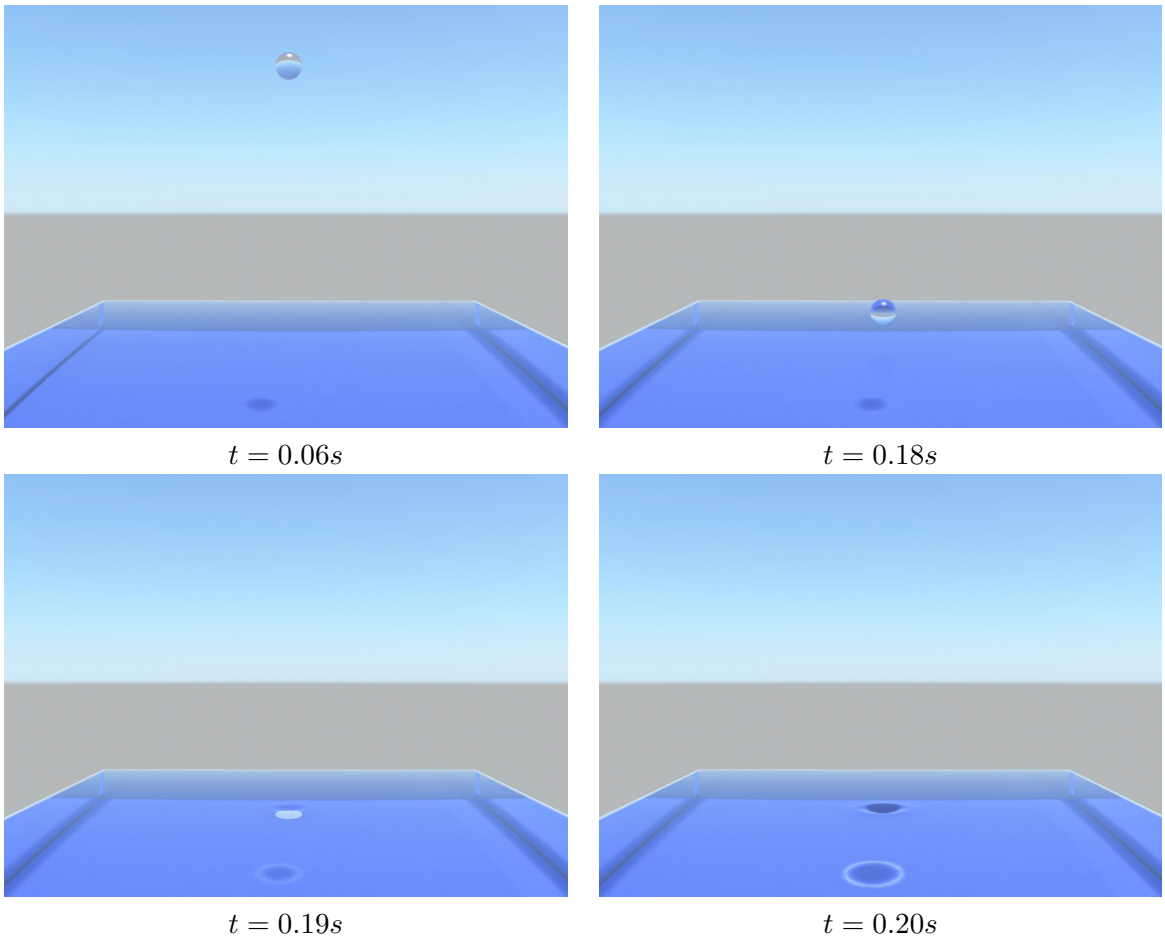


Abb. 8.14: Beim Einschlag des Partikels deformiert die Tropfeneinschlagsheuristik die Wasseroberfläche.

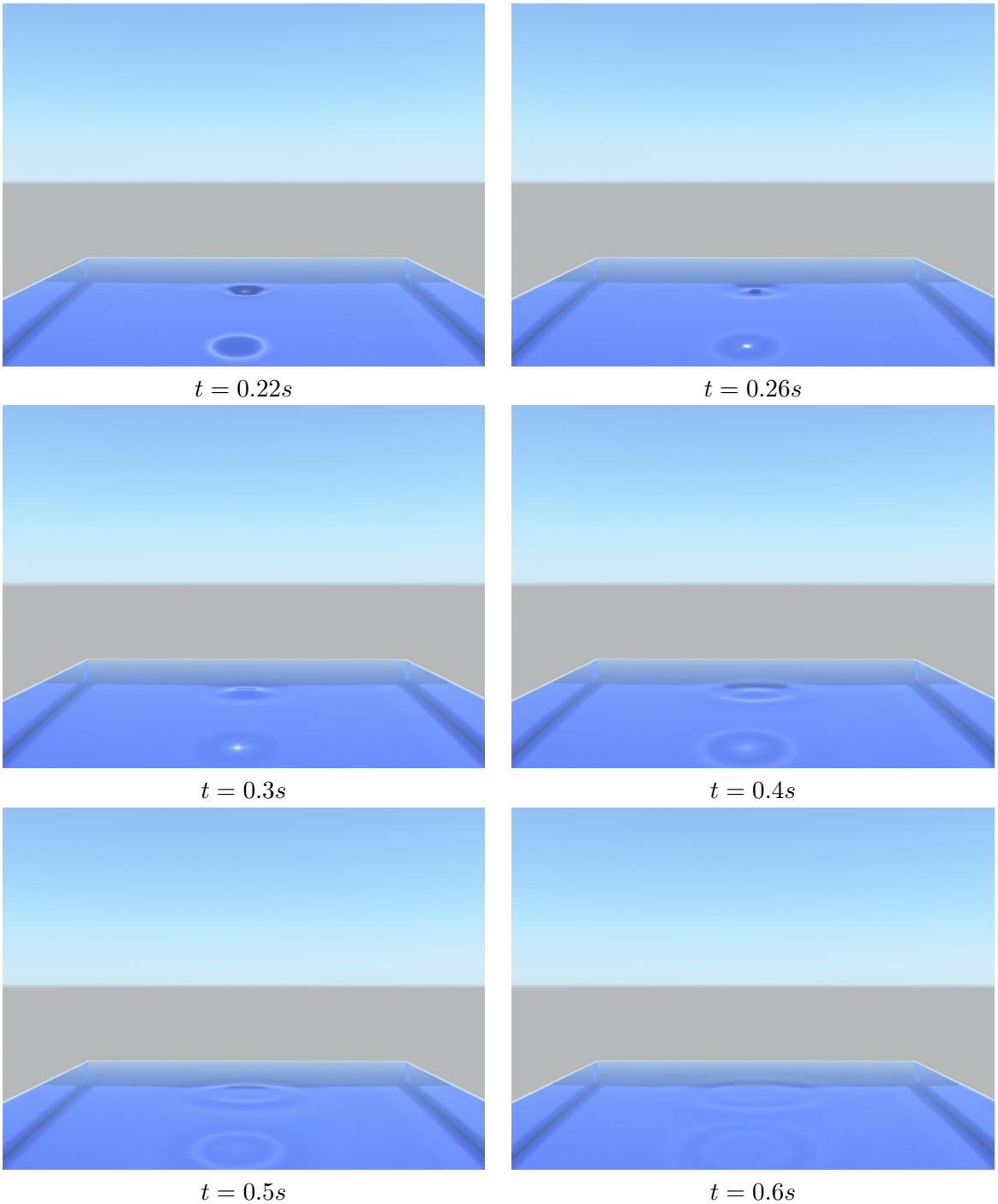


Abb. 8.15: Die deformierte Wasseroberfläche regt die erwünschte kreisförmige Wellenbildung an.

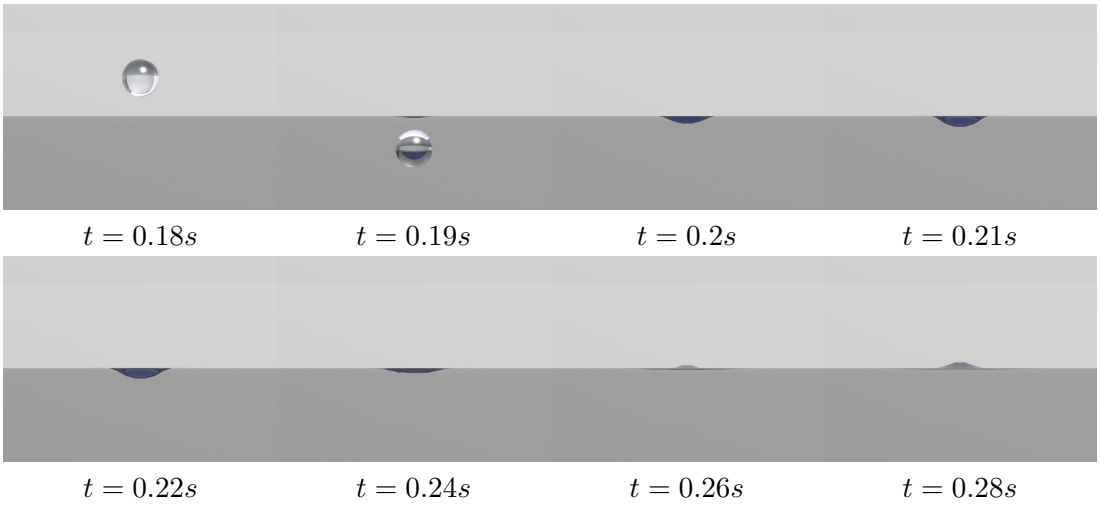


Abb. 8.16: Eine orthogonale Seitenansicht der Tropfeneinschlagsposition.

8.5.2 Regen-Simulation

Als Erweiterung der vorangegangenen Simulation soll bei dem zweiten Testbeispiel zu den Tropfeneinschlägen eine größere Anzahl an Tropfen in die Szene emittiert werden. Das Ziel der Simulation ist die Nachbildung eines Wasserbeckens bei Regen.

Die allgemeinen Parameter der Berechnung entsprechen denen in Tabelle 8.6 auf Seite 130. Abweichend vom letzten Abschnitt wird hier die Datei mit Tropfenstartpositionen durch ein Skript erzeugt. Das Skript legt randomisiert die Startpositionen von 219 Tropfen fest. Diese werden jeweils mit einem zeitlichen Abstand von $0.01s$ gestartet.

Simuliert wird eine Zeit von 4.5 Sekunden. Mit 16 parallelen Prozessen braucht dies auf dem eingangs beschriebenen Cluster 7 Stunden und 20 Minuten. Bei der Simulation entstehen 1.3 Gigabyte Oberflächendaten. Ein Abspeichern der vollen Simulationsdaten würde zu einem Speicherverbrauch von 10.5 Gigabyte führen. Die Rendering-Zeit für das Ergebnis-Video beträgt auf der Graphik-Workstation eine Stunde und 48 Minuten. Der Film läuft in Zeitlupe, um den sehr schnellen Tropfeneinschlagsprozess erkennen zu können.

Eine Serie von Einzelbildern des Videos ist in Abbildung 8.17 auf Seite 134 zu sehen. Sehr gut kann man die Überlagerungen der erzeugten Wellen erkennen. Insgesamt wirkt die Regensimulation sehr realistisch.

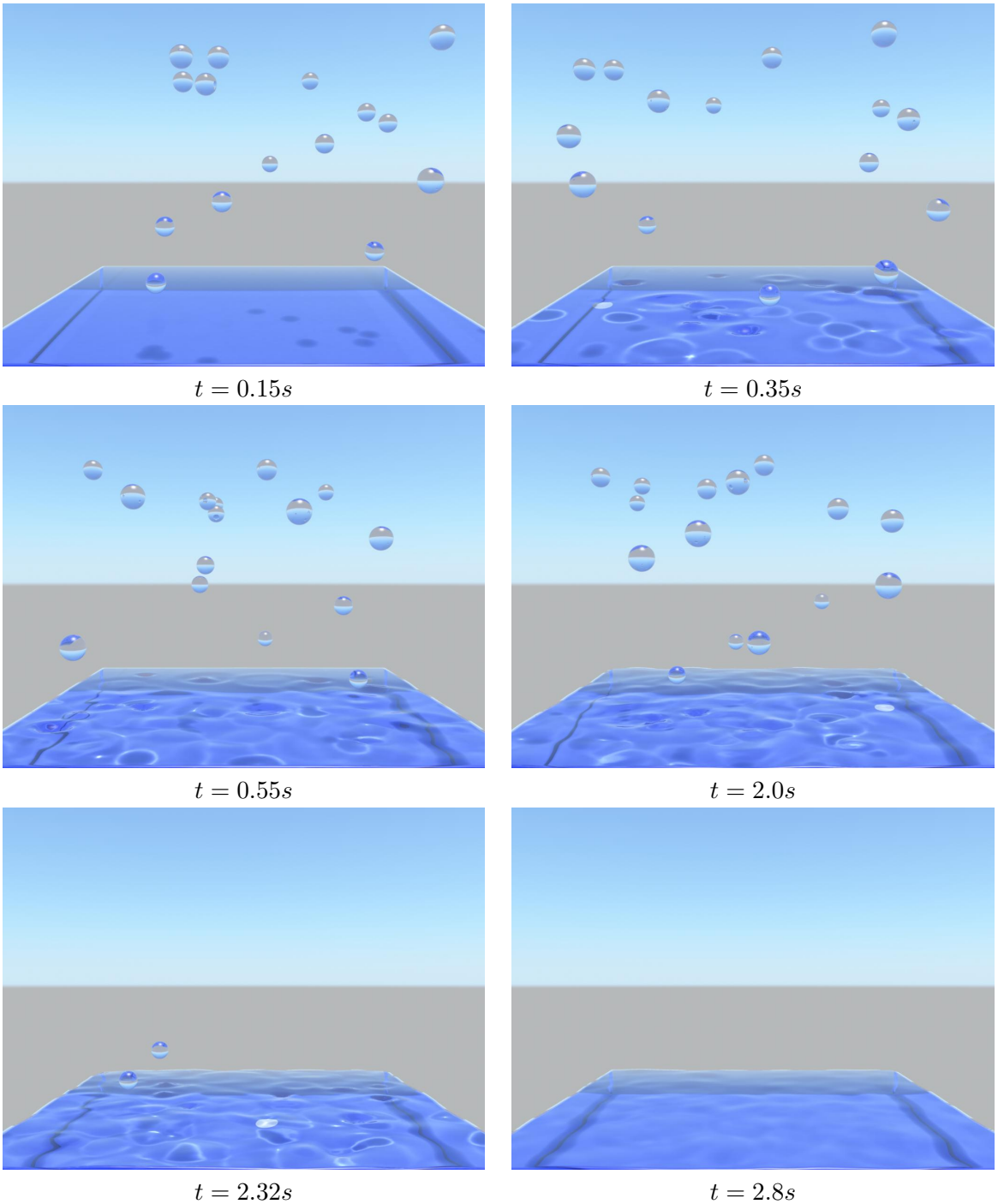


Abb. 8.17: Das Wasserbecken bei der Regensimulation vor, während und nach den Tropfeneinschlägen.

8.5.3 Tropfeneinschläge bei größeren Skalen

Bei den beiden bisherigen Tropfen-Simulationen hat die Simulationsumgebung jeweils die Größe von $20cm \times 20cm \times 20cm$ bei einer Auflösung von $80 \times 80 \times 80$. Dementsprechend hat jede Zelle der Diskretisierung eine Größe von $0.25cm \times 0.25cm \times 0.25cm$. Schlägt nun ein Tropfen des Radius $0.5cm$ in die Wasseroberfläche ein, so wirkt die Einschlagskraft auf ca. $4^3 = 64$ Zellen. Diese große Zellen-Zahl führt dazu, dass ein Tropfeneinschlag sehr gut aufgelöst werden kann.

In diesem Abschnitt soll geklärt werden, ob die Heuristik für den Tropfeneinschlag bei der selben Auflösung auch auf größeren Skalen funktioniert. Bei gleich großen Tropfen bedeutet dies, dass die Kraft des Einschlags auf weniger Zellen wirkt.

Als Basis der Simulation von Tropfeneinschlägen bei größeren Skalen dient abermals ein mit Wasser gefülltes Becken. Grundlegende Simulationsparameter finden sich in Tabelle 8.7.

Größe des Simulationsgebietes:	$1.0\,m \times 1.5\,m \times 1.0\,m$
Auflösung:	$80 \times 120 \times 80$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$0.3\,m$
Radius der emittierten Tropfen:	$0.005\,m$
Grundgeschwindigkeit der emittierten Tropfen:	$(0, 0, -0.7)^t\ km/m$ *(gejittert)*
Fallhöhe der Tropfen:	ca. $1.15\,m$
`DropRemovalRadiusFactor`	3.0
`ForceFactor`	300.0

Tabelle 8.7: Parameter des Testproblems für Tropfeneinschläge bei größeren Skalen.

Man kann den Werten in der Tabelle entnehmen, dass jede Diskretisierungszelle eine Größe von $0.0125cm \times 0.0125cm \times 0.0125cm$ hat. Demzufolge sind die Tropfen wie gewünscht bei einem Einschlag kleiner als eine Zelle und wirken somit nur sehr begrenzt.

