Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Implementierung einer Erweiterung zur Behandlung von Hochgeschwindigkeitsaufprall mit dem SiPER SPH-Code

Tom Cesko

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. habil. Miriam Mehl
Betreuer/in:	Marvin Becker, M.Sc.
Beginn am:	4. Oktober 2018
Beendet am:	4. April 2019

Kurzfassung

Die Finite-Element-Methode ist gut geeignet, um die Flugbahn und Restgeschwindigkeit eines Projektils bei einem Hochgeschwindigkeitsaufprall vorherzusagen. Große Schwierigkeiten hat die Methode jedoch bei der Vorhersage des Bruchverhaltens eines Projektils. Im Gegensatz dazu ist die Smoothed Particle Hydrodynamics-Methode (SPH) ein verbreiteter Ansatz, um den Bruch von formbaren Körpern zu simulieren. Deshalb soll der Hochgeschwindigkeitsaufprall, als Alternative, mit einer SPH-Simulation durchgeführt werden. Um die zugrundeliegenden Gleichungen zu verstehen, wird eine quelloffene Umgebung gewählt. Das SiPER Simulationsprogramm bietet eine solche quelloffene SPH-Methode, muss aber um eine Rigid Wall und ein Feststoffmodell erweitert werden. Um die Entwicklung zu ermöglichen, wird ausführlich beschrieben, wie die SiPER Entwicklungsumgebung aufgesetzt wird und wie sie möglichst effizient verwendet werden kann. Zusätzlich wird detailliert beschrieben, wie SiPER konfiguriert wird und was die angebotenen Konfigurationen bedeuten. Zum Testen der Erweiterungen wird außerdem ein Testfall beschrieben. Die Erweiterungen selbst werden methodisch erklärt und daraufhin auch praktisch im Code umgesetzt. Dabei wird auf die Schwierigkeiten einer künstlichen Kraft eingegangen und wie diese bewältigt werden können. Außerdem werden ein Überblick über die Arbeit mit SiPER und Vorschläge zur Verbesserung der Codequalität gegeben.

Abstract

The finite element method is well suited to predict the trajectory and residual velocity of a projectile in a high velocity impact. However, the method has great difficulty in predicting the fracture of the projectile. In contrast, the Smoothed Particle Hydrodynamics (SPH) method is a widely used approach to simulate the fracture of ductile bodies. Therefore, the high velocity impact, as an alternative, will be performed with a SPH simulation. In order to understand the underlying equations, an open source environment is chosen. The SiPER simulation program offers such an open source SPH method, but must be extended by a rigid wall and a material model for solids. In order to enable the development, it is described in detail how the SiPER development environment is set up and how it can be used as efficiently as possible. In addition, it describes in detail how SiPER is configured and what the offered configurations mean. A test case is also described for testing the extensions. The extensions themselves are methodically explained and then practically implemented in the code. The difficulties of an artificial force and how these can be overcome are dealt with. In addition an overview of the work with SiPER and suggestions for the improvement of the code quality are given.

Inhaltsverzeichnis

1 Einleitung/Motivation **1**
- 1.1 Problemstellung . . . 1
- 1.2 SiPER . . . 1
- 1.3 Vorgehen . . . 2

2 Aufsetzen der SiPER Entwicklungsumgebung **3**
- 2.1 VirtualBox . . . 3
- 2.2 Windows Subsystem for Linux (WSL) . . . 5
- 2.3 Zusammenfassung . . . 8

3 SiPER Konfiguration **9**
- 3.1 ISPH Solver . . . 9
- 3.2 Erweiterungen/Defines . . . 11
- 3.3 Testfall . . . 15
- 3.4 ParticleGridMaker . . . 15
- 3.5 SiPER Aufbau . . . 17
- 3.6 Zusammenfassung . . . 18

4 Rigid Wall **19**
- 4.1 Methoden . . . 19
- 4.2 Implementierung . . . 24
- 4.3 Zusammenfassung . . . 26

5 Künstliche Viskosität **27**
- 5.1 Rigid Wall Partikel . . . 27
- 5.2 Künstliche Viskosität . . . 29
- 5.3 Implementierung . . . 30
- 5.4 Zusammenfassung . . . 31

6 Ergebnis **33**
- 6.1 Rigid Wall . . . 33
- 6.2 SiPER . . . 36

7 Zusammenfassung und Empfehlung **39**

Literaturverzeichnis **41**

Abbildungsverzeichnis

3.1 Tropfen des Testfalls 15
3.2 Screenshot des ParticleGridMaker 16
3.3 SiPER ISPH Flussdiagramm 17

4.1 Rigid Wall Methoden 23
4.2 Tropfen auf pure Boundary Force 26

6.1 Druckskala 33
6.2 Tropfen mit 1245 Partikel auf Rigid Wall mit $\alpha = 0$ 34
6.3 Tropfen mit 1245 Partikel auf Rigid Wall mit $\alpha = 1$ 34
6.4 Tropfen mit 1245 Partikel auf Rigid Wall mit $\alpha = 100$ 34
6.5 Tropfen mit 69 Partikel auf Rigid Wall mit $\alpha = 0$ 35
6.6 Tropfen mit 69 Partikel auf Rigid Wall mit $\alpha = 1$ 35
6.7 Tropfen mit 69 Partikel auf Rigid Wall mit $\alpha = 100$ 36

Tabellenverzeichnis

4.1 Vergleich der Rigid Wall Methoden 23

Verzeichnis der Listings

2.1 Installationsskript für gcc8 6
2.2 Installationsskript für PETSc 6
2.3 Installationsskript für weitere Pakete 7

3.1 Gridfile für 6 Partikel 9
3.2 Debugger Warteschleife 14

4.1 Rigid Wall Konfiguration 24
4.2 Implementierung Rigid Wall Kraft 25
4.3 Implementierung der Γ-Funktion 25
4.4 Implementierung der Wandabstandsberechnung 25

5.1 Implementierung Wand-Partikel-Generierung 28
5.2 Implementierung der künstlichen Viskosität 30
5.3 Implementierung künstlicher Viskosekraft 31