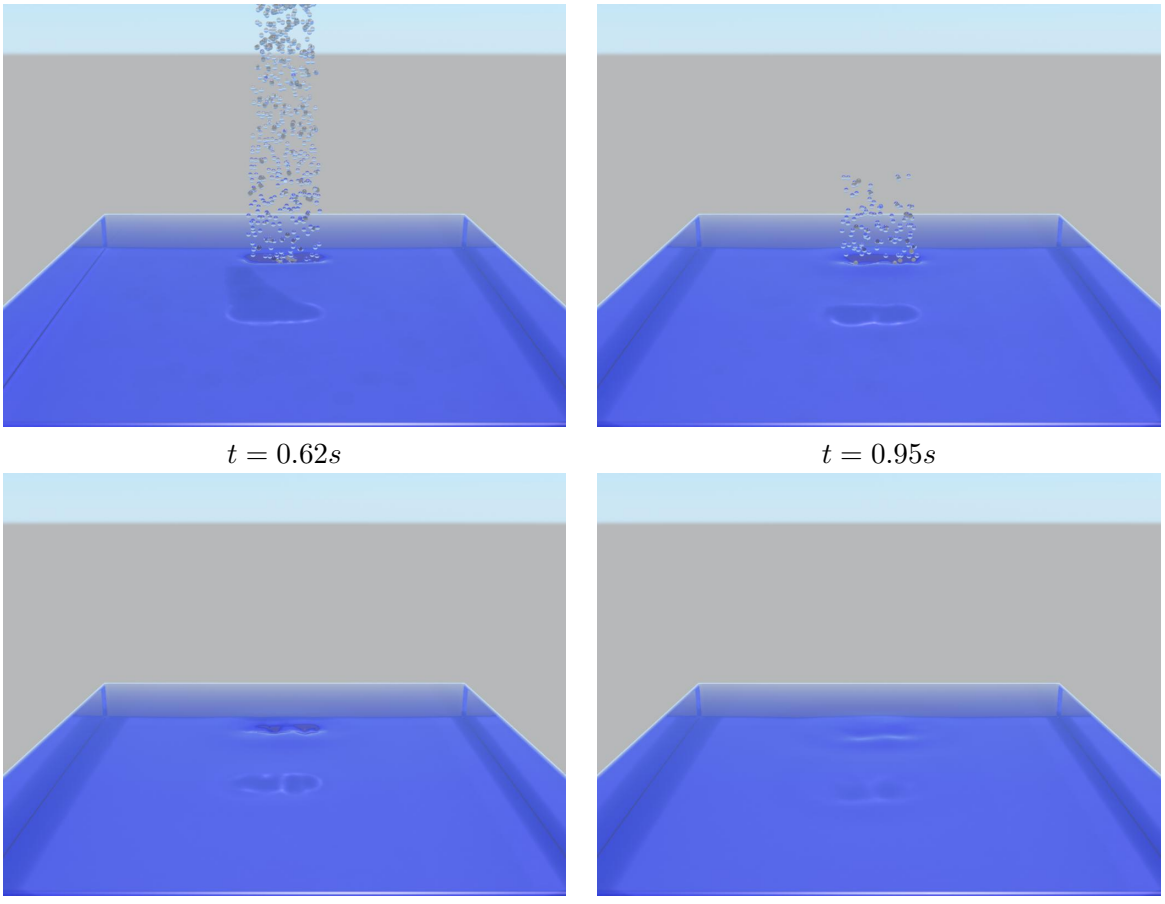
Bei der Simulation werden 2000 Tropfen emittiert. Diese starten alle an einem Punkt, haben aber eine unterschiedliche Geschwindigkeit: Unter Anwendung von Randomisierung wird die in Tabelle 8.7 angegebene Grundgeschwindigkeit in die X- und Y-Richtung gejittert. Die Startpositionen, die Startzeitpunkte und die initialen Geschwindigkeiten der Tropfen werden über den schon in den beiden vorherigen Beispielen benutzten Mechanismus angegeben. Insgesamt kann das vorliegende Beispiel als eine Simulation des Versprühens von Wassertropfen verstanden werden.

Simuliert wird die Szene über einen Zeitraum von 4 Sekunden. Eine Berechnung dauert auf dem Cluster mit 16 parallelen Prozessen 26 Minuten. Bei der Simulation werden polygonale Netze im Umfang von 1.1 Gigabyte erzeugt. Ein Abspeichern der vollen Simulationsdaten würde einen Speicherplatz von 13.7 Gigabyte beanspruchen. Das Video der simulierten Szene läuft in vierfacher Zeitlupe, um den Tropfeneinschlagsprozess genauer studieren zu können. Es kann in 2 Stunden und 5 Minuten gerendert werden. Einzelbilder des Videos finden sich in Abbildung 8.18 auf Seite 136.

Aus der Abbildung kann man entnehmen, dass die Tropfeneinschlags-Heuristik tatsächlich

auch für größere Skalen bzw. eine geringere Zellen-Zahl pro Tropfen funktioniert. Beim Aufschlag der *Tropfenwolke* deformiert sich wie gewünscht die Wasseroberfläche. Dies führt dazu, dass noch während des Einschlags der *Wolke* die erwünschte Wellenbewegung auf der Wasseroberfläche einsetzt. Dabei entspricht korrekterweise die Form der Wellenbewegung der Form der Aufschlagsstelle der Tropfen.

Insgesamt kann man den drei hier vorgestellten Simulationen von Tropfeneinschlägen entnehmen, dass der implementierte Mechanismus zur Rückkopplung des Partikelsystems mit der Fluidoberfläche erfolgreich eingesetzt werden kann.



$t = 0.62s$ $t = 0.95s$

$t = 1.0s$ $t = 1.28s$

Abb. 8.18: Die Tropfeneinschlagsheuristik funktioniert auch bei größeren Skalen, wie diese Bildsequenz zeigt.

8.6 Ein vollständig berechneter frei fallender Tropfen

In diesem Abschnitt wird die Simulation eines frei fallenden Tropfens mit anschließendem Einschlag in ein Wasserbecken erörtert. Statt, wie im vorangegangenen Abschnitt, den Tropfen über einen Partikel und den Einschlag über eine Heuristik zu berechnen, soll hier der Tropfen vollständig über die Level-Set–Methode simuliert werden. Das Simulationsbeispiel dient der Demonstration des außerordentlich geringen Modellfehlers im NaSt3DGPF-Löser der durch den Einsatz des Kontinuumsmodells der Navier-Stokes–Gleichungen sowie durch die Level-Set–Technik erreicht werden kann.

Unter bestimmten physikalischen Voraussetzungen entsteht beim Einschlag eines Wassertropfens in ein Wasserbecken ein sogenannter *Rayleigh-Jet* [13], [60]. Ein Rayleigh-Jet (oder auch *central jet*) ist eine Wassersäule die sich nach dem Tropfeneinschlag aufbäumt. Dieses Phänomen kann nur durch eine sehr genaue numerische Simulation nachgebildet werden. Somit ist es ein ideales Beispiel um die Fähigkeiten des NaSt3DGPF-Lösers und folglich auch die Fähigkeiten des hier vorgestellten Systems zur Animation von Fluiden zu testen. Alle wesentlichen Parameter der Simulation sind in Tabelle 8.8 zusammengefasst.

Größe des Simulationsgebietes:	$5.76\,cm \times 3.72\,cm \times 5.76\,cm$
Auflösung:	$180 \times 120 \times 180$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$0.96\,cm$
Durchmesser des fallenden Tropfens:	$0.576\,cm$
Fallhöhe (vom Tropfenmittelpunkt aus):	$2.325\,cm$
Grundgeschwindigkeit des Tropfens:	$0\,m/s$

Tabelle 8.8: Simulationsparameter bei der Berechnung des frei fallenden Tropfens.

Durchgeführt wird die Berechnung auf dem Cluster *Himalaya* mit insgesamt 64 Prozessoren. Da mit sehr kleinen Skalen gearbeitet wird, reicht eine Simulation von einer Sekunde um alle Effekte des Tropfeneinschlags sehen zu können. Für die Simulation benötigt das Clustersystem 21 Stunden und 20 Minuten. Hierbei erzeugt der Oberflächenextraktor 2.9 Gigabyte an Daten. Würde man die vollen Simulationsdaten abspeichern, entstünden Dateien im Umfang von 37 Gigabyte. Ein gerendertes Video aus einer seitlichen Ansicht in zehnfacher Zeitlupe liegt der Arbeit bei. Es entsteht auf der Graphik-Workstation in einer Zeit von einer Stunde und 36 Minuten.

In den Abbildungen 8.19 und 8.20 auf den Seiten 138 und 139 sind verschiedene Zeitpunkte der Simulation als Bilder dargestellt. So zeigen die ersten vier Bilder von Abbildung 8.19 wie der Wassertropfen in das flache Wasserbecken eintaucht. Die beiden darauffolgenden Visualisierungen geben wieder, wie der Rayleigh-Jet entsteht. Zum Zeitpunkt $t = 0.161s$ (in Abbildung 8.20) hat die Wassersäule ihre maximale Auslenkung erreicht. Anschließend fällt sie in sich zusammen, erzeugt aber gleichzeitig weitere kleine Wassersäulen. Einer dieser Nachschwinger ist in dem Bild zum Zeitpunkt $t = 0.293s$ zu sehen.

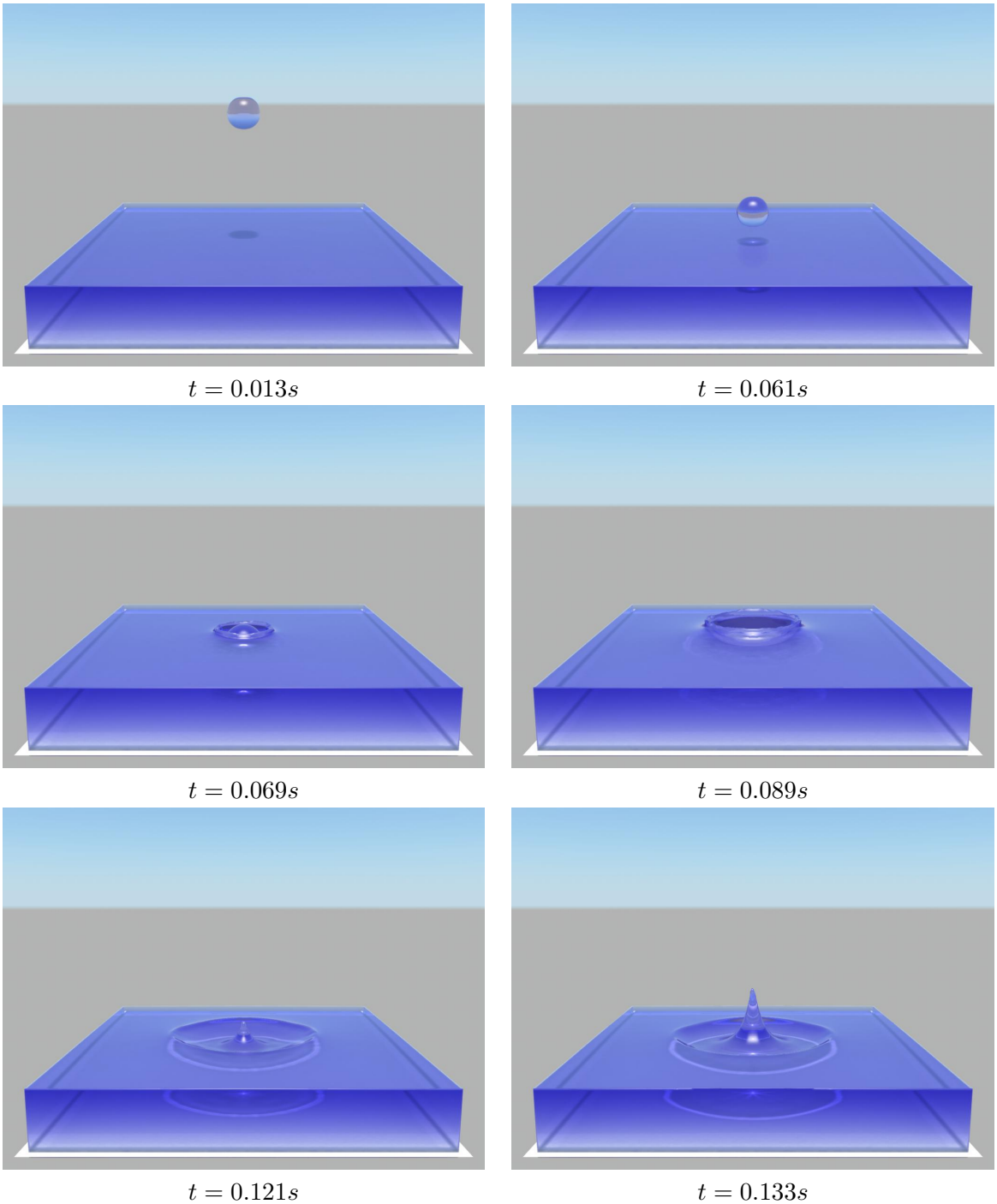


Abb. 8.19: Beim Einschlag des physikalisch korrekt simulierten Tropfens entsteht ein *Rayleigh-Jet.*

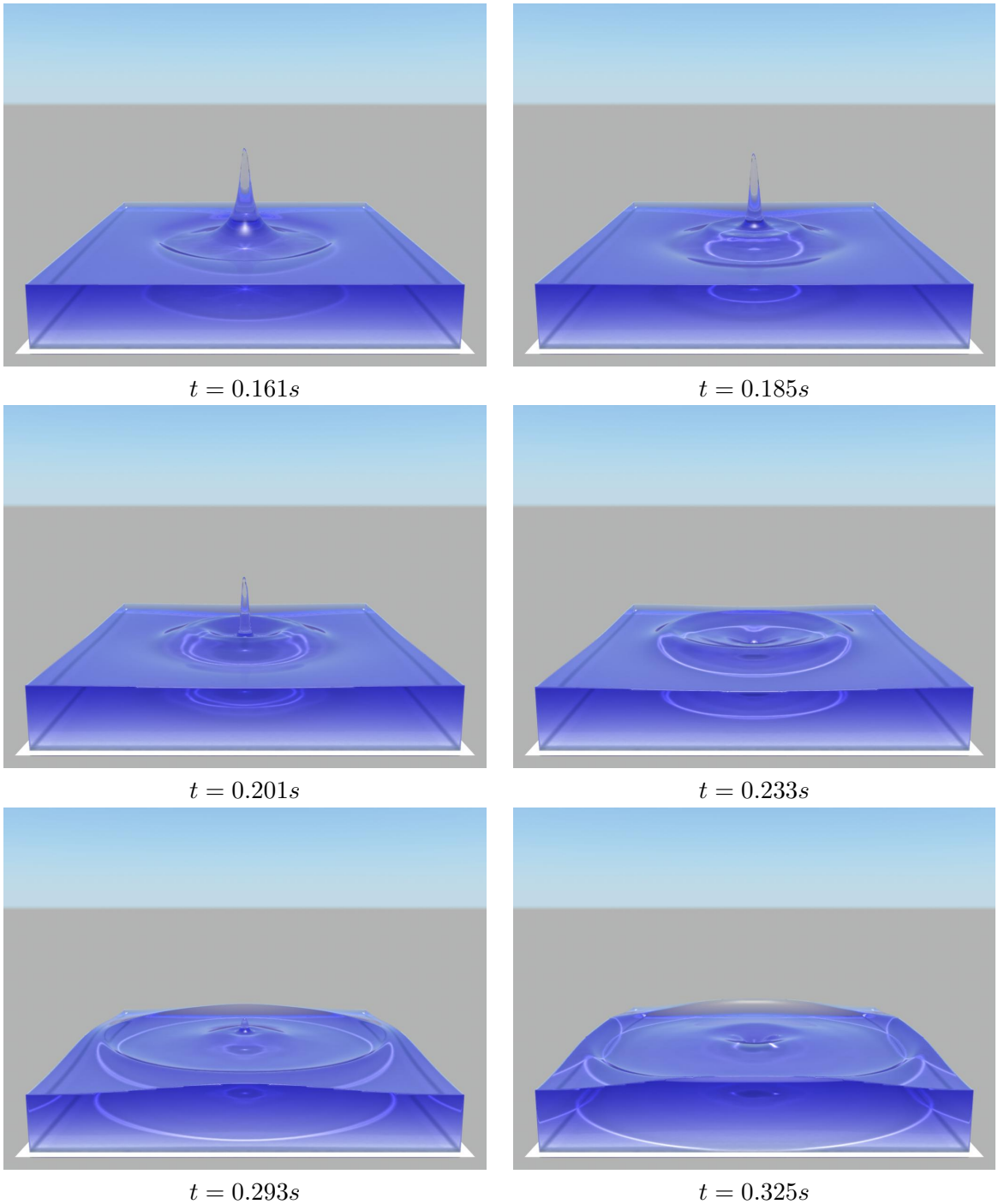


Abb. 8.20: Nachdem der *Rayleigh-Jet* seine maximale Höhe erreicht hat, fällt er in sich zusammen und bring die Wasseroberfläche zum Schwingen.

8.7 Eine aufsteigende Blase

Nachfolgend wird ein Testproblem vorgestellt bei dem in einem Wasserbecken eine große Luft-Blase aufsteigt. Das Problem zeigt auf visuell anspruchsvolle Art die Vorzüge eines vollständig zweiphasig implementierten Strömungslösers. Simuliert man nicht nur die freie Oberfläche der Flüssigkeit (wie in [25]), sondern auch das Gas, lassen sich einige zusätzliche natürliche Effekte nachbilden. Ein Beispiel hierfür sind Blasen im Wasser.

Die grundlegenden Parameter der Simulation sind in Tabelle 8.9 aufgeführt. Eine Modellierung der Simulationsumgebung erfolgt wie bei allen hier vorgestellten Simulationen in Maya. Als Standard-Fluidtyp wird die Flüssigkeit Wasser gewählt. Für die Luftblase wird eine polygonale Sphäre in die Szene eingefügt und über den `SimulationFluidShape`-Knoten als gasförmiges Fluid ausgewiesen.

Größe des Simulationsgebietes:	$20\,cm \times 20\,cm \times 20\,cm$
Auflösung:	$150 \times 150 \times 150$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Initialer Durchmesser der Blase:	$6\,cm$
Initiale Geschwindigkeit der Blase:	$0\,m/s$
Starthöhe der Blase (vom Mittelpunkt aus):	$6\,cm$

Tabelle 8.9: Parameter für die Simulation der aufsteigenden Luftblase.

Man kann Tabelle 8.9 entnehmen, dass im vorliegenden Beispiel eine recht große Luftblase simuliert wird. In der Literatur ist bekannt, dass solche großen Blasen schnell instabil werden [26]. Dementsprechend ist eine erfolgreiche numerische Simulation dieses Effektes eine Herausforderung.

Der Blasenaufstieg benötigt 0.4 Sekunden Simulationszeit und erfordert auf dem Cluster *Himalaya* bei 32 Knoten à 2 Prozessoren eine Laufzeit von 61 Stunden und 23 Minuten. Bei der Berechnung entstehen 3.8 Gigabyte Oberflächendaten, anstatt der vollen Simulationsdaten, die 29 Gigabyte an Speicher benötigen. Das aus der Simulation erzeugte Video zeigt den Vorgang des Blasenaufstiegs in einer 20-fachen Zeitlupe. Die Graphik-Workstation benötigt für seine Erzeugung eine Stunde und 29 Minuten. In Abbildung 8.21 auf Seite 141 werden einige Zeitpunkte der Simulation durch gerenderte Bilder wiedergegeben.

Das erste Bild in Abbildung 8.21 stellt den initialen Zustand der Simulation dar. Schon nach wenigen Millisekunden beginnt sich die Blase erheblich zu deformieren. Zum Zeitpunkt $t = 0.12s$ ist die Blase nur noch ringförmig, hat also ein Loch in der Mitte. Diese Topologieveränderung wird durch die Level-Set–Funktion sehr gut wiedergegeben. In den weiteren Bildern kann man erkennen, dass der Luftring immer weiter in die Breite gezogen wird. Dieses Verhalten entspricht den Beobachtungen von realen Versuchen [54].

Insgesamt zeigen die Ergebnisse des vorliegenden Simulationsbeispiels ganz klar den Vorzug eines physikalisch korrekten zweiphasigen Fluidsimulators. Eine derart realistische Animation des Aufstiegs einer großen Luftblase können nur wenige andere Strömungslöser erzeugen.

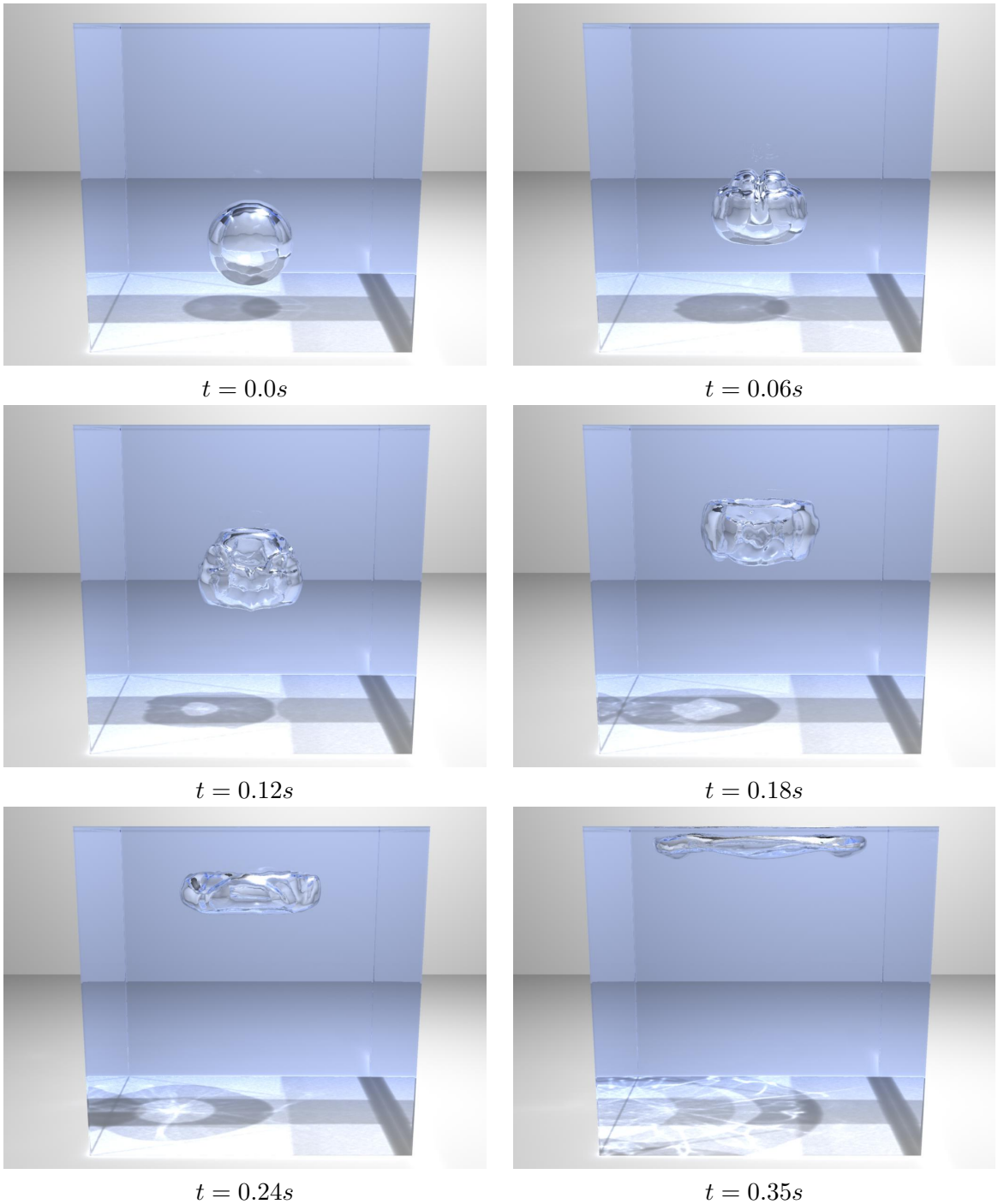


Abb. 8.21: Große Luftblasen im Wasser werden beim Aufstieg instabil.

8.8 Ein schnell fahrendes Boot im Wasser

Das nachfolgende Beispiel stammt aus dem Bereich der ingenieurswissenschaftlichen Anwendungen von Strömungssimulationen. Es soll ein schnell fahrendes Boot im Wasser simuliert werden. Der Hintergrund für eine solche Simulation ist häufig die Analyse und Optimierung der Form von Bootsrümpfen. Ein Ziel entsprechender Optimierungsprozesse besteht in der Reduktion von Verwirbelungen des Wassers entlang des Bootsrumpfes und hinter dem Boot. Solche Verwirbelungen führen häufig dazu, dass die Kraft des Motors bzw. bei Segelbooten die Windkraft nicht optimal in Bewegung übertragen wird.

Für die Simulation eines fahrenden Bootes im Wasser gibt es zwei mögliche Betrachtungsweisen die sich aus der Wahl des Bezugssystems ergeben. Auf der einen Seite kann man das Wasser als fest und das Boot als beweglich betrachten. Die Simulation eines fahrenden Bootes besteht dann darin, dass das Boot mit einer vorgegebenen Geschwindigkeit durch das Wasser bewegt wird. Auf der anderen Seite kann man aber auch die Position des Bootes als fest annehmen und die Fahrt durch den Transport des Wassers nachbilden. Für die zuvor erwähnte Anwendung der Optimierung von Bootsrümpfen wird häufig die zweite Betrachtungsweise gewählt, da so der für den Ingenieur wesentliche Bereich in der Umgebung des Bootsrumpfes stets fest ist und entsprechend einfacher visualisiert werden kann. Ein weiterer Vorteil ist, dass nur eine kleine Umgebung um das Boot simuliert werden muss anstatt die gesamte Region nachzubilden, durch die das Boot fährt.

Im vorliegenden Beispiel wird die zweite Vorgehensweise gewählt: Das Boot hat eine feste Position und das Wasser wird transportiert. Als Bootsmodell wurde ein über polygonale Netze dargestelltes Motorboot von [1] bezogen. Es wird in 1.8 Meter tiefes Wasser gesetzt. Auf einer Seite des Simulationsgebietes wird über einen einfachen Einströmungsbereich Wasser mit einer Geschwindigkeit von $8.33\,m/s$ bzw. $30\,km/h$ in die Simulationsumgebung transportiert. Eine Ausströmungsbedingung auf der gegenüberliegenden Seite lässt das Fluid wieder abfließen. Alle weiteren wichtigen Simulationsparameter finden sich in Tabelle 8.10.

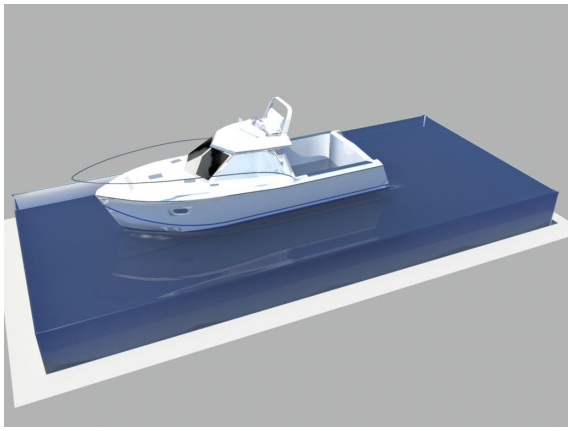
Größe des Simulationsgebietes:	$10\,m \times 8\,m \times 22\,m$
Auflösung:	$180 \times 80 \times 220$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$1.8\,m$
Wassergeschwindigkeit:	$30\,km/h$

Tabelle 8.10: Parameter der Simulation eines schnell fahrenden Bootes.

Simuliert wird eine Zeit von 6 Sekunden. Beim Einsatz von 32 Knoten benötigt das Clustersystem für die Berechnung eine Laufzeit von 114 Stunden und 26 Minuten. Die während der Simulation erzeugten polygonalen Netze belegen ca. 1.2 Gigabyte Speicherplatz. Beim Abspeichern der vollen Simulationsdaten für jeden visualisierten Zeitschritt würden 14 Gigabyte an Daten entstehen. Eine photorealistische Visualisierung der Bootssimulation liegt als Video der elektronischen Ausgabe dieser Arbeit bei. Das Video kann in 42 Minuten auf der Graphik-Workstation gerendert werden.

In Abbildung 8.22 sind Einzelbilder des visualisierten Simulationsresultats dargestellt. Sie geben verschiedene wichtige Zeitpunkte wieder. Das erste Bild zeigt den Zustand kurz nach dem Start der Simulation. Zu diesem Zeitpunkt scheint das Wasser noch zu ruhen. Tatsächlich wurde jedoch die Geschwindigkeit des Einströmungsrandes als initiale Geschwindigkeit des gesamten Wasserblocks gesetzt. Demzufolge bewegt sich das Fluid bereits. Es hat aber noch nicht das für ein fahrendes Boot charakteristische Strömungsprofil.

Nach 0.4 Sekunden kann man erste Deformationen der Wasseroberfläche erkennen. Weitere 0.4 Sekunden später hat sich bereits eine leichte Bugwelle gebildet. Auch die Verwirbelungen am Heck des Motorbootes entstehen. Das Bild zum Zeitpunkt $t = 2.4s$ zeigt den quasistationären Zustand der Strömung der sich bei der Bewegung des Bootes im Wasser entwickelt. Nun hat sich das charakteristische Strömungsprofil ausgebildet. Dementsprechend sollten die Simulationsresultate erst zu diesem Zeitpunkt für eine ingenieurswissenschaftliche Analyse der Strömungsbedingungen eingesetzt werden.



$t = 0.04s$

$t = 0.4s$

$t = 0.8s$

$t = 2.4s$

Abb. 8.22: Ein Boot fährt im Wasser und erzeugt dabei eine Bug- und eine Heckwelle.

8.9 Ein Staudamm aus dem Wasser abgelassen wird

Im nächsten Beispiel soll in einem vereinfachten Modell die Strömungssituation beim Ablassen eines Staudammes nachgebildet werden. Eine solche Simulation ist für Ingenieure von Interesse, die die auftretenden erheblichen Kräfte und die daraus resultierenden Belastungen für den Staudamm studieren wollen.

Das vereinfachte Modell besteht aus einem in Maya modellierten Wasserbehälter, der in der initialen Konfiguration fast vollständig mit Wasser gefüllt ist. An einer Seite des Behälters befinden sich drei große Auslauföffnungen. Davor steht ein Becken mit Wasser. Es ist zu erwarten, dass im Verlauf der Simulation die Schwerkraft das Wasser durch die Öffnungen drückt und der Behälter sich in das Wasserbecken entleert. Die Parameter der Simulation sind in Tabelle 8.11 zusammengefasst.

Größe des Simulationsgebietes:	$24\,m \times 14\,m \times 30\,m$
Auflösung:	$120 \times 70 \times 150$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels im Wasserbehälter:	$11\,m$
Höhe des Wasserspiegels im Auslaufbecken:	$4\,m$

Tabelle 8.11: Parameter für die Simulation des Staudammes aus dem Wasser abgelassen wird.

Simuliert wird eine Zeit von 8.5 Sekunden. Die Berechnung des Ergebnisses erfordert bei dem Einsatz von 64 Prozessoren eine Laufzeit von 3 Stunden und 43 Minuten. Zusammengenommen benötigen die erzeugten Oberflächendaten 2 Gigabyte Speicherplatz im Gegensatz zu den vollen Simulationsdaten die 12 Gigabyte einnehmen würden. Abbildung 8.23 auf Seite 145 zeigt eine perspektivische Ansicht der gerechneten Simulation zu verschiedenen Zeitpunkten. Außerdem ist der elektronischen Ausgabe der Arbeit ein Video in Echtzeit beigefügt. Das photorealistische Rendering der Einzelbilder des Videos benötigt eine Laufzeit von einer Stunde und 38 Minuten.

In den Bildern in Abbildung 8.23 kann man sehr gut erkennen, wie sich der Behälter mit Wasser über die Zeit hinweg entleert. Zum Zeitpunkt $t = 1.33s$ schlagen die Wasserstrahlen in das Wasserbecken ein. Darauffolgend kann man 2.05 Sekunden nach dem Beginn der Simulation erste Turbulenzen im Auslaufbecken erkennen. Über die Zeit hinweg entleert sich der Wasserbehälter immer weiter, während das Wasserbecken immer voller wird. Dies kann man in den beiden letzten Bildern von Abbildung 8.23 sehr gut erkennen.

Alternativ zu den gezeigten Bildern sind in Abbildung 8.24 auf Seite 146 Einzelbilder zu den selben Simulationszeitpunkten aus einer seitlichen Perspektive dargestellt. Dies ermöglicht es, die Vorgänge unter der Wasseroberfläche besser zu erkennen. So sieht man in den Abbildungen für die Zeitpunkte $t = 2.05s$ und $t = 3.45s$ sehr schön, wie das einströmende Wasser Luft mit unter die Wasseroberfläche transportiert. Ebenso lässt sich im Profil der Anstieg des Wasserspiegels besser erkennen. Ein Video der Profilansicht liegt ebenfalls der elektronischen Ausgabe dieser Arbeit bei.