6.1 Beispiel Präprozessor 37
6.2 Beispiel allgemeine Vektorlösung 38

Abkürzungsverzeichnis

CFL Courant-Friedrichs-Lewy-Kriterium. 13

GUI Graphical User Interface. 7

ICVT Institut für Chemische Verfahrenstechnik. 1

ISPH Incompressible SPH. 9

PETSc Portable, Extensible Toolkit for Scientific Computation. 1

PPE Pressure Poisson Equation. 9

SPH Smoothed Particle Hydrodynamics. 1

WSL Windows Subsystem for Linux. 3, 5

Symbolverzeichnis

x Allgemeiner Punkt im Raum. 13

M Anzahl der Nachbarpartikel. 12

ρ Dichte. 9

p Druck, Durchschlag. 9

$\vec{g}$ Gravitation. 10

r_0 Initialer Partikelabstand. 19

σ Isotroper Spannungstensor, Oberflächenspannungskoeffizient. 14

$W(r,h)$ Kernelfunktion. 11

D Koeffizient der Lennard-Jones Repulsive Force. 19

C Konstante zwischen des Particle Shifting. 12

$\vec{f}_{CLF}$ Kraft der Kontaktliniendynamik. 10

κ Krümmung. 14

V Lokale Verschiebungsgeschwindigkeit. 20

m Masse. 10

U_{max} Maximale Partikelgeschwindigkeit. 12

$\vec{n}$ Normalenvektor. 14

$\vec{f}_{CSF}$ Oberflächenspannungskraft. 10

F Penalty Kraft. 20

$\vec{r}$ Position. 10

G Schubmodulo. 21

K Skalierungsfaktor der Penalty Kraft. 21

h Smoothinglänge. 10

α Verschiebungsdistanz, Viskositätsfaktor. 12

R Verschiebungsdistanz, Radii. 12

H Wassertiefe. 19

τ Zeitschritt. 9

1 Einleitung/Motivation

1.1 Problemstellung

Die Finite-Element-Methode ist gut geeignet, um die Flugbahn und Restgeschwindigkeit eines Projektils bei einem Hochgeschwindigkeitsaufprall vorherzusagen. Große Schwierigkeiten hat die Methode jedoch bei der Vorhersage des Bruchverhaltens eines Projektils [Bec18]. Im Gegensatz dazu ist die Smoothed Particle Hydrodynamics (SPH)-Methode ein verbreiteter Ansatz, um den Bruch von formbaren Körpern zu simulieren. Insbesondere große Geschwindigkeiten, Druckänderungen und Verformungen lassen sich mit dieser Methode gut verarbeiten. Um eine leichte Erweiterung um neue numerische Schemata und Material- bzw. Druckmodelle zu ermöglichen ist es notwendig, dass ein genaues Verständnis über die gelösten Gleichungen vorhanden ist. Um dies zu garantieren, ist ein freier Zugriff auf den Quellcode der Simulationsumgebung zwingend erforderlich. Das SPH-Simulationsprogramm SiPER des Instituts für Chemische Verfahrenstechnik (ICVT) bietet dahingehend eine solide Basis, bedarf allerdings einiger Erweiterungen, um die genannte Applikation des Hochgeschwindigkeitsaufpralls zu simulieren. Diese Arbeit beschäftigt sich mit genau diesen Erweiterungen.

1.2 SiPER

Das SiPER Simulationsprogramm ist ein expliziter 3D SPH-Solver, der die Simulation von Fluiden und Gasen in 1, 2 und 3 Dimensionen unterstützt [HKH+13]. Es ist in der Programmiersprache C geschrieben und darauf ausgelegt möglichst einfach verständlich und auch erweiterbar zu sein, ohne ausgeprägte Programmierkenntnisse zu besitzen. Dennoch kann SiPER mit einer großen Zahl von Partikeln umgehen und ist dabei sogar noch performant. Für die Parallelisierung wird auf das Message Passing Interface (MPI) gesetzt, das mit Hilfe des Portable, Extensible Toolkit for Scientific Computation (PETSc) implementiert ist. PETSc ist ein Framework, welches Datenstrukturen und Routinen für das Lösen von wissenschaftlichen Anwendungen, die durch partielle Differentialgleichungen modelliert sind, anbietet [BAA+17]. SiPER und der zugehörige Sourcecode ist auf Anfrage beim ICVT frei zugänglich, ermöglicht also Einblicke in den Solver und bietet die Möglichkeit der Erweiterung. Da SiPER nur Fluide und Gase simulieren kann, muss es um ein Materialmodell für Feststoffe und um eine Randbedingung (Rigid Wall) erweitert werden, damit es für den Hochgeschwindigkeitsaufprall angewendet werden kann.

1.3 Vorgehen

Als erster Schritt wird die SiPER Entwicklungsumgebung aufgesetzt und auf Lauffähigkeit geprüft. Dafür wird ein Testfall erstellt, der auch im späteren Verlauf zum Testen der Randbedingung verwendet wird. Danach werden geeignete Methoden für die Randbedingung in Form einer Rigid Wall verglichen und der am besten geeignete Ansatz implementiert. Die Implementierung wird mit dem erstellten Testfall geprüft, um dann für den Aufprall verwendet werden zu können. Im Anschluss soll ein geeignetes Materialmodell für Feststoffe eingeführt und ein Taylor Bar Impact Test zur Validierung durchgeführt werden.

2 Aufsetzen der SiPER Entwicklungsumgebung

Für die Anwendung und Entwicklung von SiPER ist es notwendig die SiPER Entwicklungsumgebung aufzusetzen, damit das Programm gebaut werden kann. SiPER wird in Form eines VirtualBox Images übergeben, das in die Oracle VirtualBox importiert werden kann. Durch den Import erscheint eine Virtual Machine, auf der OpenSUSE Leap 42.3 installiert ist, zusammen mit allen weiteren Komponenten, die für die Entwicklung von SiPER notwendig sind. Da die Entwicklung innerhalb einer Virtual Machine Einschränkungen haben kann und auch immer wieder Probleme mit dem Speicherplatz aufgetreten sind, wird im Folgenden erklärt, wie das Aufsetzen mit Hilfe der VirtualBox funktioniert, als Alternative aber auch der verwendete Ansatz einer manuellen Installation in das Windows Subsystem for Linux (WSL), das eine Entwicklung direkt in Windows 10 ermöglicht. Im späteren Verlauf der Arbeit wurde es zusätzlich möglich SiPER über ein GitLab Repository zu beziehen. Hierfür wird allerdings ein Login benötigt, das auf Anfrage beim ICVT erhältlich ist, zusammen mit dem Repository Link.

2.1 VirtualBox

Um das Image zu importieren, muss die aktuelle Version der Oracle VirtualBox installiert werden, die auf https://www.virtualbox.org/ heruntergeladen werden kann. Über die `Appliance Import` Funktion lässt sich die mitgelieferte .ova-Datei importieren und danach starten.

2.1.1 Speicherplatz erweitern

Da die virtuelle Festplatte des Images nur wenige MB an Speicherplatz übrig hat, können fast keine zusätzlichen Pakete oder Tools installiert werden. Dies ist für die reine Anwendung von SiPER nicht relevant, bei der Entwicklung oder auch Aktualisierung von PETSc reicht der Speicher jedoch nicht aus. Um diesen zu erweitern, muss die Virtual Machine ausgeschaltet sein, die Disk vom vmdk-Format (Virtual Machine Disk), dessen Größe nicht veränderbar ist, in das vdi-Format (VirtualBox Disk Image) kopiert und anschließend die Größe angepasst werden. Dies lässt sich in der VirtualBox im Werkzeug `Medien` erreichen. Anschließend muss das verwende Medium der SiPER Virtual Machine auf das neue Medium umgehängt werden. Dies lässt sich über die Option `Ändern` der Virtual Machine erreichen. Danach kann die Virtual Machine wieder gestartet werden. Um auch dem Betriebssystem mitzuteilen, dass die Festplatte größer geworden ist, muss im Terminal noch die Partition vergrößert werden. Dies geht bei laufender Root Partition unter OpenSUSE nur mit `fdisk`. Ist `fdisk` gestartet, müssen die folgenden Befehle ausgeführt werden:

1. `d`, um die betroffene Partition zu löschen.
2. `n`, um die Partition neu zu erstellen. Beim Setzten der Speicherblöcke sind die vorgeschlagenen Standardwerte korrekt, wenn der ganze verfügbare Speicherplatz der virtuellen Festplatte verwendet werden soll.
3. `t`, um den Typ der Partition zu setzen. Dieser muss auf $ID = 83$, also Linux, gesetzt werden.
4. `a`, um die Partition weiterhin als bootfähige Partition zu benutzen.
5. `p`, um die Änderungen zu überprüfen.
6. `w`, um die Änderungen zu übernehmen. Ohne diesen Schritt gehen die Änderungen nach dem Schließen von `fdisk` verloren.
7. Abschließend ist ein Neustart der Virtual Machine notwendig.