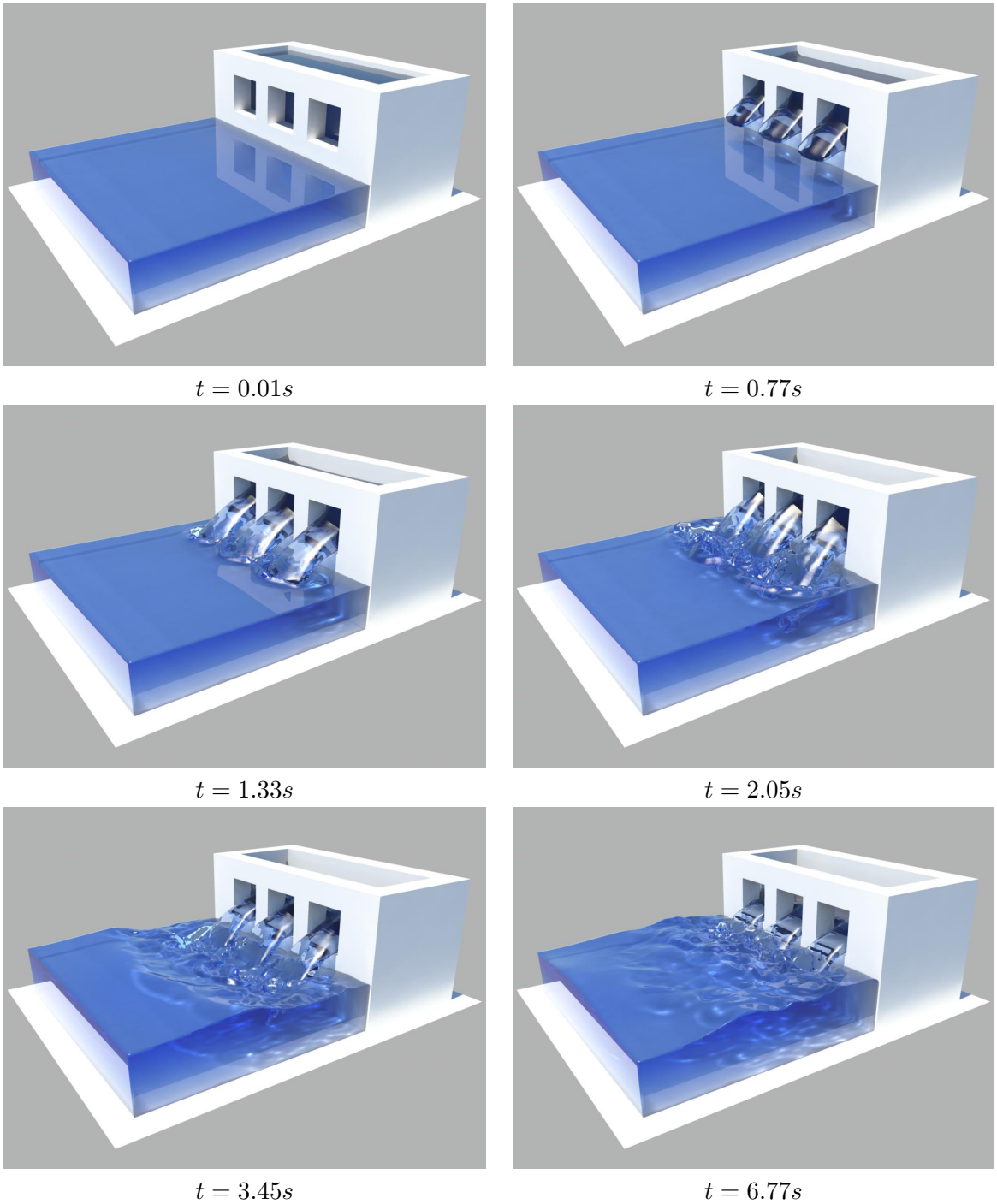
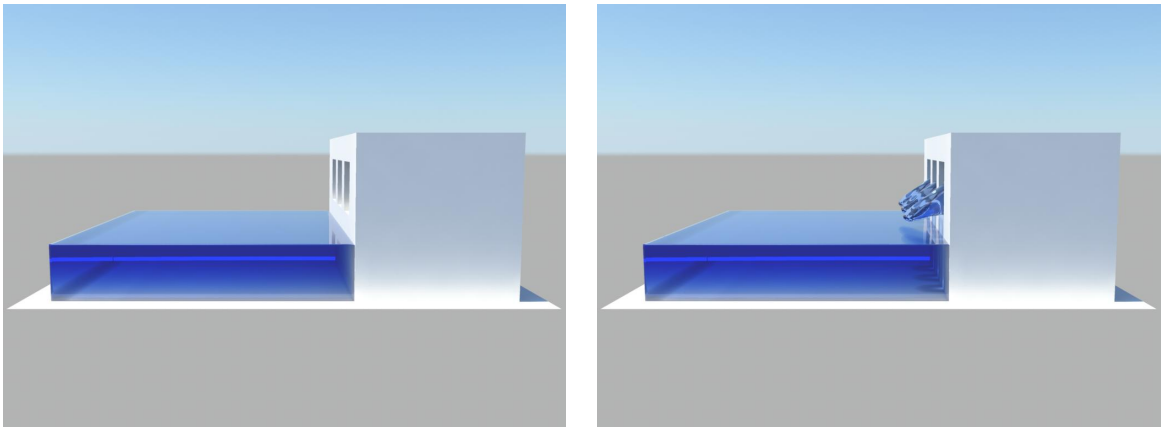
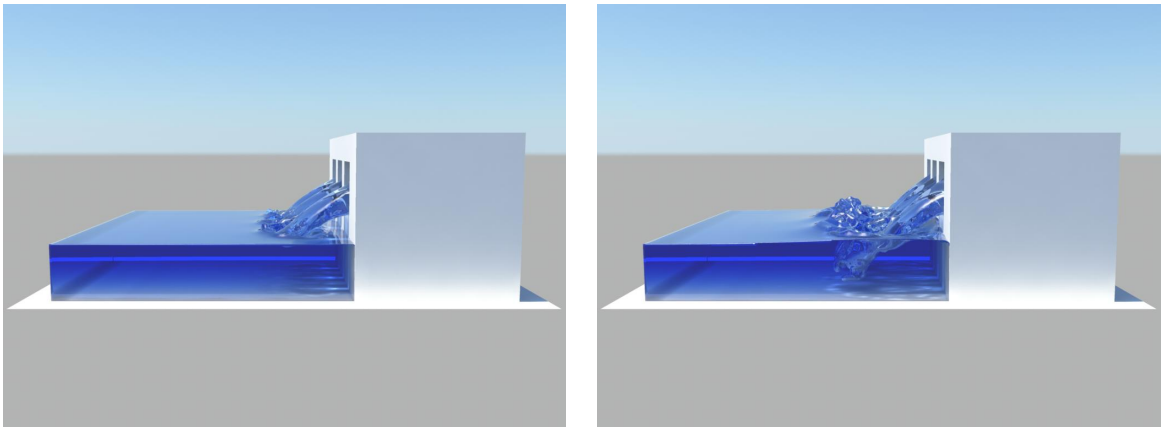


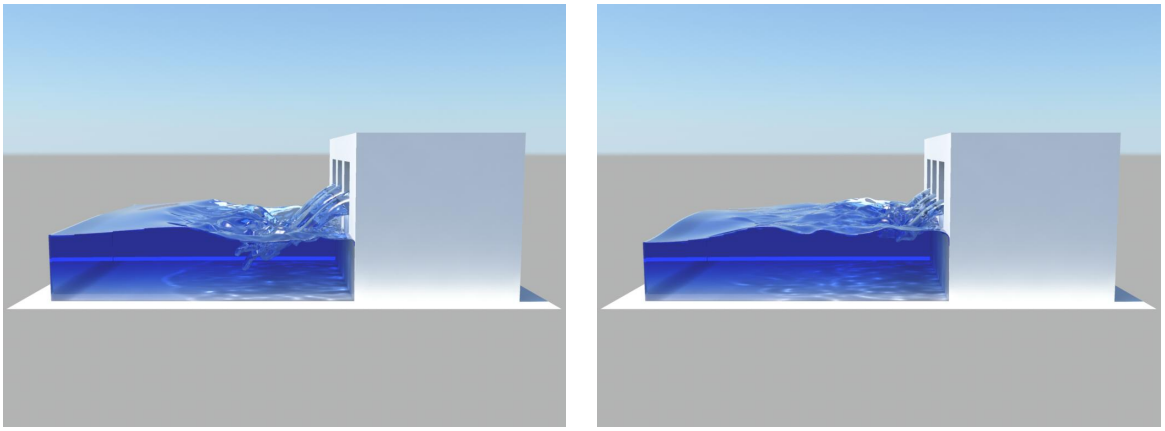
Abb. 8.23: Perspektivische Ansicht der Simulation eines Staudamm-Modelles, aus dem Wasser abgelassen wird.



$t = 0.01s$ $t = 0.77s$



$t = 1.33s$ $t = 2.05s$



$t = 3.45s$ $t = 6.77s$

Abb. 8.24: Profilansicht der Staudammsimulation.

8.10 Ein steiniges Flussbett

Als weiteres ingenieurswissenschaftliches Anwendungsbeispiel wird in diesem Abschnitt die turbulente Strömung in einem Flussbett mit Steinen modellhaft nachgebildet. Strömungsprofile von steinigen Flussbetten sind für die Planung von Uferbefestigungen von Interesse. Mit ähnlichen Simulationen kann man Fluss-Bereiche mit einem erhöhten Sedimentabtrag identifizieren und so Maßnahmen gegen ein Versanden durchführen.

Bei der vorliegenden Simulation wird ein Fluss mit einer Tiefe von 4 Metern nachgebildet. Am Boden werden verteilt liegende Steine durch halbierte Kugeln modelliert. Die Kugeln haben einen Radius von einem Meter. In der initialen Konfiguration wird der Fluss durch einen Wasserquader dargestellt. Das Fließen des Flusses wird wie schon bei der Boot-Simulation durch einen einfachen Einströmungsbereich sowie eine Ausströmregion realisiert. Die Wassergeschwindigkeit beträgt $5 m/s$. Alle weiteren Simulationsparameter sind in Tabelle 8.12 angegeben.

Größe des Simulationsgebietes:	$20\,m \times 10\,m \times 40\,m$
Auflösung:	$100 \times 50 \times 200$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$4\,m$
Höhe der Steine:	$1\,m$
Wassergeschwindigkeit:	$5\,m/s$

Tabelle 8.12: Parameter der Flusssimulation mit einem steinigen Flussbett.

Es werden 100 Sekunden simuliert. Das Clustersystem berechnet diese Zeitspanne auf 64 Prozessoren in außerordentlich guten 22 Stunden und 12 Minuten. Würde man die gesamten Simulationsdaten für jeden visualisierten Zeitschritt speichern, verbrauchten diese ca. 112 Gigabyte an Speicherplatz. Statt dessen benötigen die durch Marching-Cubes–Algorithmus extrahierten polygonalen Netze nur ca. 20 Gigabyte. Das Rendering der 2500 benötigten Einzelbilder für das aus der Simulation erzeugte Video erfordert eine Zeit von 17 Stunden und 51 Minuten. Sechs Einzelbilder des Videos sind in Abbildung 8.25 auf Seite 148 zusammengefasst.

Das erste Bild von Abbildung 8.25 zeigt den Fluss in seiner initialen Konfiguration. Obwohl sich das Wasser schon bewegt, ist die Wasseroberfläche noch flach. In den nächsten beiden Bildern kann man zunächst geringe Deformationen in der Wasseroberfläche erkennen, die sich durch das auf die Steine treffende Wasser ergeben. Nach ca. 19 Sekunden entstehen die ersten turbulenten Verwirbelungen. Sie liegen in der Nähe des Einflussrandes da hier die ersten Steine liegen. Über die Zeit hinweg ($t = 23.01\,s$ und $t = 48.81\,s$) wird das turbulente Wasser weiter zum Ausströmungsrand auf der rechten Seite transportiert. An den Steinen weiter rechts entstehen keine so deutlichen Turbulenzen wie links, da das Wasser beim Vorbeifließen an den ersten Steinen bereits deutlich abgebremst wurde.

Spätestens nach 50 Sekunden ist die gesamte Flussströmung turbulent. Ab diesem Zeitpunkt kann man die Simulation entweder für ingenieurswissenschaftliche Analysen oder für eine sehr gelungene Flussanimation nutzen.

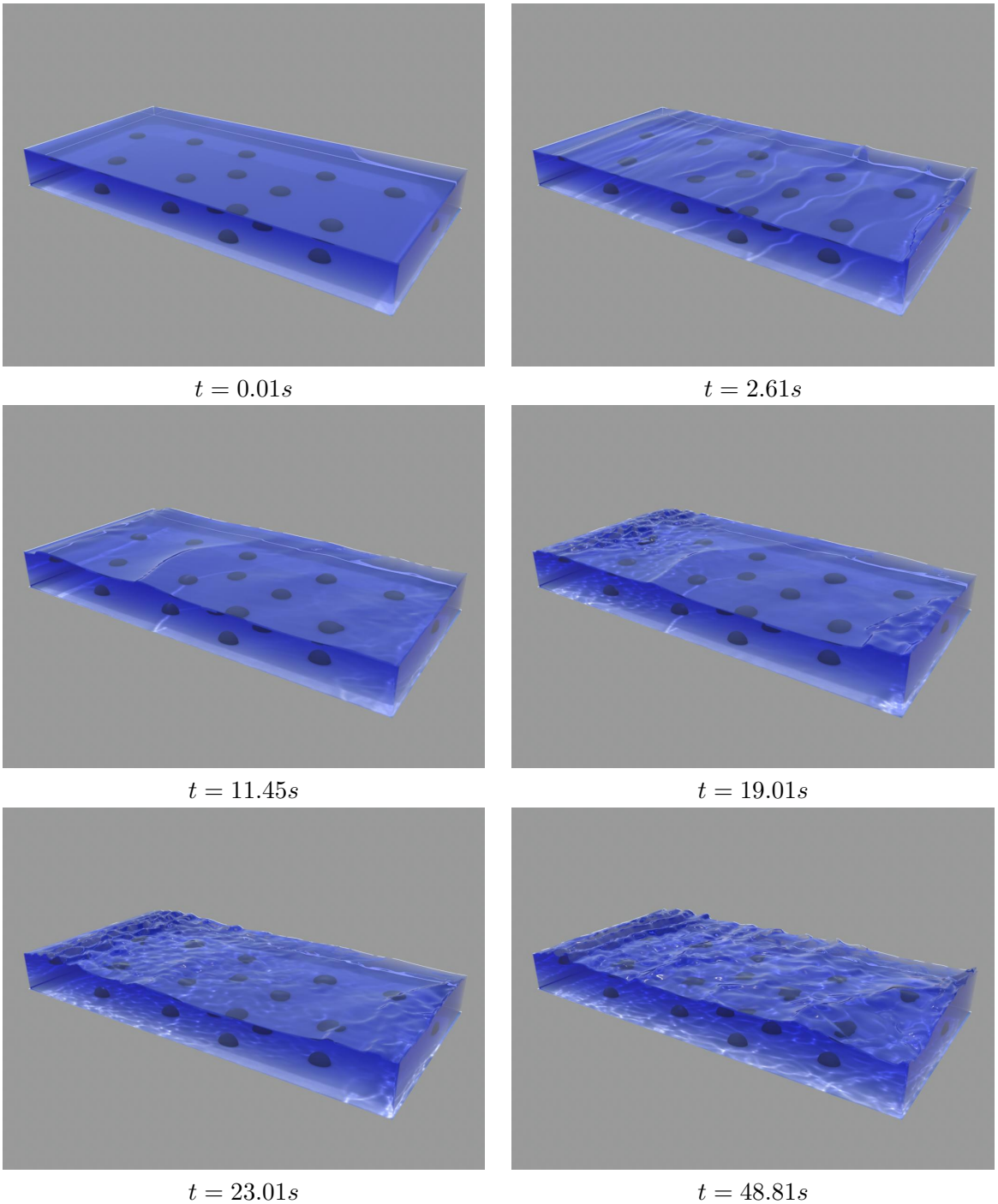


Abb. 8.25: Die Simulation des Flusses mit steinigem Bett wird erst nach einer Anlaufphase richtig turbulent.

8.11 Wasser in einer Gasse

Nachdem in den vorangegangenen Abschnitten Ergebnisse von ingenieurswissenschaftlich motivierten Simulationen vorgestellt wurden, sollen in diesem und den folgenden Abschnitten Simulationen mit einer Ausrichtung auf die Computeranimation diskutiert werden. Den Anfang macht eine Simulation von Wasser, das durch eine Gasse fließt. Eine derartige Animationsaufgabe ist häufig ein Teil größerer Überflutungsszenarien von Städten in Filmen. Abbildung 8.26 zeigt den initialen Zustand der Animationsszene aus der Vogelperspektive.

Abb. 8.26: Der initiale Zustand der Simulation von Wasser in einer Gasse. Ansicht von oben.

In der Abbildung sind die Hindernisbereiche grau und das zu simulierende Wasser blau dargestellt. Zunächst ist das Wasser als flüssiger Quader gegeben, der sich im Verlauf der Simulation in die abknickende Gasse ergießt. Es ist zu beachten, dass die Simulationsaufgabe nicht trivial ist. So kann man erwarten, dass das Wasser gegen die ihm gegenüberliegende Wand prallt. Die große Herausforderung für den Strömungslöser ist dabei die möglichst genaue Berechnung der beim Aufprall auftretenden Kräfte und Geschwindigkeiten.

Alle wichtigen Simulationsparameter kann man Tabelle 8.13 entnehmen. Insbesondere zeigen die dort angegebenen Werte, dass es sich bei der simulierten Gasse um ein kleines Modell der Wirklichkeit handelt.

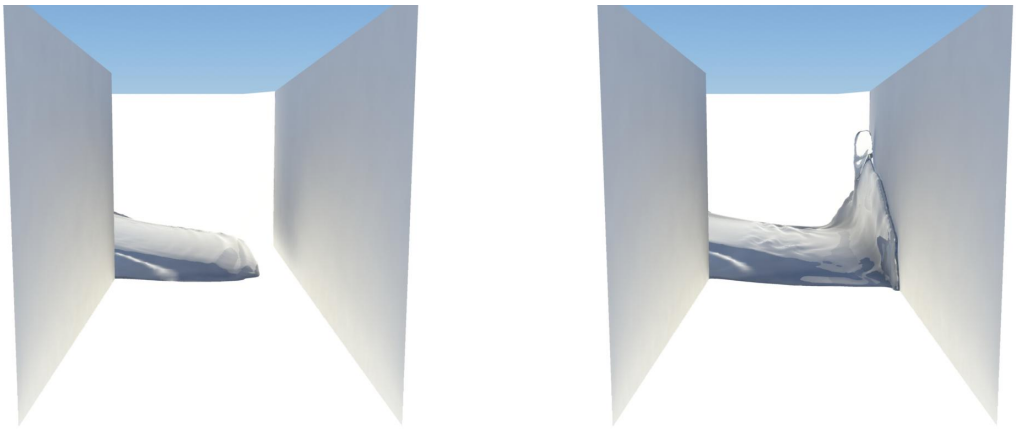
Größe des Simulationsgebietes:	$2.5\,m \times 1.5\,m \times 3.0\,m$
Auflösung:	$166 \times 100 \times 200$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Initiale Höhe des Wasserquaders:	$0.7\,m$

Tabelle 8.13: Simulationsparameter der Simulation von Wasser in einer Gasse.

Die Simulation wird auf dem Cluster *Himalaya* mit 64 Prozessoren durchgeführt. Für die Berechnung der simulierten Zeit von 1.47 Sekunden benötigt dieser 13 Stunden und 51 Minuten. Während der Simulation entstehen 705 Megabyte an Oberflächendaten. Beim Abspeichern der vollen Strömungsdaten würden statt dessen ca. 21 Gigabyte Daten anfallen. Der elektronischen Ausgabe dieser Arbeit liegen zwei Videos der Simulation bei. Ihr Rendering benötigt jeweils ca. einer Stunde. Beide Videos sind in vierfacher Zeitlupe und zeigen die Gasse von einem der beiden Enden her. In den Abbildungen 8.27, 8.28 und 8.29 auf dieser und den beiden folgenden Seiten werden Einzelbilder der beiden Animationen gezeigt.

Im Verlauf der Simulation fließt der initial gegebene Wasserblock wie gewünscht in die Gasse ab. Dies lässt sich sehr gut in der Abbildung 8.29 nachvollziehen. Nach ca. einer halben Sekunde prallt das Wasser auf die gegenüberliegende Wand. Seine große Geschwindigkeit bewirkt, dass es an der Wand hochschnellt. Die maximale Höhe erreicht es nach 0.8 Sekunden. In den Abbildungen 8.27 und 8.28 kann man sehr gut die Dynamik des aufsteigenden Wassers erkennen. Insbesondere zeigen diese Visualisierungen sehr deutlich, wie die Flüssigkeitssäule in der Zeit nach $t = 0.8s$ langsam in sich zusammenfällt und schließlich auf dem Boden aufschlägt.

Insgesamt ist die soeben vorgestellte Simulation eine positive Demonstration für die Arbeit des Navier-Stokes–Lösers, der selbst hochdynamische und turbulente Strömungen wie sie bei dem Aufprall einer Wassermasse auf eine Wand vorkommen sehr Wirklichkeitsgetreu nachbilden kann.



$t = 0.4s$ $t = 0.6s$

Abb. 8.27: Das Wasser trifft beim Abfließen in der Gasse auf eine Wand.

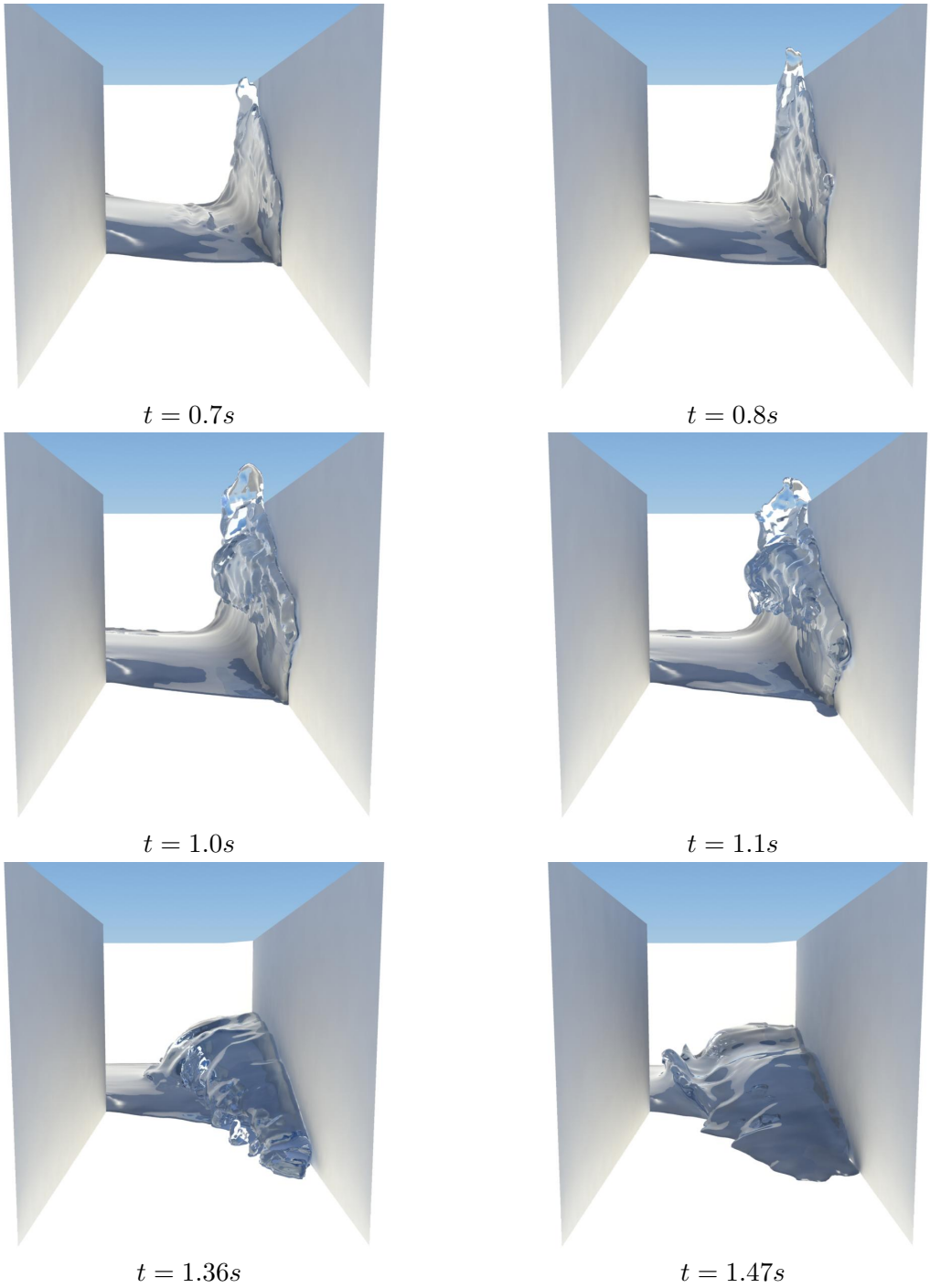
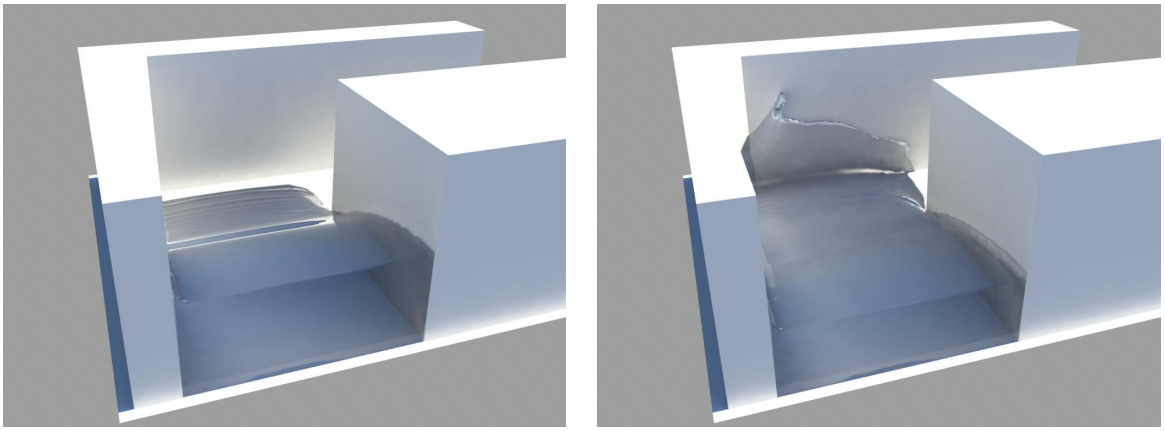
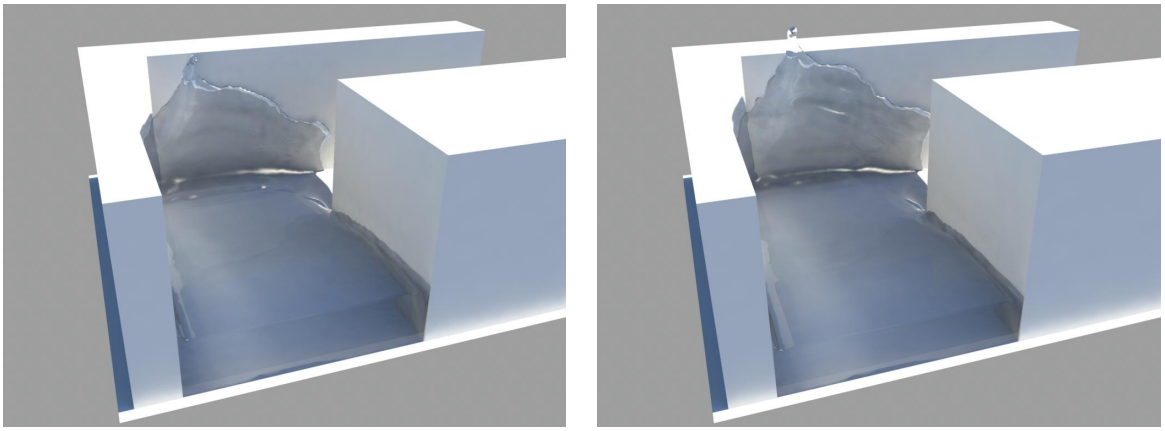


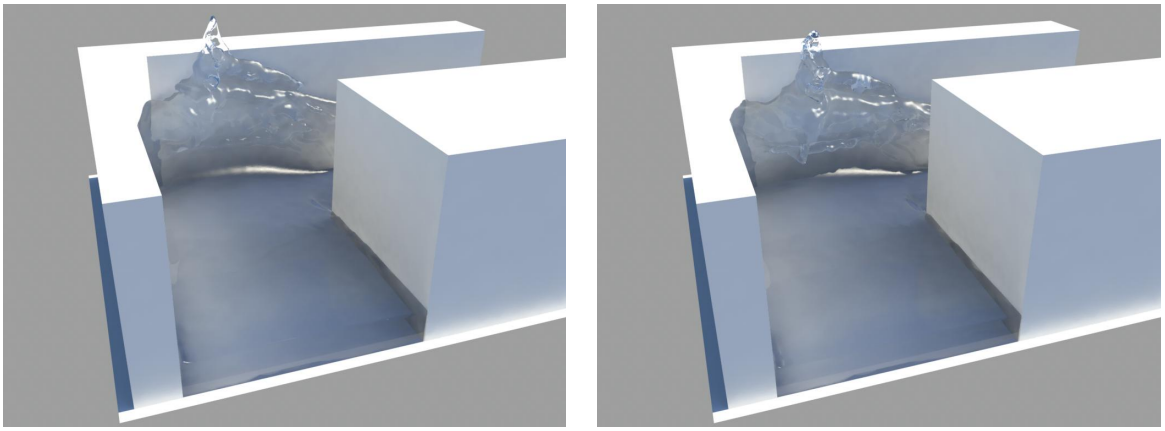
Abb. 8.28: Beim Aufprall auf die Wand schnellt das in der Gasse simulierte Wasser in die Höhe und fällt anschließend in sich zusammen.



$t = 0.4s$ $t = 0.6s$



$t = 0.7s$ $t = 0.8s$



$t = 1.0s$ $t = 1.1s$

Abb. 8.29: Die Wassermenge ergießt sich in die Gasse.

8.12 Die Tower Bridge - Eine großskalige Flusssimulation

In diesem Abschnitt wird die Simulation eines Flusses, der unter einer Brücke her fließt, vorgestellt. Dabei wird in außerordentlich großen Skalen gearbeitet. So wird ein Fluss mit einer Breite von 250 Metern auf einer Länge von 400 Metern bei einer Tiefe von 8 Metern nachgebildet. Die Fließgeschwindigkeit beträgt 4 Meter pro Sekunde. Alle wichtigen Informationen über die Simulation sind in Tabelle 8.14 zusammengefasst.

Größe des Simulationsgebietes:	$400\,m \times 40\,m \times 250\,m$
Auflösung:	$400 \times 40 \times 250$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$8.0\,m$
Fließgeschwindigkeit:	$4.0\,m/s$

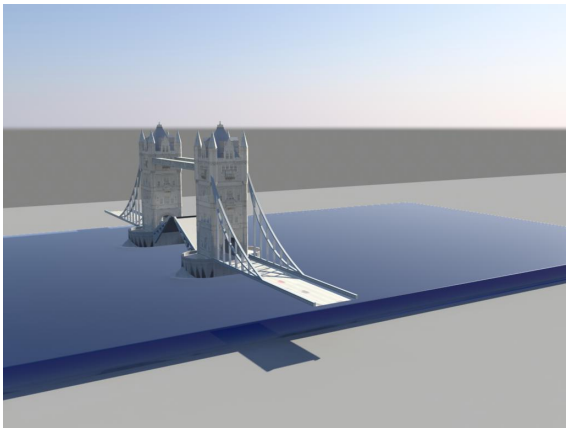
Tabelle 8.14: Parameter der großskaligen Flusssimulation.

Die in der Simulation verwendete Brückengeometrie stellt die Londoner Tower Bridge dar. Sie stammt von [36]. Da das polygonale Netz der Brücke in seiner ursprünglichen Form nicht geschlossen war, wurden die Brückenpfeiler aus der Geometrie extrahiert und einzeln durch Nachmodellieren geschlossen. Nur so ist eine korrekte Voxelisierung als Hindernis möglich.

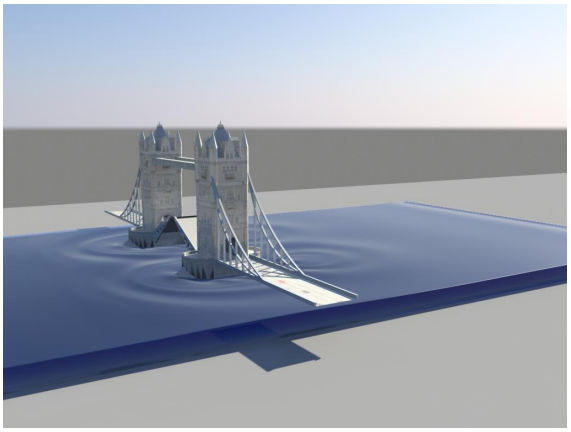
Bei der Konfiguration der Simulationsumgebung erfolgt die Modellierung des Flusses über einen als Fluidobjekt ausgewiesenen polygonalen Quader. Die Bewegung des Gewässers wird über einen einfachen Einströmrand (`SimulationInflowBoundary`) und eine natürliche Ausströmbedingung (`SimulationInOutBoundary`) realisiert.

Simuliert wird eine Sequenz von 50 Sekunden Realzeit. Mit 64 Prozessen benötigt der *Himalaya*-Cluster für die Berechnung sehr gute 7 Stunden und 18 Minuten. Dabei entstehen polygonale Netze im Umfang von 37 Gigabyte. Beim Abspeichern der vollen Simulationsdaten würde man ca. 220 Gigabyte generieren. Das der Arbeit beigefügte Video zeigt das extrem realistische Simulationsresultat. Ein Rendering hiervon benötigt 9 Stunden. Alternativ zur elektronisch beigefügten Animation präsentiert Abbildung 8.30 auf Seite 154 gerenderte Einzelbilder zu verschiedenen Zeitpunkten. Abbildung 8.31 auf Seite 155 zeigt ebenfalls die Simulationsergebnisse, jedoch aus einer anderen Perspektive. Ein Video aus dieser zweiten Perspektive ist ebenfalls verfügbar.

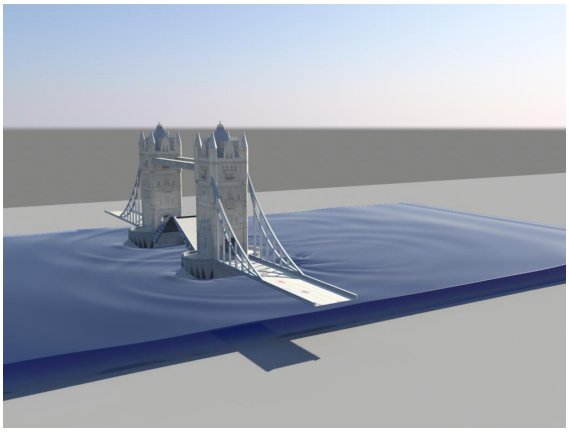
Das erste Bild von Abbildung 8.30 gibt einen vermeindlich stehenden Fluss wieder. Wie schon bei der Simulation des Bootes und des Flusses mit steinigem Flussbett, ist der Fluidblock aber schon zu Beginn der Simulation in Bewegung. Es haben sich nur noch nicht die typischen Verwirbelungen gebildet, die zu einer Deformation der Wasseroberfläche führen. Nach 10 Sekunden entstehen in der Simulation erste Wellen durch das Auftreffen der Strömung auf die Brückenpfeiler. Weitere 10 Sekunden später kommen Verwirbelungen hinter den Pfeilern auf. Deren Transport in Strömungsrichtung lässt sich nach 30 Sekunden erkennen. Gleichzeitig bilden sich immer weitere Strudel, so dass der Fluss nach 50 Sekunden einen turbulenten Zustand im Bereich hinter den Pfeilern erreicht hat. Der quasistationäre Zustand am Ende der Simulationszeit gibt sehr realistisch das tatsächliche Verhalten eines Flusses wieder.



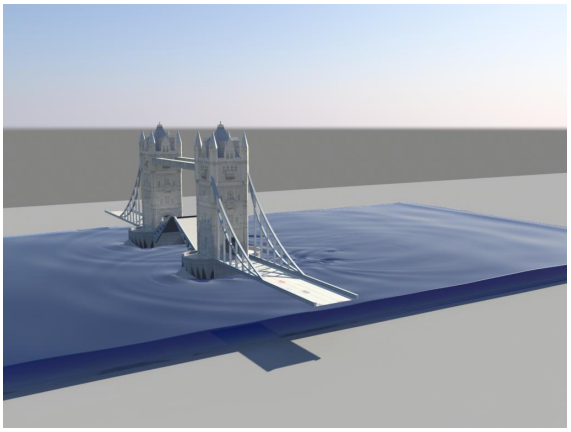
$t = 0s$



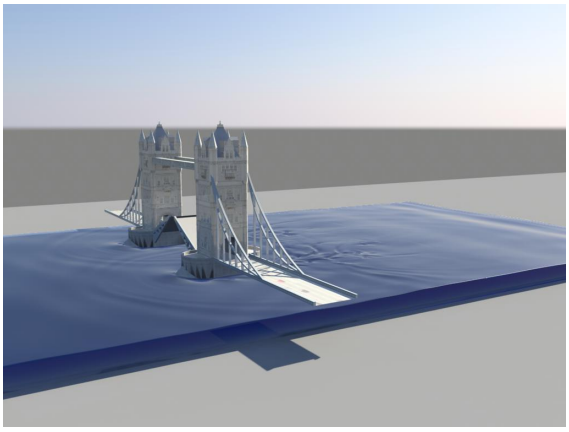
$t = 10s$



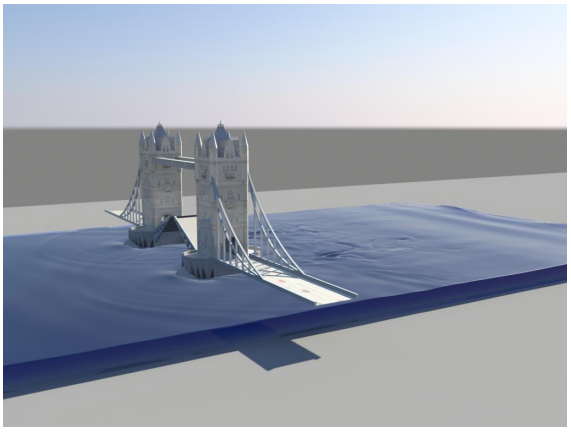
$t = 20s$



$t = 30s$

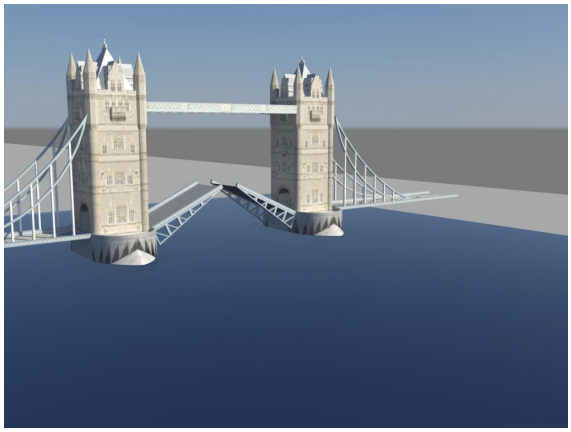


$t = 40s$

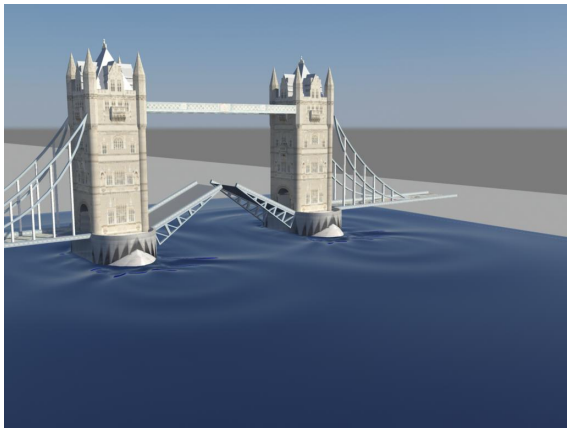


$t = 50s$

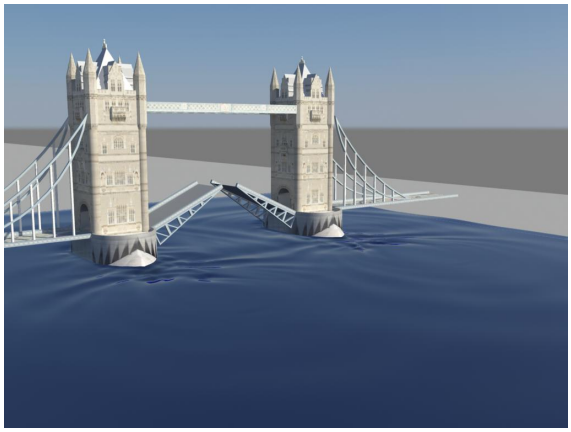
Abb. 8.30: Der simulierte Fluss strömt unter der Tower Bridge her. Dabei entstehen starke Verwirbelungen.