Jetzt ist der Speicherplatz freigegeben und kann verwendet werden.

2.1.2 Netzwerkkonfiguration

Ein weiter auftretendes Problem ist die Netzwerkkonfiguration, die dafür sorgt, dass PETSc oder genauer gesagt MPICH innerhalb der Virtual Machine nicht kommunizieren kann. Dies macht sich bemerkbar, sobald SiPER mit mehr als einem Prozess ausgeführt wird und mit einer Fehlermeldung abbricht. Eine Lösung dafür ist es den HOSTNAME zu ändern. Dafür muss mit `su root` auf den Root-User gewechselt und `/etc/HOSTNAME` editiert werden. Der dort vorgeschlagene Name kann auf etwas beliebiges anderes geändert werden. Anschließend muss `/etc/hosts` bearbeitet werden. Auch hier muss `localhost` durch den gewählten Namen ersetzt werden [Rhe15]. Anschließend ist es möglich SiPER in der Virtual Machine auch mit mehreren Prozessen auszuführen.

2.1.3 Compiler Aktualisierung

Zusätzlich war es in diesem sehr speziellen Fall nötig eine aktuellere Compiler-Version zu installieren, da die verwendete Ryzen R7 1800X CPU von der installierten `gcc`-Version noch nicht unterstützt wurde und somit weder PETSc noch SiPER gebaut werden konnte. Dazu muss `gcc8` zusammen mit dem `C++` und dem `Fortran` Compiler installiert werden. Damit die neu installierte Version auch immer verwendet wird, sollten außerdem Alternativen eingeführt werden, siehe Listing 2.1. Standardmäßig wird nun die aktuellere Version 8 verwendet, wenn zum Beispiel `gcc` aufgerufen wird. Überprüft werden kann dies mit `gcc --version`. Da der `gcc8` jetzt als Standard gesetzt ist, können die folgenden Schritte und Skripte regulär ausgeführt werden, ohne auf den verwendeten Compiler zu achten.

2.1.4 PETSc Aktualisierung

Alle benötigten Anwendungen und auch PETSc sind in der Virtual Machine vorhanden. Es ist allerdings zu empfehlen für die Entwicklung die aktuelle PETSc Version zu verwenden, diese steht über bitbucket zum Klonen per `git` bereit. Danach muss `configure` und `make` ausgeführt werden. Abschließend ist es möglich die Installation zu überprüfen. Das Skript aus Listing 2.2 installiert PETSc an der aktuellen Position in einem neuen Verzeichnis `petsc`. Damit sind PETSc und das mitgelieferte MPICH fertig installiert und können verwendet werden. Zur Verwendung des Compiler Wrappers exportiert das Skript noch die Pfade und erstellt die Umgebungsvariablen `PETSC_DIR` und `PETSC_ARCH`. Die Variablen werden vom SiPER Build-Skript verwendet.

2.2 Windows Subsystem for Linux (WSL)

Um SiPER bequem auf einer Windows 10 Plattform zu entwickeln, müssen alle benötigten Anwendungen und Pakete manuell installiert werden. Voraussetzung für problemloses Arbeiten mit SiPER ist das 1809 Update von Windows 10, sowie ein 64-bit System. Überprüft werden kann dies in den Einstellungen unter `System` im Tab `Info`. Direkt auf Windows zu installieren sind eine aktuelle Eclipse CDT (https://www.eclipse.org/cdt/) für die eigentliche Entwicklung und ein aktueller ParaView Release (https://www.paraview.org/) für die visuelle Darstellung der Ausgabe. Natürlich können auch beliebige andere Entwicklungsumgebungen und Visualisierungstools für VTK-Dateien verwendet werden, an dieser Stelle dient aber das VirtualBox Image als Vorbild. Für die Eclipse CDT ist es außerdem empfehlenswert die Eclipse XML Editors and Tools zu installieren, die im Menü `Help` unter dem Punkt `Install New Software…` zu finden sind.

2.2.1 WSL Installation

Der Kern dieses Ansatz ist das Windows Subsystem for Linux (WSL). Hierbei handelt es sich nicht um eine Virtual Machine, sondern um eine Kernel Schnittstelle, die Linux-ELF64-Binaries ausführen kann [CWRS16]. Der Vorteil daran ist vor allem die enge Verbindung zum eigentlich Windows System. Wichtig ist jedoch, dass die Kodierung und Zeilenumbrüche der in WSL verwendeten Dateien im UNIX Stil sind. Das WSL Feature muss über die Anwendung `Windows-Features aktivieren oder deaktivieren` installiert werden. Hier muss einfach der Haken bei `Windows-Subsystem für Linux` gesetzt und eventuell ein Neustart ausgeführt werden. Danach kann im Windows Store eine Distribution heruntergeladen werden. Wie in der VirtualBox wird OpenSUSE Leap 42 verwendet (https://aka.ms/wslstore). Anschließen kann die Distribution direkt als Programm gestartet und damit installiert werden. Nach erfolgreicher Installation und einem Systemneustart kann sie auch über die Windowsbefehlszeile mit den Befehlen `wsl` oder `bash` gestartet werden. Danach steht das gewohnte Linux Terminal zur Verfügung.

2.2.2 Compiler Aktualisierung

Auch hier ist die Compiler-Version zu alt für die in der Arbeit verwendete Ryzen R7 1800X CPU und wurde daher aktualisiert, um PETSc und SiPER bauen zu können. Dafür wurde `gcc8` zusammen mit dem zugehörigen C++ und Fortran Compiler installiert und Alternativen angelegt, um bequem

zwischen den Versionen wechseln zu können, siehe Listing 2.1. Standardmäßig wird dadurch der `gcc8` verwendet, überprüft werden kann das nach der Installation mit `gcc --version`. Die folgenden Schritte können dann regulär ausgeführt werden, ohne auf den verwendeten Compiler zu achten.

```
#!/bin/bash
sudo zypper install gcc8 gcc8-c++ gcc8-fortran
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 30 \
                         --slave /usr/bin/g++ g++ /usr/bin/g++-4.8 \
                         --slave /usr/bin/gfortran gfortran /usr/bin/gfortran-4.8
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 50 \
                         --slave /usr/bin/g++ g++ /usr/bin/g++-8 \
                         --slave /usr/bin/gfortran gfortran /usr/bin/gfortran-8
```

Listing 2.1: Installiert den gcc8 Compiler und erstellt Alternativen dafür.