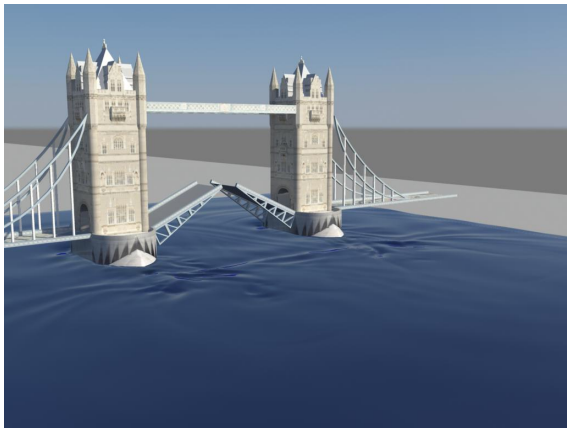
$t = 0s$



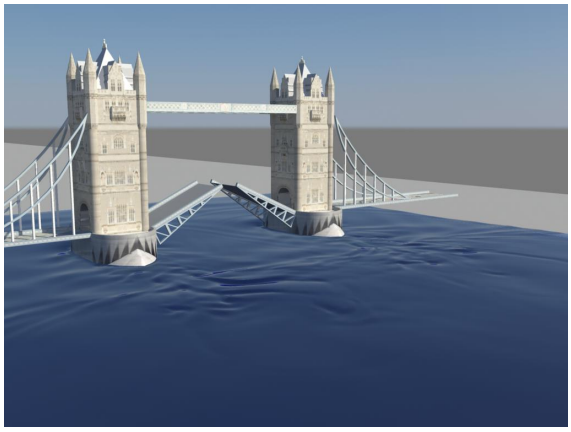
$t = 10s$



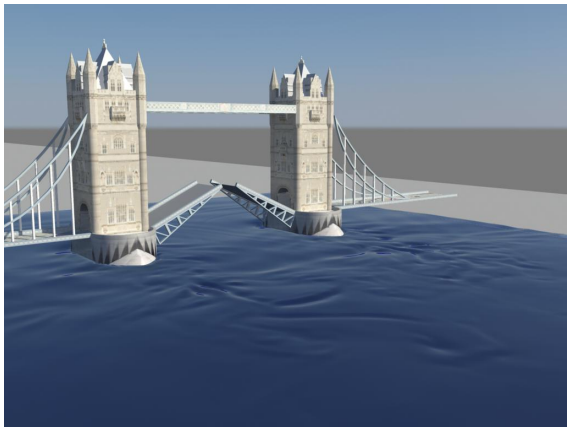
$t = 20s$



$t = 30s$



$t = 40s$



$t = 50s$

Abb. 8.31: Der unter der Tower Bridge her fließende Fluss aus einer anderen Perspektive.

8.13 Ein Leuchtturm bei hohem Wellengang

Die nachfolgend vorgestellte Simulation bildet einen Leuchtturm im Meer nach, gegen den eine große Welle brandet. Mit diesem Beispiel soll der hohe Grad an visuellem Realismus gezeigt werden, der durch den Einsatz des Partikelsystems für Gischtpartikel erreicht werden kann.

Abb. 8.32: Profilansicht der initialen Konfiguration zur Leuchtturmsimulation.

Abbildung 8.32 zeigt eine Profilansicht der initialen Konfiguration der Simulationsumgebung. Am linken Bildrand ist eine Wassersäule zu erkennen. Diese fällt zu Beginn der Simulation in sich zusammen und erzeugt so eine Welle. Die Leuchtturmgeometrie wurde von [1] bezogen. Für eine korrekte Voxelisierung wurde die Grundform der Geometrie nachmodelliert. Das ist notwendig, da das polygonale Netz nicht der Forderung nach einem *geschlossenen* Objekt entspricht. Eine verbesserte Luftzirkulation wird durch eine natürliche Ausströmbedingung am oberen und (bezüglich Abbildung 8.32) rechten Rand der Simulationsumgebung erreicht. Für eine Visualisierung von Gischttropfen wird das an die Level-Set–Funktion gekoppelte Partikelsystem eingesetzt. Die weiteren Simulationsparameter können Tabelle 8.15 entnommen werden. Alle Konfigurationsschritte erfolgen wie gewohnt in Maya.

Größe des Simulationsgebietes:	$60\,m \times 12\,m \times 25\,m$
Auflösung:	$480 \times 96 \times 192$
Flüssig-Phase:	Wasser bei 20^oC
Gas-Phase:	Luft bei 20^oC
Volumenkraft:	Standard-Schwerkraft
Höhe des Wasserspiegels:	$2.0\,m$
Radius der emittierten Tropfen:	$0.005\,m$
`DropRemovalRadiusFactor`	4.0
`InjectionProbabilityFactor`	1000.0
`CurvatureThreshold`	-4.0

Tabelle 8.15: Parameter der Leuchtturmsimulation.

Simuliert wird ein Zeitraum von 7 Sekunden. Auf dem Clustersystem benötigen dafür 64 Prozessoren eine Laufzeit von 46 Stunden und 40 Minuten. Die extrahierten polygonalen Netze beanspruchen einen Speicherplatz von ca. 5 Gigabyte. Würde man wie in der ursprünglichen

Form des Strömungssimulators für jeden Zeitschritt die gesamten Simulationsdaten abspeichern, entstünde eine Datenmenge von ca. 124 Gigabyte. Das aus einer perspektivischen Ansicht photorealistisch gerenderte Video der Simulation ist der Arbeit beigelegt. Die Abbildungen 8.33 und 8.34 auf den Seiten 158 und 159 zeigen Einzelbilder der Animation.

Das erste Bild von Abbildung 8.33 gibt noch einmal den initialen Zustand der Simulation wieder. Nachdem die Wassersäule in sich zusammengefallen ist, erzeugt sie eine Welle ($t = 1.85s$). In der dritten Graphik der Abbildung ist nun zu sehen, wie die Welle auf den Leuchtturm trifft und dabei erste Gischtpartikel entstehen. Die darauffolgend dargestellten Zeitpunkte geben verschiedene Stadien der sich am Leuchtturm brechenden Welle wieder. Insbesondere werden weitere Tropfenpartikel erzeugt. Die visualisierten Partikel approximieren sehr gut die Gischtwolke, die nicht durch die Diskretisierung der Navier-Stokes–Gleichungen aufgelöst werden kann.

In Abbildung 8.34 wird der weitere Fortschritt der Welle dargestellt. Zum Zeitpunkt $t = 3.17s$ fließt die durch den Leuchtturm aufgespaltene Welle wieder zusammen. Die großen Kräfte die sich dabei entfalten erzeugen ebenfalls große Mengen an Gischt. Über die Zeit hinweg fallen die Partikel auf Grund der Schwerkraft ins Wasser und werden entfernt. Gleichzeitig werden aber immer weiter neue Partikel an der Front der Welle erzeugt. Das der Arbeit beigefügte Video zeigt die Leuchtturmsequenz bis die Welle aus dem Bild herausrollt. Abbildung 8.35 auf Seite 161 bietet eine detailreiche, vergrößerte Ansicht des Leuchtturmes nach 2.95 und 3.5 Sekunden.

Mit der Leuchtturmsimulation kann sehr gut der Erfolg des implementierten Gesamtsystems zur Konfiguration und Animation von Fluiden gezeigt werden. Durch die intuitiv bedienbare Benutzerschnittstelle in Maya lässt sich die Simulation in wenigen Minuten zusammenstellen. Der eingesetzte Strömungslöser NaSt3DGPF erzeugt eine sehr realistische Wassersimulation. Ergänzt wird diese durch den neu implementierten Mechanismus zur Approximation von Gischt oder Wassertropfen. Durch die Integration der Fluidoberflächen- und Partikeldaten in Maya kann der leistungsfähige Renderer *Mental Ray* eingesetzt werden. Dieser erzeugt schließlich aus den gegebenen Geometrien und Partikeln sehr realistische Bilder.

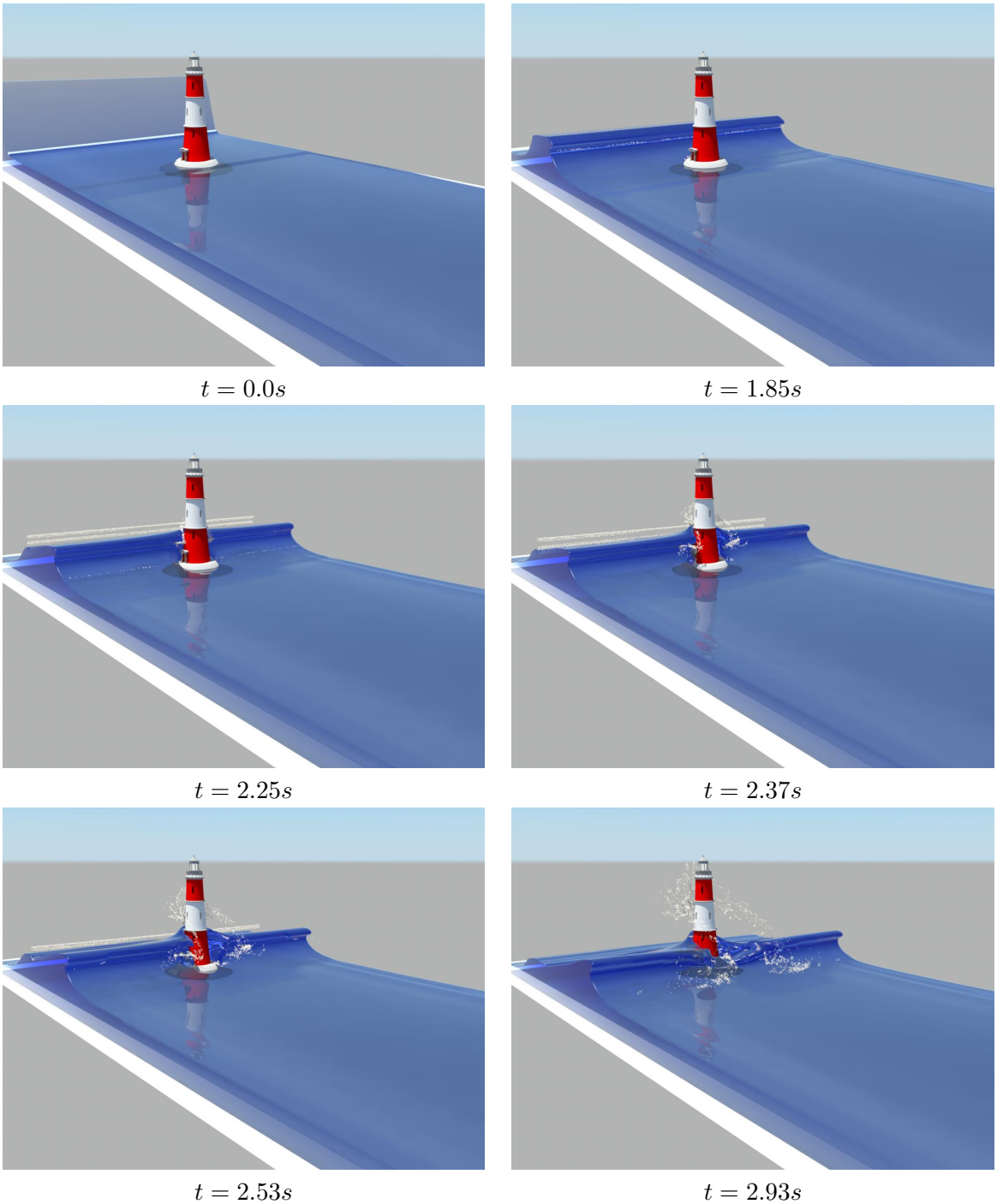
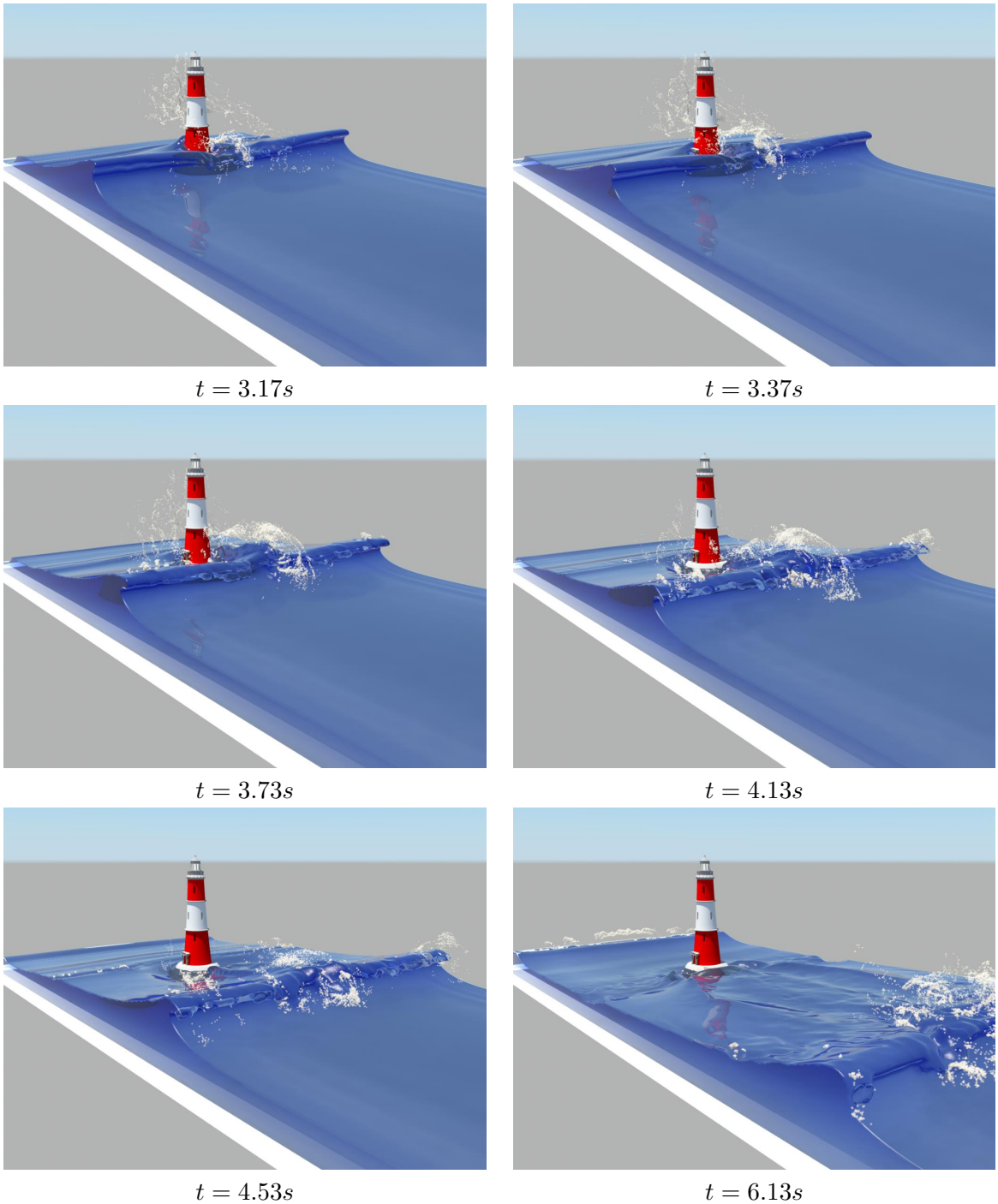


Abb. 8.33: Aus der Wassersäule entsteht eine Welle, die gegen den Leuchtturm brandet und eine Gischtwolke erzeugt.



$t = 3.17s$ $t = 3.37s$

$t = 3.73s$ $t = 4.13s$

$t = 4.53s$ $t = 6.13s$

Abb. 8.34: Hinter dem Leuchtturm rollt die Welle weiter. Dabei entsteht weiterhin Gischt.

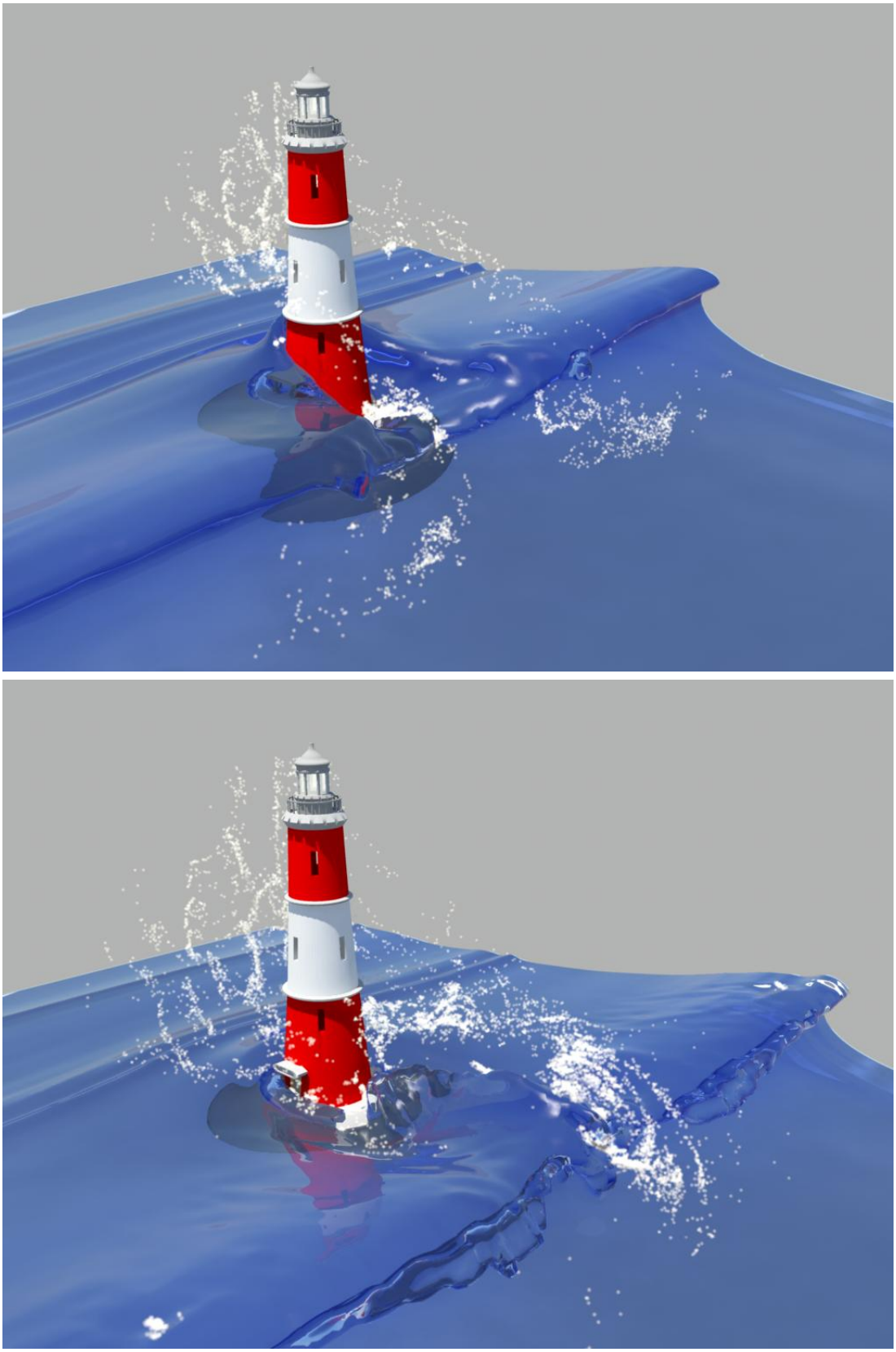


Abb. 8.35: Die nachgebildete Gischt wirkt sehr realistisch.

9 Zusammenfassung und Ausblick

Zusammenfassung

Im Rahmen dieser Arbeit wurde mit dem dreidimensionalen zweiphasigen Navier-Stokes–Löser NaSt3DGPF erstmalig ein Strömungslöser mit einem sehr geringen Modell- und Diskretisierungsfehler in das 3D-Animationsprogramm Autodesk Maya eingebettet. Hierdurch kann der vollständige Verarbeitungsablauf einer Strömungssimulation, bestehend aus der Konfiguration, Simulation und Visualisierung im Sinne eines *virtuellen Labors* interaktiv in Maya durchgeführt werden. Alle unnötigen manuellen Arbeiten werden dem Benutzer durch vollautomatische Mechanismen abgenommen. Bei der Visualisierung wird Photorealismus für eine möglichst allgemeinverständliche Aufbereitung der Graphiken verwendet.

Eine Anbindung des Strömungslösers an Maya ermöglicht die Erstellung von Fluidanimationen mit einem nie dagewesenen geringen Modellfehler. Durch eine neu entwickelte Zwei-Wege-Kopplung der Level-Set–Methode mit einem Partikelsystem für massebehaftete Partikel werden selbst kleinste Details einer Fluidanimation durch eine hybride Gitter- und Partikel-Diskretisierung aufgelöst und visualisiert. Insbesondere dient das Partikelsystem der Animation von Wassertropfen und Gischt-Spritzern und der Interaktion mit der freien Oberfläche.

Für die Realisierung der benannten Aufgaben wurden mehrere Maya-Plugins entwickelt und entscheidende Erweiterungen für den parallelen Strömungslöser umgesetzt. Eine Verwendung der Maya-typischen Funktionsbausteine wie Knoten, Kommandos, Manipulatoren oder Templates für den *Attribute Editor* führt zu einem Programmdesign, das für Benutzer von Maya sehr leicht verständlich ist. Man kann hier fast schon von einer *Simulationsumgebung für Jedermann* sprechen.

Die erste Aufgabe der vorliegenden Arbeit bestand in der Entwicklung von einer interaktiven Konfigurationsumgebung für Strömungssimulationen in Maya. Es musste zudem ein Weg gefunden werden, wie aus den in der Umgebung gesammelten Informationen eine Szenenbeschreibung für den Strömungslöser NaSt3DGPF erzeugt werden kann. Beide Aufgaben konnten durch die Umsetzung von Plugins für Maya erreicht werden. Mit der entwickelten Konfigurationsumgebung ist nun ein effizientes und äußerst benutzerfreundliches Arbeiten möglich. Die Szenenbeschreibung wird ohne jeden manuellen Eingriff automatisiert aus Maya heraus erstellt. Für die Generierung von diskretisierten Fluid- und Hindernisgeometrien wurden einige hochspezialisierte Voxelisierungsverfahren entwickelt und implementiert. Das Canyon-Beispiel im Ergebniskapitel konnte zeigen, dass sich nun für Fluidsimulationen selbst komplizierteste Hindernisgeometrien und Einflussbereiche mit nur wenigen Mausklicks zusammenstellen lassen.

Im Allgemeinen stellt die Visualisierung von dreidimensionalen zeitabhängigen Daten einer Strömungssimulation auf Grund des immensen Speicherbedarfes ein großes Problem dar. Mit dieser Arbeit wurde das Problem bezüglich der Visualisierung der freien Oberfläche gelöst. Ein hocheffizientes paralleles Isoflächenextraktionsverfahren auf Basis des Marching-Cubes–

Algorithmus wurde in den parallelen Navier-Stokes–Löser implementiert. Das Verfahren generiert vollautomatisch, zeitgleich mit der Simulation polygonale Netze von der Fluidoberfläche. Ein Speichern der umfangreichen Simulationsdaten kann so entfallen. Durch ein Benchmarking-Problem sowie verschiedene Simulationstestfälle konnten die Effizienz der Implementierung und der eingesparte Speicherplatz nachgewiesen werden.

Die extrahierten zeitabhängigen Oberflächendaten müssen zur Visualisierung in Autodesk Maya eingelesen werden. Ein manuelles Laden der polygonalen Netze verbietet sich jedoch auf Grund ihrer großen Anzahl. Zudem ist ein zeitgleiches Verwalten von zum Teil mehreren tausend hochaufgelösten Meshes in einer Maya-Szene mit aktuellen Workstations nicht möglich. Diese Probleme wurden im Rahmen der vorliegenden Arbeit ebenfalls mit einem Maya-Plugin gelöst. Es liest vollautomatisch das jeweils aktuell benötigte polygonale Netz in eine Szene ein und stellt es wie jedes gewöhnliche statische Mesh dar. Für den Benutzer ermöglicht dieses hocheffiziente Vorgehen ein sehr einfaches und komfortables Arbeiten.

Der Einsatz der Level-Set–Funktion als Fluidoberfläche führt im Zusammenhang mit der Animation zu Schwierigkeiten. So ruft die Bindung der Funktion an die Diskretisierungsgenauigkeit des Strömungslösers zum Teil visuell unerwünschte Effekte hervor, wenn Fluidfeatures wie Tropfen oder Gischt nicht mehr aufgelöst werden können und folglich verschwinden. Diesem Diskretisierungsartefakt wurde in der vorliegenden Arbeit durch die Zwei-Wege-Kopplung der Level-Set–Funktion an ein Partikelsystem entgegengetreten. Sphärenförmige Tropfen werden als Partikel diskretisiert und in einem System für massebehaftete Partikel über ein physikalisch motiviertes Transportmodell bewegt. Das Partikelsystem wurde in dieser Arbeit in den Löser NaSt3DGPF implementiert und parallelisiert.

Für die Kopplung der Partikel an die Level-Set–Funktion wurde ein vollautomatisches Verfahren entwickelt, das stets dann Tropfen in das Partikelsystem emittiert, wenn sich diese normalerweise von der Fluidoberfläche ablösen. Die Rückkopplung der diskreten Partikel an die gitterbasiert diskretisierte Level-Set–Funktion besteht in der Modellierung von Tropfeneinschlägen und den daraus resultierenden Deformationen der freien Oberfläche. Es wurde ein einfaches und effizientes Modell entwickelt, mit dem die bei einem Tropfeneinschlag entstehenden Kräfte approximiert werden können. Implementiert wurde die Rückkopplung durch eine Erweiterung der diskretisierten Navier-Stokes–Gleichungen.

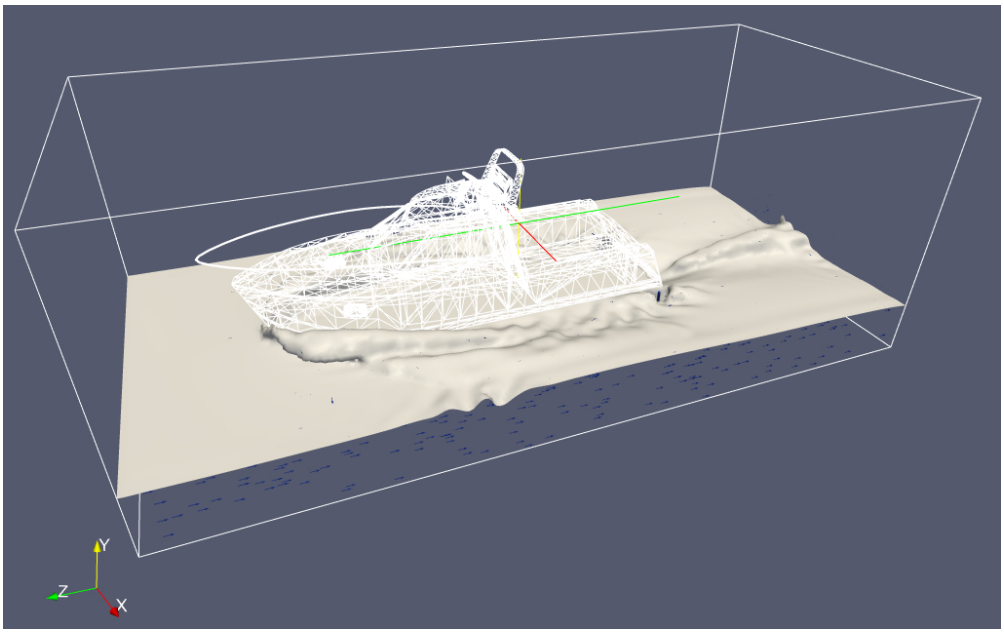
Die durch Partikel modellierten Tropfen sollen ebenfalls in Maya visualisiert werden. Hierfür wurde ein Plugin zum vollautomatischen Einlesen von zeitabhängigen Partikelpositionsdaten entwickelt. Das Plugin nutzt die Visualisierungsfähigkeiten des in Maya eingebauten Partikelsystems und ermöglicht somit dem geübten Maya-Nutzer ein direktes Arbeiten ohne jede Einarbeitung.

In einigen Beispielen konnte im Ergebniskapitel dieser Arbeit der verbesserte visuelle Eindruck beim Einsatz der Kopplung der Level-Set–Funktion mit einem Partikelsystem gezeigt werden. Für die Tropfenemission sei hier stellvertretend die Simulation eines Wasserstrahls, der auf ein Wasserbecken auftrifft, genannt. Die Modellierung der Tropfeneinschläge in die freie Oberfläche führte zu der erwünschten Wellenbildung um die Einschlagsstellen. Animationen von Regen lieferten überzeugende Ergebnisse und untermauerten so die positiven Resultate.

Verschiedene Simulationen konnten zeigen, dass gerade der Einsatz von einem zweiphasigen physikalisch korrekten Navier-Stokes–Löser von hohem Interesse für den Bereich der Fluidanimation sein kann. So konnten Effekte wie der Aufstieg einer Luftblase im Wasser und der

Rayleigh-Jet nach dem Einschlag eines vollständig simulierten Tropfens erfolgreich und äußerst realistisch nachgebildet werden.

Mehrere ingenieurswissenschaftlich motivierte Simulationen unterstrichen die in der Einleitung gemachte Aussage, dass Photorealismus zu einem erhöhten Verständnis wissenschaftlicher Ergebnisse für die Allgemeinheit führt. Hier seien die Simulation eines Staudammes, der abgelassen wird, die Simulation eines schnell fahrenden Bootes sowie die Simulation eines Flusses, der über ein steiniges Flussbett fließt, erwähnt. Die beiden nachfolgenden Abbildungen zeigen noch einmal die Unterschiede zwischen der üblicherweise verwendeten wissenschaftlichen Visualisierung und dem hier eingesetzten allgemeinverständlichen Ansatz.



Das sehr gute Zusammenspiel der implementierten Komponenten für die Animation konnte anhand mehrerer 3D-Szenen demonstriert werden. Die Simulation von Wasser, das durch eine Gasse fließt, stand stellvertretend für Überschwemmungsszenarien. Mit der Simulation eines Flusses, der unter der Tower Bridge her strömt, wurde eine außerordentlich großskalige Animationsaufgabe erfolgreich absolviert. Als weitere umfangreiche Simulation entstand eine realistisch animierte Szene von einer Welle, die gegen einen Leuchtturm prallt und dabei Gischt erzeugt.

Im Gegensatz zu anderen Publikationen wurde in der vorliegenden Arbeit eine hohe Zahl an verschiedenen erfolgreichen Simulationen gezeigt. Dieses beweist nicht nur, dass das eingesetzte Simulationssystem sehr zuverlässig arbeitet, sondern auch, dass durch die einfache Konfiguration und Bedienung des Strömungslösers NaSt3DGPF in Maya sowie dem hohen Grad an Automatisierung die Arbeitsabläufe bei einer Simulation sehr effizient sind und somit genügend Zeit für verschiedene Simulationen bleibt.

Ausblick

Das im Rahmen dieser Arbeit entwickelte Framework zur Konfiguration, Visualisierung und Animation bildet eine solide Basis für zukünftige Erweiterungen.

Es wurden noch nicht alle Funktionen des Strömungslösers in Autodesk Maya integriert. So kann dieser als weitere Größe auch die Temperatur mitberechnen. Der durch Temperaturunterschiede angeregte Fluidtransport könnte einen visuell interessanten Effekt darstellen. Eine weitere Funktionalität des Strömungssimulators ist die Berechnung der Ausbreitung von Chemikalien. Diese könnte derart genutzt werden, dass ein volumetrisches Rendering von im Wasser schwimmenden Sedimenten möglich wird. Des Weiteren steht bereits im NaSt3DGPF-Löser ein Verfahren zur adaptiven Verfeinerung der Level-Set–Funktion zur Verfügung [8]. Wegen der besseren Approximation der freien Oberfläche, sollte eine noch bessere Visualisierung realisierbar sein.

Außerdem wären einige weitere Arbeiten im Bereich der Voxelisierung sinnvoll. Die bisher umgesetzten Verfahren zur Voxelisierung der Hindernis- und Fluid-Geometrien auf der Basis von Strahlenschnitten sollten durch effizientere Algorithmen ersetzt werden, damit die Konfiguration von Fluidsimulationen noch schneller durchgeführt werden kann.

Soll die entwickelte Einbettung des Strömungssimulators auch für wissenschaftliche Visualisierungen genutzt werden, wäre eine Erweiterung von Maya um geeignete Module für die Darstellung von Vektorfeldern oder Skalarfeldern denkbar.

Ein anderer Ansatzpunkt für zukünftige Entwicklungen ist der Löser NaSt3DGPF. Soll bei diesem eine stärkere Ausrichtung auf die Animation stattfinden, scheint eine Integration der Partikel-Level-Set–Methode von Fedkiw et al. [25], [27] ein erster wichtiger Schritt zu sein. Mit diesem Verfahren sollten sich dünne Fluid-Schichten besser darstellen lassen. Gleichzeitig könnte man wie in [25] die Partikel der Level-Set–Funktion nutzen um zusätzliche Tropfeneffekte zu realisieren. Dabei wäre es aber wichtig, den außerordentlich geringen Modell- und Diskretisierungsfehler des Strömungssimulators zu erhalten.