2.2.3 PETSc Installation

Mit dem richtigen Compiler kann PETSc installiert werden. Dazu wird das Repository per `git` geklont, `configure` ausgeführt und `make` ausgeführt. Danach kann die Installation noch überprüft werden. Die dazu benötigten Befehle finden sich in Listing 2.2. Das Skript installiert PETSc an der aktuellen Position in ein neues Verzeichnis `petsc`. Damit PETSc und das mitgelieferte MPICH benutzbar sind bzw. der mpicc Compiler Wrapper aufrufbar ist, werden die Pfade noch exportiert und die Umgebungsvariablen `PETSC_DIR` und `PETSC_ARCH` erstellt. Die Variablen werden im Build-Skript von SiPER verwendet. Nach einem Neustart der `bash`, gibt auch `mpicc --version` eine Ausgabe, die zu der von `gcc --version` ausgegebenen Version passen sollte.

```
#!/bin/bash
git clone -b maint https://bitbucket.org/petsc/petsc petsc
cd petsc
sudo ./configure --download-fblaslapack --with-shared --download-hypre --download-mpich
CURRENT_DIR=$(pwd)
ARCH=arch-linux2-c-debug
make PETSC_DIR=$CURRENT_DIR PETSC_ARCH=$ARCH all
make PETSC_DIR=$CURRENT_DIR PETSC_ARCH=$ARCH check

echo >> ~/.bashrc
echo PATH=\$PATH:$CURRENT_DIR/$ARCH/bin:$CURRENT_DIR/$ARCH/include >> ~/.bashrc
echo LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:$CURRENT_DIR/$ARCH/lib >> ~/.bashrc
echo >> ~/.bashrc
echo export PETSC_DIR=$CURRENT_DIR >> ~/.bashrc
echo export PETSC_ARCH=$ARCH >> ~/.bashrc
```

Listing 2.2: Klont und installiert die aktuelle PETSc Version und exportiert die Pfade.

2.2.4 SiPER Abhängigkeiten

Des Weiteren benötigt SiPER noch einige weitere Pakete, um voll funktional zu sein. Diese werden durch das Skript aus Listing 2.3 installiert. Einige der Pakete sind zwingend nötig, da sie im Simulationscode verwendet werden. Andere sind für das Graphical User Interface (GUI) notwendig, das in dieser Arbeit aber nicht verwendet oder betrachtet wird. Durch das Installieren aller Pakete durch das Skript, sind alle Abhängigkeiten erfüllt und SIPER kann gebaut werden.

```
1 #!/bin/bash
2 sudo zypper in doxygen libxml2-devel gsl gsl-devel fftw3-devel PackageKit-Qt5-devel \
3                        libQt5Gui-devel libQt5Designer5 libQt5Core-devel libQt5Widgets-devel \
4                        libQt5OpenGL-devel libQt5Xml-devel libqt5-creator libqt5-qtbase-devel \
5                        libqt5-qtxmlpatterns-devel gdb
```

Listing 2.3: Installiert alle notwendigen Pakete für SiPER.

2.2.5 Eclipse CDT Konfigurieren

Um das Debugging über die Windows Eclipse CDT möglich zu machen, ist es notwendig passwortfreien Root Zugriff zu ermöglichen. Dazu muss mit dem Befehl `/usr/sbin/visudo`, die sogenannte `sudoer-Datei` bearbeitet und das Kommentarzeichen in Zeile `%wheel ALL=(ALL) NOPASSWD: ALL` entfernt werden. Danach ist der Standard Benutzer der wheel-Gruppe hinzuzufügen. Mit `sudo /usr/sbin/usermod -aG wheel $USER` wird der aktuelle Benutzer dieser Gruppe hinzugefügt. In einem späteren Schritt ist es dann möglich die Windows Eclipse CDT mit dem `gdb` Debugger an den SiPER Prozess anzuhängen.

Da nun alle Vorbereitungen abgeschlossen sind, kann man die Eclipse CDT regulär über Windows starten. Es wird davon ausgegangen, dass sich das SiPER Repository bereits auf dem System befindet und der Codestand mindestens dem dieser Arbeit entspricht. In der Eclipse `git` Perspektive kann das Repository über die Funktion `Add an exisiting local Git Repository to this view` hinzugefügt werden. Über das Kontextmenü des Repositorys ist es dann möglich die Funktion `Import Projects…` auszuführen, um die Projekte des Repositorys zu importieren. In der C/C++ Perspektive sollten nun die beiden Projekte `ICVTSPH` und `src` zu sehen sein. Im Projekt `src` sind zwei Build Targets definiert. Es sollte nun problemlos möglich sein per Doppelklick einen Clean Build auszuführen. Durch den Clean Build werden noch einige systemspezifische Links angelegt, damit auch Eclipse die Pfade korrekt liest, was nur möglich ist, wenn PETSc bereits gebaut ist. Es sei angemerkt, dass der Build im WSL ausgeführt wird. Eventuell muss der Indexer noch einen Rebuild ausführen, damit Eclipse die Includes korrekt erkennt.

Die Launch Konfiguration `Debug Siper` ist im Projekt enthalten und ermöglicht den Start des Debuggers, der sich dann an den ebenfalls startenden Prozess `Siper` anhängen lässt. Die Auswahl muss allerdings manuell getroffen werden. Wichtig ist, dass das DEBUG Define in der Input-Datei gesetzt ist, da SiPER sonst nicht auf den Debugger wartet. Umgekehrt ist es wichtig DEBUG nicht zu setzen, falls man nicht Debuggen möchten, da SiPER sonst auf den Debugger wartet und somit hängt. Genaueres zur Input-Datei folgt in Kapitel 3.

2.3 Zusammenfassung

SiPER wurde in der Virtual Box aufgesetzt und die nötigen Anpassungen vorgenommen, um die entstandene Speicherknappheit und die aufkommenden Netzwerkprobleme zu beheben. Alternativ wurde SiPER auf einer Windows 10 Plattform aufgesetzt. Dabei ist das WSL die Schnittstelle, um mit SiPER weiterhin unter Linux zu arbeiten. In beiden Fällen wurden der Compiler und PETSc aktualisiert. Auf der Windows 10 Plattform wurde außerdem noch die Eclipse CDT für eine komfortable Entwicklung eingerichtet, sodass diese direkt mit dem WSL arbeiten kann. SiPER kann jetzt gebaut, benutzt und erweitert werden. Die SiPER Konfiguration, die diese Arbeit benutzt wird in Kapitel 3 beschrieben.

3 SiPER Konfiguration

Da SiPER eine umfangreiche Konfiguration anbietet, wird im Folgenden detailliert erklärt wie SIPER zu konfigurieren ist und welche Konfiguration für den Testfall dieser Arbeit gewählt wurde. Zunächst zum Aufbau der Konfiguration: SiPER bekommt lediglich eine Input-Datei, in Form einer XML-Datei, übergeben. Diese XML-Datei beinhaltet alles, was SiPER an Konfigurationen anbietet. Angefangen bei den Präprozessor Defines, die in Abschnitt 3.2 behandelt werden, bis hin zum Ausgabedateiname. In dieser XML-Datei ist außerdem eine Textdatei angegeben, die als Gridfile dient, also die Startpositionen der Simulationspartikel enthält. Für eine geringe Partikelanzahl ist beispielhaft ein Gridfile in Listing 3.1 zu sehen. Dabei ist jede Zeile ein Partikel mit einer ID, die aufsteigend sein muss, einem Typ oder auch einer Color, und den Koordinaten X, Y und Z.

```
1  ###N_PARTICLES [X Y Z]###
2  100 100 1
3
4  ID  TYPE X    Y    Z
5  0   0    46.0 18.0 0.0
6  1   0    46.0 19.0 0.0
7  2   0    46.0 20.0 0.0
8  3   0    46.0 21.0 0.0
9  4   0    46.0 22.0 0.0
10 5   0    47.0 17.0 0.0
```

Listing 3.1: Gridfile für 6 Partikel mit der Größe des Rechenraums in Zeile 4 und Partikel ID, Typ und Koordinaten für alle drei möglichen Dimensionen.

3.1 ISPH Solver

Als Solver wird für diese Arbeit der bereits in SiPER implementierte Incompressible SPH (ISPH) Solver verwendet [HKH+16]. Dabei basiert das ISPH-Modell auf der Projektionsmethode, die erstmals von Cummins und Rudman vorgeschlagen wurde [CR99]. Bei dieser Methode berechnet sich der Druck durch das Lösen einer Pressure Poisson Equation (PPE) mit Hilfe des Kriteriums der Inkompressibilität

$$\nabla \cdot \left(\frac{1}{\rho} \nabla p \right) = \frac{\nabla \cdot \vec{u}^*}{\tau}, \tag{3.1}$$

nach dem die Divergenz der Geschwindigkeit 0 ist. Dabei sind τ, ρ, p und $\vec{u}^*$ der Zeitschritt, die Dichte, der Druck und die prädikative Geschwindigkeit.

Gleichung (3.1) lässt sich diskretisieren als

$$\nabla \cdot \left(\frac{1}{\rho}\nabla p\right)_a = \sum_b \frac{m_b}{\rho_b} \frac{4}{\rho_a + \rho_b} \frac{p_{ab}\vec{r}_{ab}}{|\vec{r}_{ab}|^2} \cdot \nabla_a W\left(r_{ab}, h\right) \tag{3.2}$$

und

$$\left(\nabla \cdot \vec{u}^*\right)_a = \sum_b \frac{m_b}{\rho_b} \left(\vec{u}_b^* - \vec{u}_a^*\right) \cdot \nabla_a W\left(r_{ab}, h\right), \tag{3.3}$$

wobei $p_{ab} = p_a - p_b$, $r_{ab} = r_a - r_b$, $\vec{r}$ die Position, m die Masse, h die Smoothinglänge und $W\left(r_{ab}, h\right)$ die Kernelfunktion aus Gleichung (3.7) bzw. Gleichung (3.8) ist. Im Predictor-Schritt wird die prädiktive Geschwindigkeit $\vec{u}^*$, basierend auf der Impulsbilanz ohne Druckkraft, berechnet

$$\vec{u}^* = \vec{u}^t + \left(\nu\Delta\vec{u} + \vec{g} + \vec{f}_{CSF} + \vec{f}_{CLF}\right) \cdot \tau. \tag{3.4}$$

Dabei sind $\vec{g}$, $\vec{f}_{CSF}$ und $\vec{f}_{CLF}$ Gravitation, Oberflächenspannungskraft (siehe auch Gleichung (3.23) und die Kraft der Kontaktliniendynamik. Für weiteres Details siehe [HKH+16]. Die Partikelposition wird im Predictor-Schritt nicht aktualisiert, sie bleibt gleich. Um die Inkompressibilität sicherzustellen wird im Corrector-Schritt der Druck berechnet, indem ein lineares Gleichungssystem iterativ gelöst wird. Daraus ergibt sich die resultierende Geschwindigkeit im folgenden Zeitschritt durch

$$\vec{u}^{t+1} = \vec{u}^* - \left(\frac{1}{\rho}\nabla p\right) \cdot \tau, \tag{3.5}$$

aus der sich die Position im folgendem Zeitschritt ergibt aus

$$\vec{r}^{t+1} = \vec{r}^t + \frac{\vec{u}^{t+1} + \vec{u}^t}{2}. \tag{3.6}$$

3.2 Erweiterungen/Defines

SiPER bietet verschiedene zusätzliche Features, die über Defines aktiviert werden können. Im Folgenden werden die in dieser Arbeit verwendeten Defines aufgezählt und deren Funktionalität erläutert.

3.2.1 DIM2

`DIM2` gibt die Dimensionen der Simulation an. Zur Auswahl stehen `DIM1`, `DIM2` und `DIM3`. In dieser Arbeit beschränken wir uns jedoch auf zwei Dimensionen, also `DIM2`.

3.2.2 DEFAULT

Das `DEFAULT` Define wird lediglich für interne und anwenderspezifische Änderungen verwendet. Es kann prinzipiell entfernt werden, bleibt jedoch als Platzhalter stehen.

3.2.3 KERNEL_WENDLAND_2D

`KERNEL_WENDLAND_2D` gibt den zu verwendenden Kernel $W(r,h)$ an. Angeboten werden `KERNEL_QUADRIC_1D`, `KERNEL_QUADRIC_2D`, `KERNEL_QUINTRIC_1D`, `KERNEL_QUINTRIC_2D`, `KERNEL_QUINTRIC_3D`, `KERNEL3`, `KERNEL_WENDLAND_1D`, `KERNEL_WENDLAND_2D` und `KERNEL_WENDLAND_3D`. Für die 2D Simulation dieser Arbeit wird `KERNEL_WENDLAND_2D` verwendet, ein zweidimensionaler Quintic-Wendland-Kernel nach [Wen96]. Die genaue Implementierung nutzt

$$W(r,h) = \begin{cases} \frac{7}{4\pi h^2}\left(1-\frac{q}{2}\right)^4(1+2q) & 0 \le q \le 2 \\ 0 & 2 < q \end{cases} \tag{3.7}$$

mit $q = \frac{r}{h}$. Die dreidimensionale Implementierung `WENDLAND_3D` verwendet

$$W(r,h) = \begin{cases} \frac{21}{16\pi h^3}\left(1-\frac{q}{2}\right)^4(1+2q) & 0 \le q \le 2 \\ 0 & 2 < q \end{cases} . \tag{3.8}$$

3.2.4 PARTICLE_SHIFTING_XU

PARTICLE_SHIFTING_XU aktiviert das Partikel-Shifting nach Xu [XSL09]. Diese Methode verhindert, dass nicht uniforme Partikelverteilungen entstehen und somit das Divergieren des linearen Solvers. Dies wird erreicht, indem die Partikelposition leicht korrigiert wird und die hydrodynamischen Variablen an der neuen Position durch die Taylorreihe dementsprechend interpoliert werden

$$\phi_{a'} = \phi_a + \delta r_{aa'} \cdot (\nabla \delta)_a + O(\delta r_{aa'}^2), \tag{3.9}$$

dabei ist ϕ eine generelle Variable, a und a' sind die alte und neue Partikelposition und $\delta r_{aa'}$ ist der Distanzvektor zwischen alter und neuer Partikelposition. Die Partikelverschiebung wird beschrieben durch

$$\delta r_a = C \alpha R_a, \tag{3.10}$$

mit C, einer Konstante zwischen $0.01 - 0.1$. α ist die Verschiebungsdistanz, die der maximalen Partikelbewegung $\alpha = U_{max}\tau$ gleicht, wobei U_{max} die maximale Partikelgeschwindigkeit ist. R_a ist der Verschiebungsvektor

$$R_a = \sum_b \frac{\overline{r}_a^2}{r_{ab}^2} \vec{n}_{ab}, \tag{3.11}$$

dabei ist r_{ab} die Distanz zwischen Partikel a und b und $\vec{n}_{ab}$ ist der Einheitsabstandsvektor zwischen a und b. $\overline{r}_a$ ist der durchschnittliche Partikelabstand in der Nachbarschaft von a, der gegeben ist durch

$$\overline{r}_a = \frac{1}{M_a} \sum_b r_{ab}, \tag{3.12}$$

wobei M_a die Anzahl der Nachbarpartikel von Partikel a ist.