Ganz allgemein hat sich bei der Arbeit mit dem Strömungslöser NaSt3DGPF gezeigt, dass dieser in Anbetracht seiner sehr guten Simulationsergebnisse vertretbare Laufzeiten hat. Eine Verbesserung der Laufzeiten, ohne dabei die Anforderungen an den Modell- und Diskretisierungsfehler zu relaxieren, wäre aber dennoch wünschenswert und würde sowohl der Fluid-

Animation als auch allen anderen ingenieurswissenschaftlichen Anwendungen des Lösers zugute kommen. Hier könnten die zur Zeit aufkommenden massiv parallelen Prozessoren aus dem Bereich der Graphikhardware von Interesse sein. So sollten die Geschwindigkeitsvorteile bei der Auslagerung bestimmter numerischer Verfahren des Lösers auf GPU-basierte Systeme überprüft werden.

Insgesamt steckt im Strömungslöser NaSt3DGPF und in der in dieser Arbeit entwickelten Anbindung an Autodesk Maya ein großes Potential für künftige Erweiterungen. Mit weiterem Engagement in diesem Bereich sollte es möglich sein, sowohl das *virtuelle Labor* in Maya zu vervollständigen, als auch einen Platz unter den führenden Programmpaketen für die Fluidanimation einzunehmen.

An dieser Stelle möchte ich die Gelegenheit nutzen, mich bei allen zu bedanken, die zum Entstehen dieser Arbeit beigetragen haben. Insbesondere danke ich Herrn Prof. Dr. M. Griebel für die Überlassung des Themas und seine engagierte Unterstützung. Viele wertvolle Ratschläge seiner Mitarbeiter haben mir das Arbeiten sehr erleichtert. Außerdem bin ich allen dankbar, die diese Arbeit Korrektur gelesen haben. Ein besonderer Dank gilt meinen Eltern für ihre Unterstützung und ihr Verständnis.

A Parameter für den Strömungslöser

Nachfolgend werden alle in dieser Arbeit neu eingeführten Konfigurationsparameter für den NaSt3DGPF-Löser vorgestellt.

Parameter	Datentyp	Standardwert	Beschreibung
`WriteMesh`	boolean	OFF	Extraktion der freien Oberfläche als polygonale Netze aktivieren / deaktivieren
`MeshesPerSecond`	integer	0	Anzahl der zu extrahierenden Oberflächen pro simulierter Sekunde (0 ⇒ Oberflächen werden für jeden Zeitschritt geschrieben); wenn `UseDropSystem` aktiviert ist, wirkt dieser Parameter analog auf die Ausgabe der Tropfen
`MeshOutputDirectory`	string	”/tmp”	Ziel-Verzeichnis für die polygonalen Netze
`MeshClosure`	boolean	OFF	Generierung der Randflächen an- bzw. abschalten
`MeshScaleFactor`	double	1.0	Faktor, um den die Weltkoordinaten der polygonalen Netze skaliert werden sollen
`UseDropSystem`	boolean	OFF	Partikelsystem für massebehaftete Partikel an-/abschalten
`DropsOutputDirectory`	string	”/tmp”	Ziel-Verzeichnis für die Ausgabe der Tropfen-Positionen und -Radien
`DropInjectionFile`	string	” ”	Datei mit Tropfenstart-Informationen vorgeben

Parameter	Datentyp	Standardwert	Beschreibung
`DropsScaleFactor`	double	1.0	Faktor, um den die Weltkoordinaten der Tropfen-Positionen und die Radien der Tropfen bei der Ausgabe skaliert werden sollen (analog zu `MeshScaleFactor`)
`DropRemovalRadiusFactor`	double	10.0	Sicherheitsfaktor für die Entfernung von Tropfen nach dem Auftreffen auf die *Wasseroberfläche* (siehe Abschnitt 7.1.3)
`InjectDrops`	boolean	OFF	automatisierte Tropfenerzeugung aktivieren / deaktivieren
`InjectionDropSize`	double	0.005	Radius der automatisch generierten Tropfen
`InjectionProbabilityFactor`	double	400.0	Wahrscheinlichkeitsfaktor für Tropfenerzeugung (siehe Abschnitt 7.2.1)
`CurvatureThreshold`	double	-10.0	Schwellenwert für die Krümmung, ab der Tropfen erzeugt werden sollen; entspricht κ_{thresh} (siehe Abschnitt 7.2.1)
`ComputeDropImpact`	boolean	OFF	Berechnung der Tropfeneinschläge an- bzw. abschalten
`ForceFactor`	double	400.0	Verstärkungsfaktor für die Einschlagskraft; entspricht ω_{opt} (siehe Abschnitt 7.2.2)
`timeveloFromFile`	string	" "	Zusatzoption für `inflow`-Objekte; Angabe einer Datei mit zeitabhängigen Geschwindigkeitsdaten

Literaturverzeichnis

[1] *Webseite 3D-CAD-Browser: Kaufplattform für 3D-Geometrien.* `http://www.3dcadbrowser.com` (letzter Zugriff 17.03.2009).

[2] Angermann, L. und P. Knabner: *Numerik partieller Differentialgleichungen.* Springer, Berlin, Heidelberg, New York, 2000.

[3] *Webseite zu ANSYS CFX.* `http://www.ansys.com/Products/cfx.asp` (letzter Zugriff 19.03.2009).

[4] Bateman, R.: *Persönliche Homepage von Robert Bateman.* `http://www.robthebloke.org` (letzter Zugriff 17.02.2009).

[5] Becker, H.: *DeViSor - Entwurf und Implementierung einer graphischen-interaktiven Steuerungs- und Visualisierungsumgebung für numerische Simulationssysteme.* Technischer Bericht, Universität Dortmund, Fachbereich Informatik, 2004.

[6] *Webseite zum freien 3D-Animationsprogramm Blender.* `http://www.blender.org` (letzter Zugriff 21.03.2009).

[7] Bridson, R. und M. Müller-Fischer: *Fluid simulation: SIGGRAPH 2007 course notes.* In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, Seiten 1–81, New York, NY, USA, 2007. ACM.

[8] Bürger, M.: *Eine adaptive Hamilton–Jacobi Level-Set-Reinitialisierung für Zweiphasenströmungen.* Diplomarbeit, Institut für Numerische Simulation, Universität Bonn, Juli 2008.

[9] Cline, H. E. und W. E. Lorensen: *Marching cubes: A high resolution 3D surface construction algorithm.* SIGGRAPH Comput. Graph., 21(4):163–169, 1987.

[10] Cohen, D., A. Kaufman und R. Yagel: *Volume Graphics.* Computer, 26(7):51–64, 1993.

[11] Comet, M. B.: *Maya API Help Page.* `http://www.comet-cartoons.com/3ddocs/mayaAPI` (letzter Zugriff 17.02.2009).

[12] *Webseite zu COMSOL.* `http://www.comsol.com` (letzter Zugriff 19.03.2009).

[13] Croce, R.: *Ein paralleler, dreidimensionaler Navier-Stokes-Löser für inkompressible Zweiphasenströmungen mit Oberflächenspannung, Hindernissen und dynamischen Kontaktflächen.* Diplomarbeit, Institut für Angewandte Mathematik, Universität Bonn, Bonn, Deutschland, 2002.

[14] Croce, R., M. Engel und M. Griebel: *NaSt3DGPF - The Parallel 3D Free Surface Navier-Stokes Solver: User's Guide.*

[15] Croce, R., M. Engel, J. Strybny und C. Thorenz: *A Parallel 3D Free Surface Navier-Stokes Solver For High Performance Computing at the German Waterways Administration.* In: *The 7th Int. Conf. on Hydroscience and Engineering (ICHE-2006)*, Philadelphia, USA, September 2006.

[16] Croce, R., M. Griebel, M. Klitz, S.V. Lomov, G. Morren, D. Roose, H. Sol, B. Verleye und I. Verpoest: *Permeability of textile reinforcements: simulation, influence of shear, validation.* In: *Composites science and technology*, Band 68, Seiten 2804–2810. Elsevier Applied Science Publishers, Oktober 2008.

[17] Croce, R., M. Griebel und M. A. Schweitzer: *A Parallel Level-Set Approach for Two-Phase Flow Problems with Surface Tension in Three Space Dimensions.* Preprint 157, Sonderforschungsbereich 611, Universität Bonn, 2004.

[18] Croce, R., M. Griebel und M. A. Schweitzer: *Numerical Simulation of Droplet-Deformation by a Level Set Approach with Surface Tension.* Preprint 395, Sonderforschungsbereich 611, Universität Bonn, Bonn, Deutschland, 2008.

[19] Décoret, X. und E. Eisemann: *Fast scene voxelization and applications.* In: *I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, Seiten 71–78, New York, NY, USA, 2006. ACM.

[20] Décoret, X. und E. Eisemann: *Single-pass GPU solid voxelization for real-time applications.* In: *GI '08: Proceedings of graphics interface 2008*, Seiten 73–80, Toronto, Ont., Kanada, 2008. Canadian Information Processing Society.

[21] Derakhshani, D., J. Kundert-Gibbs, E. Kunzendorf und M. Larkins: *Mastering Maya 8.5.* Sybex, 2007.

[22] Dornseifer, T., M. Griebel und T. Neunhoeffer: *Numerical Simulation in Fluid Dynamics, a Practical Introduction.* SIAM, Philadelphia, 1998.

[23] Ewert, B.: *Howto–Seite zu MEL und der Maya-API.* `http://ewertb.soundlinker.com` (letzter Zugriff 17.02.2009).

[24] *Webseite zum freien Strömungslöser FEATFLOW.* `http://www.featflow.de` (letzter Zugriff 19.03.2009).

[25] Fedkiw, R. und N. Foster: *Practical animation of liquids.* In: *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, Seiten 23–30, New York, NY, USA, 2001. ACM.

[26] Fedkiw, R., M. Kang und X.-D. Liu: *A Boundary Condition Capturing Method for Multiphase Incompressible Flow.* Journal of Scientific Computing, 15(3):323 – 360, 2000.

[27] Fedkiw, R., N. Kwatra, F. Losasso und J. Talton: *Two-Way Coupled SPH and Particle Level Set Fluid Simulation.* IEEE Transactions on Visualization and Computer Graphics, 14(4):797–804, 2008.

[28] Fedkiw, R., F. Losasso, A. Selle und T. Shinar: *Multiple interacting liquids.* In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, Seiten 812–819, New York, NY, USA, 2006. ACM.

[29] Fefferman, Ch. L.: *Existence and Smoothness of the Navier-Stokes Equation - Official Problem Description.* In: *The Millenium Problems*, Cambridge, USA, 2000.

[30] *Webseite zum Programm Fluent.* `http://www.fluent.com` (letzter Zugriff 19.03.2009).

[31] Forcadel, N.: *Webseite zu: Autumn School: Introduction to numerical methods for moving boundaries, École des Ponts ParisTech, Paris (Frankreich).* `http://cermics.enpc.fr/~forcadel/methodenumerique/Accueil.htm` (letzter Zugriff 25.11.2008).

[32] Ganic, E.N. und S. Sikalo: *Phenomena of droplet-surface interactions.* Experimental Thermal and Fluid Science, 31(2):97 – 110, 2006. Second International Thermal Science Seminar (ITSS II).

[33] Gingold, R. A. und J. J. Monaghan: *Smoothed particle hydrodynamics - Theory and application to non-spherical stars.* Mon. Not. Roy. Astron. Soc., 181:375–389, November 1977.

[34] Glassner, A. S. (Herausgeber): *An introduction to ray tracing.* Academic Press Ltd., London, UK, UK, 1989.

[35] *Webseite der Firma 3DAliens, die glu3D anbietet.* `http://3daliens.com` (letzter Zugriff 19.03.2009).

[36] *Webseite der Google 3D-Galerie.* `http://sketchup.google.com/3dwarehouse` (letzter Zugriff 15.04.2009).

[37] Gould, D.: *Complete Maya Programming - An Extensive Guide to MEL and the C++ API.* Morgan Kaufmann, 1. Auflage, 2003.

[38] Gould, D.: *Complete Maya Programming Volume II - An In-Depth Guide to 3D Fundamentals, Geometry, and Modeling.* Morgan Kaufmann, 1. Auflage, 2005.

[39] Grossmann, Ch. und H.-G. Roos: *Numerik partieller Differentialgleichungen.* Teubner-Studienbücher: Mathematik. Teubner, Stuttgart, 2. edition Auflage, 1994.

[40] *Maya-Forum auf der Computer-Graphik–Seite Highend3d.* `http://www.highend3d.com/boards/index.php?showforum=291` (letzter Zugriff 17.02.2009).

[41] *Entwicklung des Cluster-Systems Himalaya vom Institut für Numerische Simulation der Universität Bonn (englisch).* `http://wissrech.ins.uni-bonn.de/research/himalaya/credits.html` (letzter Zugriff 02.03.2009).

[42] *Beschreibung der Cluster-Systeme des Instituts für Numerische Simulation der Universität Bonn (englisch).* `http://wissrech.ins.uni-bonn.de/research/himalaya/hardware.html` (letzter Zugriff 02.03.2009).

[43] Iglberger, K., U. Rüde und N. Thürey: *Free Surface Flows with Moving and Deforming Objects for LBM.* Proceedings of Vision, Modeling and Visualization 2006, Seiten 193–200, Nov 2006.

[44] Ju, T., F. Losasso, S. Schaefer und J. Warren: *Dual contouring of hermite data.* In: *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, Seiten 339–346, New York, NY, USA, 2002. ACM.

[45] Kameier, F., E. Kunz, O. Paschereit und H. Schade: *Strömungslehre.* Gruyter, 3. Auflage, 2007.

[46] Kaufman, A. und E. Shimony: *3D scan-conversion algorithms for voxel-based graphics.* In: *SI3D '86: Proceedings of the 1986 workshop on Interactive 3D graphics*, Seiten 45–75, New York, NY, USA, 1987. ACM.

[47] Kazmir, C. und M. R. Wilkins: *MEL Scripting for Maya Animators.* Morgan Kaufmann, 2. Auflage, 2005.

[48] Klitz, M.: *Homogenised Fluid Flow Equations in Porous Media with Application to Permeability Computations in Textiles.* Diplomarbeit, Institut für Numerische Simulation, Universität Bonn, Juli 2006.

[49] Lee, Y. T. und A. Requicha: *Algorithms for computing the volume and other integral properties of solids. II. A family of algorithms based on representation conversion and cellular approximation.* Commun. ACM, 25(9):642–650, 1982.

[50] Livny, B.: *mental ray for Maya, 3ds MAX and XSI.* Wiley Publishing, Inc., Indianapolis, 2008.

[51] Lucy, L. B.: *A numerical approach to the testing of the fission hypothesis.* Astronomical Journal, 82:1013–1024, Dezember 1977.

[52] Mahintorabi, K.: *Maya 8./8.5: 3D-Grafik und 3D-Animation.* Mitp-Verlag, Heidelberg, 1. Auflage, 2007.

[53] *Webseite des Herstellers von Autodesk Maya.* `http://www.autodesk.de` (letzter Zugriff 07.04.2009).

[54] Meshkov, E. E., V. V. Popov, V. G. Rogachev, A. I. Tolshmyakov und I. G. Zhidov: *Formation of a vortex ring during ascent of a large air bubble in water.* Journal of Applied Mechanics and Technical Physics, 18(3):341 – 343, 1977.

[55] Montani, C., R. Scateni und R. Scopigno: *A modified look-up table for implicit disambiguation of Marching Cubes.* The Visual Computer, 10(6):353–355, Dezember 1994.

[56] *Webseite des Message Passing Interface Forum mit Standardisierungsdokumenten zu MPI.* `http://www.mpi-forum.org` (letzter Zugriff 07.04.2009).

[57] Osher, S. und J. A. Sethian: *Fronts propagating with curvature dependent speed: algorithms based on Hamilton-Jacobi formulations.* Journal of Computational Physics, 79:12–49, 1988.

[58] *Webseite zum Visualisierungsprogramm Paraview.* `http://www.paraview.org` (letzter Zugriff 21.03.2009).

[59] *Webseite zum Programm Realflow.* `http://www.realflow.com` (letzter Zugriff 19.03.2009).

[60] Rein, M.: *Phenomena of liquid drop impact on solid and liquid surfaces.* Fluid Dynamics Research, 12(2):61 – 93, 1993.

[61] Rüde, U., M. Stamminger und N. Thürey: *Animation of open water phenomena with coupled shallow water and free surface simulations.* In: *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, Seiten 157–164, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association.

[62] Rüde, U. und N. Thürey: *Stable free surface flows with the lattice Boltzmann method on adaptively coarsened grids.* Computing and Visualization in Science, 2008.

[63] Schwarz, H. A.: *Gesammelte Mathematische Abhandlungen*, Band 2, Seiten 133–143. Springer Verlag, Berlin, Deutschland / Heidelberg, Deutschland / London, UK / etc., 1890.

[64] Sinclair, I. J. und I. R. Smith: *Deep water waves in lakes.* Freshwater Biology, 2(4):387–399, 1972.

[65] *Webseite zum Strömungspaket STAR-CCM+.* `http://www.cd-adapco.com/products/STAR-CCM_plus` (letzter Zugriff 19.03.2009).

[66] *Webseite zum Visualisierungswerkzeug Tecplot.* `http://www.tecplot.com` (letzter Zugriff 21.03.2009).

[67] Thuerey, N.: *Fluid Simulation with Blender.* Dr. Dobbs Journal, January 2006.

[68] *Liste der Top 500 schnellsten Rechnersysteme im Juni 2005.* `http://www.top500.org/list/2005/06/500` (letzter Zugriff 02.03.2009).

[69] Vogel, H.: *Gerthsen Physik.* Springer-Verlag, Berlin-Heidelberg, 20. Auflage, 1999.

[70] *Spezifikation zum Wavefront OBJ - Dateiformat.* `http://www.martinreddy.net/gfx/3d/OBJ.spec` (letzter Zugriff 12.12.2008).