3.2.5 SHIFT_UMAX_GLOBAL

SHIFT_UMAX_GLOBAL erweitert das Partikel-Shifting aus Abschnitt 3.2.4, sodass als maximale Geschwindigkeit U_{max} nicht mehr die maximale Geschwindigkeit der Nachbarpartikel betrachtet wird, sondern die maximale Geschwindigkeit aller Partikel im gesamten Rechengebiet.

3.2.6 ADAPTIVE_TIMESTEPPING

ADAPTIVE_TIMESTEPPING aktiviert das adaptive Zeitschrittprinzip mit folgenden Kriterien, um eine stabile Simulation zu gewährleisten [HKH+16]. Der Zeitschritt für die Integration ist begrenzt durch das Courant-Friedrichs-Lewy-Kriterium (CFL)

$$\tau_{CFL} = \frac{\alpha_{CFL} \cdot r_0}{|\vec{u}_{max}|}, \tag{3.13}$$

mit $\alpha_{CFL} = 0.05$ und einem Kriterium für viskose Diffusion

$$\tau_{visc} = \frac{\alpha_{visc} \cdot r_0^2}{\nu_{max}}, \tag{3.14}$$

mit $\alpha_{visc} = 0.0625$. Daraus berechnet sich die maximale Schrittweite als

$$\tau_{max} = min \left\langle \tau_{CFL}, \tau_{visc} \right\rangle . \tag{3.15}$$

Ist τ_{max} kleiner als der aktuell benutzte Zeitschritt, wird dieser entsprechend verkleinert. Ist τ_{max} groß genug, so wird auch der benutzte Zeitschritt vergrößert.

3.2.7 CORRECTED_SPH

CORRECTED_SPH aktiviert die Verwendung der korrigierten Kernelfunktionen nach Bonet und Lok [BL99]. Dafür wird die Geschwindigkeitsapproximation beschrieben als

$$\vec{u}\,(x) = \sum_b V_b \vec{u}_b \widetilde{W}_b\,(x), \quad \text{mit } \widetilde{W}_b = \frac{W_b\,(x)}{\sum_b V_b W_b\,(x)}. \tag{3.16}$$

Dabei ist x ein allgemeiner Punkt im Raum. Der Geschwindigkeitsgradient ergibt sich aus

$$\nabla \vec{u}_a = \sum_b \frac{m_b}{\rho_b} \vec{u}_b \otimes \widetilde{\nabla} \widetilde{W}_b\,(x_a), \tag{3.17}$$

dabei ist der korrigierte Kernelgradient

$$\widetilde{\nabla} \widetilde{W}_b\,(x_a) = L_a \nabla \widetilde{W}_b\,(x_a)\,. \tag{3.18}$$

Daraus ergibt sich die Korrekturmatrix

$$L_a = \left(\sum_b \frac{m_b}{\rho_b} \nabla \widetilde{W}_b\,(x_a) \otimes x_b \right)^{-1}. \tag{3.19}$$

Letztendlich können die inneren Ersatzkräfte, nach dem Variationsprozess aus [BL99] mit Gleichung (3.17), eines Partikels geschrieben werden als

$$T_a = \frac{m_a}{\rho_a} \sum_b \frac{m_b}{\rho_b} \sigma_b \widetilde{\nabla} \widetilde{W}_a (x_b). \tag{3.20}$$

Dabei ist σ_b der isotrope Spannungstensor.

3.2.8 FREE_SURFACE_FORCE

`FREE_SURFACE_FORCE` aktiviert die Oberflächenkräfte an freien Oberflächen nach [HONL17]. Dafür werden die Normalen $\vec{n}$ des Fluids durch den Gradienten der Color-Funktion berechnet

$$\left(\frac{\nabla C}{[C]} \right)_a = \vec{n}_a = - \sum_b \frac{m_b}{\rho_b} \nabla W_{ab}. \tag{3.21}$$

Daraus berechnet sich die Krümmung κ als Divergenz der Einheitsnormalen

$$- \left(\nabla \cdot \hat{\vec{n}} \right)_a = \kappa_a = \sum_b \frac{m_b}{\rho_b} \left(\hat{\vec{n}}_a - \hat{\vec{n}}_b \right) \widetilde{\nabla} \widetilde{W}_{ab} \tag{3.22}$$

unter Benutzung des korrigierten SPH Kernels aus Abschnitt 3.2.7. Dabei werden Partikel für deren Normale $|\vec{n}| < \frac{0.01}{h}$ gilt übersprungen. Außerdem werden Normale und Krümmung nach der Berechnung noch einmal geglättet. Letztendlich ist die Kraft der Oberflächenspannung

$$\vec{f}_{CSF,a} = \sigma_a \kappa_a \hat{\vec{n}}_a \, |\vec{n}_a| \tag{3.23}$$

mit σ_a als Oberflächenspannungskoeffizient.

3.2.9 DEBUG

Ist das `DEBUG` Define gesetzt, gibt SiPER zu Beginn der Ausführung die Prozess ID der laufenden SiPER Prozesse aus und wartet in einer Endlosschleife, bis sich ein Debugger einhängt und das `i` aus Listing 3.2 auf einen Wert $\neq 0$ setzt.

```
1   int i = 0;
2   char hostname[256];
3   gethostname(hostname, sizeof(hostname));
4   printf("PID %d on %s ready for attach\n", getpid(), hostname);
5   fflush(stdout);
6   while (0 == i)
7     sleep(5);
```

Listing 3.2: Endlosschleife, die ein Warten auf den Debugger ermöglicht. Ist der Debugger eingehängt, kann das i über diesen geändert und die Ausführung fortgesetzt werden.

3.3 Testfall

Zum Testen der Implementierung einer Rigid Wall wurden zwei unterschiedlich große Tropfen verwendet, die in einer konstanten Geschwindigkeit von $1\frac{m}{s}$ gegen die Wand prallen. Dafür wurden zwei unterschiedlich große 2D Tropfen, mit dem eigens für die diese Arbeit entwickelten Programm `ParticleGridMaker` (siehe Abschnitt 3.4), generiert. Der eine Tropfen besteht aus nur 69 Partikeln (Abbildung 3.1a), der andere aus 1245 Partikeln (Abbildung 3.1b).

(a) 69 Partikel **(b)** 1245 Partikel

Abbildung 3.1: Zwei für den Testfall generierte Tropfen mit unterschiedlichen Partikelzahlen.

Weitere Einstellungen betreffen das σ der Oberflächenspannung aus Gleichung (3.23), welches 0 gesetzt wurde, sodass die Oberflächenspannungskräfte deaktiviert sind, ohne das `FREE_SURFACE_FORCE`-Feature zu deaktivieren. Dadurch wird erreicht, dass die Oberflächenspannungskräfte, deren aktuelle Implementierung nicht mit Randbedingungen umgehen kann, nicht aktiv ist, der Einfluss auf die Druckkräfte jedoch erhalten bleibt und somit am Rand der Tropfen kein Druckabfall entsteht. Das σ kann nur direkt im Code gesetzt werden, es gibt dafür keine Konfigurationsmöglichkeit. Es wird nur ein Prozess verwendet und vom Launch-Skript des Testfalls werden noch die Start-Parameter `-pc_type hypre -pc_hypre_type boomeramg -pc_hypre_boomeramg_max_iter 5` hinzugefügt. Diese Parameter wurden vom ursprünglichen SiPER Testfall `TropfenFreieOberflaeche` übernommen.

3.4 ParticleGridMaker

Sollen verschiedene Tropfen getestet werden, werde auch entsprechende Gridfiles benötigt. SiPER bot bisher keinerlei Möglichkeit diese Gridfiles für Tropfen bequem zu generieren. Stattdessen gab es eine Reihe von MATLAB-Skripten, die an die eigenen Bedürfnisse angepasst werden mussten. Deshalb wurde im Zuge der Arbeit das Programm `ParticleGridMaker` entwickelt. Das Programm bietet eine GUI, mit dem bequem Tropfen mit verschiedener Partikelzahl und Partikeldichte generiert und

auch als SiPER kompatible Gridfiles abgespeichert werden können. Es ist in der Programmiersprache Java in Version 8 geschrieben und die GUI verwendet das JavaFX Framework. Zur Ausführung der Jar-Datei wird dementsprechend eine Java 8 JRE benötigt, wie sie unter https://www.java.com/ zum Download bereit steht.

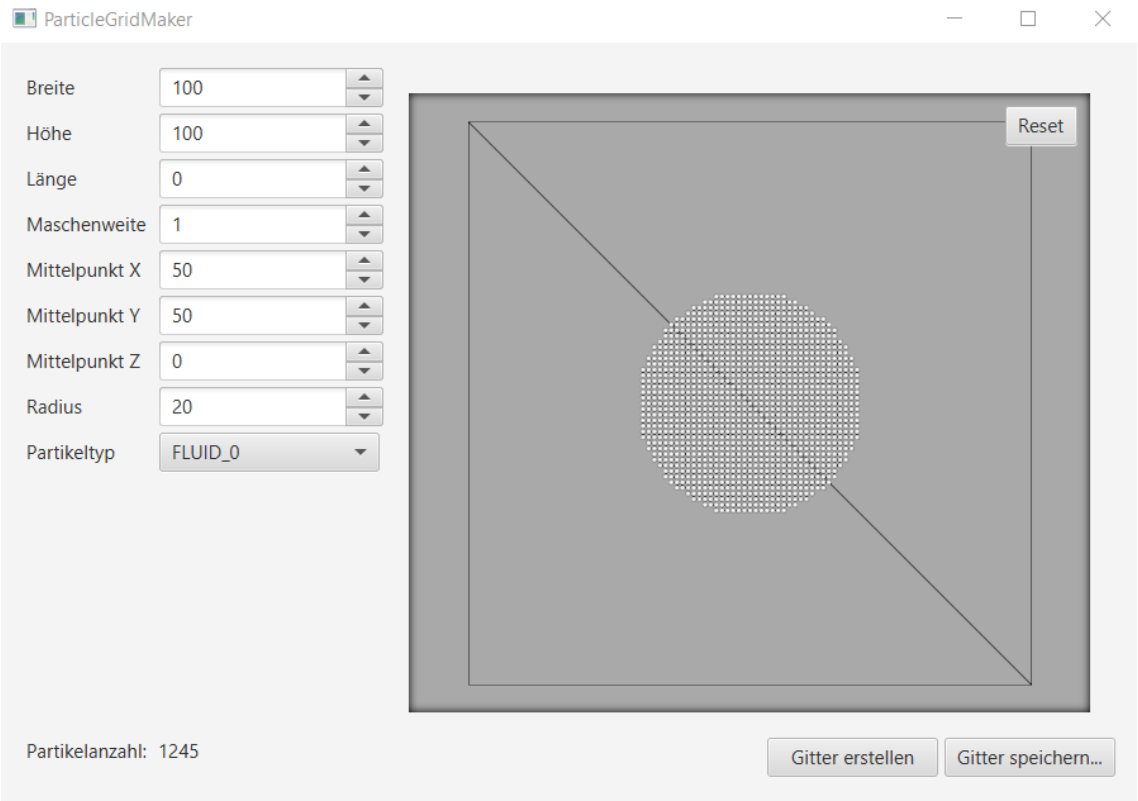


Abbildung 3.2: Screenshot des ParticleGridMaker. Auf der linken Seite können Breite, Höhe und Länge des Rechenraums , die Maschenweite des Partikelgitters, der Ort des Mittelpunkts des Tropfens als X, Y, und Z Koordinaten, der Radius des Tropfens und auch der Typ der Partikel angegeben werden. Auf der rechten Seite können die generierten Partikel in 3D betrachtet werden. Zusätzlich ist auch der Rechenraum dargestellt.

In der GUI lassen sich der Rechenraum und der Tropfen einfach konfigurieren. Über die Schaltfläche `Gitter erstellen` wird das Gitter generiert und dargestellt. Die Kamera des Vorschaufensters lässt sich mit der Maus schwenken und über die `Reset`-Schaltfläche auf die ursprüngliche Position zurücksetzten. So kann der generierte Tropfen überprüft werden, ohne eine Testsimulation durchführen zu müssen. Auch wird direkt die Partikelanzahl angezeigt. Über die Schaltfläche `Gitter speichern...` kann der Tropfen dann als SiPER Gridfile abgelegt und daraufhin verwendet werden.

Das Programm kann in seiner jetzigen Form nur runde Tropfen generieren. Eine Erweiterung zum Einlesen von 3D Meshes und der Generierung von Partikelgittern aus 3D Meshes ist problemlos möglich, wurde in dieser Arbeit jedoch nicht weiterverfolgt, da dies nicht nötig wurde.

3.5 SiPER Aufbau

SiPER bietet mehrere Solver an. Wie in Abschnitt 3.1 beschrieben wird die bereits implementierte ISPH Lösung verwendet. Der Ablauf dieser Lösung ist in Abbildung 3.3 zu sehen.

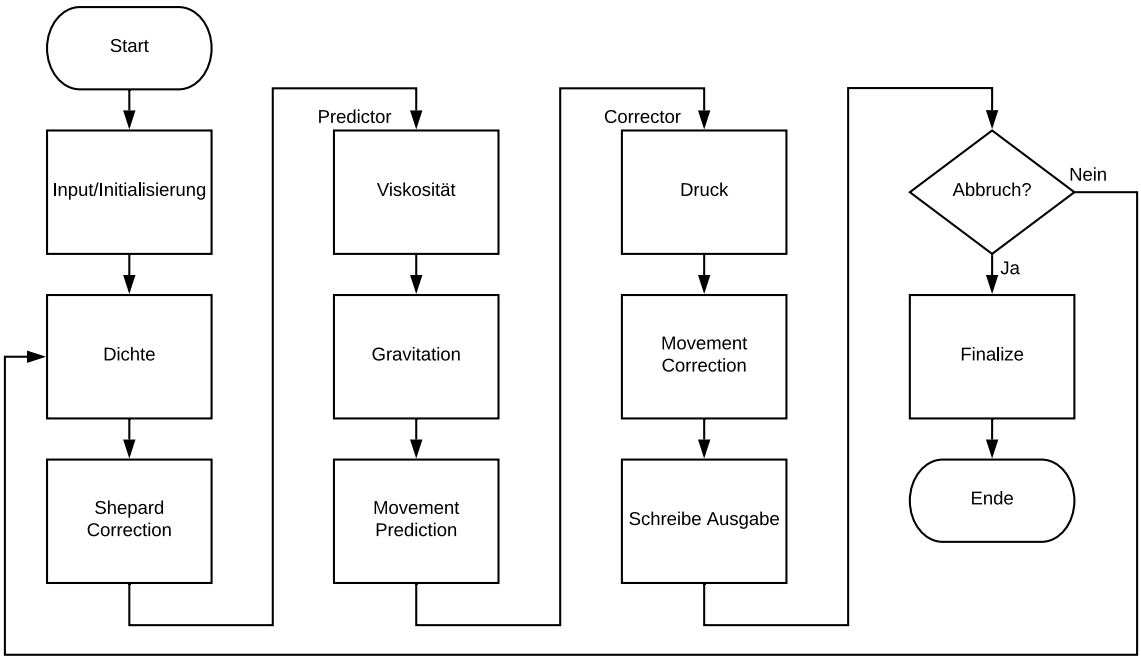


Abbildung 3.3: SiPER ISPH Flussdiagramm in vereinfachter Form, ohne Kommunikationsroutinen. Dabei ist die Aufteilung der Berechnungen der Attribute in Predictor- und Corrector-Schritt zu erkennen.

Falls SiPER mit mehr als einem Prozess verwendet wird, finden sich an den notwendigen Stellen noch Kommunikationsabläufe, diese wurden in Abbildung 3.3 allerdings weggelassen, da wir, wie in Abschnitt 3.3 beschrieben, nur einen Prozess verwenden und das Diagramm sehr unübersichtlich werden würde.

Die Daten der Partikel werden in einer großen Matrix gehalten, in der jede Spalte ein Partikel darstellt. Somit kann mit der ID eines Partikels, die gleich dem Spaltenindex ist, jedes andere Attribut abgefragt werden. Auf die einzelnen Attributlisten kann zusätzlich auch über eigene Pointer zugegriffen werden, die Matrix wird aber teilweise auch direkt verwendet. Die Aufteilung des Codes für 1D, 2D und 3D wird größtenteils mit Defines und somit mit `#ifdef` im Code gelöst, was dazu führt, dass viele Berechnungen drei Mal implementiert sind. Die Header-Dateien werden in einen Sammelheader gehängt, der in allen C-Dateien als Abhängigkeit eingebunden wird. Dadurch muss der Compiler immer vollständig neu bauen, wenn eine Header-Datei geändert wird, was dazu führt, dass kleine Änderungen teilweise viel Zeit beanspruchen können.

3.6 Zusammenfassung

Der verwendete ISPH Solver und die verwendeten SiPER Features wurden gezeigt und erklärt. Außerdem wurde ein Testfall mit zwei unterschiedlich großen Tropfen erstellt und die Konfiguration dieser aufgezeigt. Dabei wurde ein Programm entwickelt, das Tropfen-Partikel-Gitter in unterschiedlichen Größen generieren kann. Abschließend wurde der bisherige Codestand und der Ablauf des SiPER Codes aufgezeigt. Dabei wurde auch ein Einblick in die interne Funktionsweise von SiPER gegeben. Da SiPER jetzt konfiguriert ist und der Ablauf dargestellt wurde, wird in Kapitel 4 die Erweiterung des SiPER Codes betrachtet.

4 Rigid Wall

Um einen Hochgeschwindigkeitsaufprall zu modellieren soll zunächst eine nicht deformierbare Wand (Rigid Wall) verwendet werden. Im ersten Schritt wird daher eine geeignete Methode zur Modellierung einer Rigid Wall ausgewählt, umgesetzt und getestet. Die Wand soll einfach zu konfigurieren sein, daher soll diese in Form einer Ebenengleichung in das Konfigurations-XML eingetragen werden können.

4.1 Methoden

In der Literatur gibt es einige Vorschläge zur Umsetzung von Boundary Conditions, mit unterschiedlichen Vor- und Nachteilen. Daher werden die verschiedenen Ansätze im Folgenden kurz vorgestellt und anschließend verglichen, um verständlich zu machen auf welcher Basis eine Entscheidung getroffen wurde.

4.1.1 Lennard-Jones Repulsive Force

Ein sehr einfacher Ansatz, der erstmals von Monaghan [Mon94] vorgeschlagen wurde, ist die Repulsive Force nach dem Lennard-Jones Potential. Sie basiert auf den Kräften, die zwischen Molekülen wirken. Für eine Wand und ein Fluid-Partikel mit einem Abstand r ist die Kraft pro Masseneinheit

$$f(r) = \begin{cases} D\left(\left(\frac{r_0}{r}\right)^{p_1} - \left(\frac{r_0}{r}\right)^{p_2}\right)\frac{r}{r^2} & r \leq r_0 \\ 0 & sonst \end{cases} . \tag{4.1}$$

Dabei sind p_1 und p_2 Konstanten, r_0 ist der Initialabstand zwischen den Partikeln und D ein Koeffizient. Für die Konstanten p_1 und p_2 gilt $p_1 > p_2$ und in den meisten Fällen $p_1 = 4$ und $p_2 = 2$. Für den Koeffizient D gilt $D = 5gH$, wobei H die Wassertiefe ist. Auch $D = 10gH$ oder $D = gH$ ergeben nach [Mon94] ähnliche Resultate. Die Kraft wird dann auf die Fluid-Partikel angewendet, deren Abstand zur Wand $\leq r_0$ ist, siehe Abbildung 4.1a. Zusätzlich ist es nötig, die in Abschnitt 4.1.4 erklärten Boundary Partikel zu verwenden, um zu erreichen, dass kein Partikel durch die Wand hindurch fliegt. Die Boundary Partikel werden dann mit in die Berechnung der Viskosität einbezogen.

4.1.2 Ghost Particles

Ein weiterer, sehr verbreiteter Ansatz sind Ghost Partikel nach Colagrossi und Landrini [CL03], bei denen die Fluid-Partikel an der Wand gespiegelt werden, siehe Abbildung 4.1b. Diese Ghost Partikel imitieren Dichte, Druck und Geschwindigkeit der eigentlichen Fluid-Partikel. Dabei gilt für die Position eines Ghost Partikel, das das Spiegelbild eines Fluid-Partikels a ist,

$$x_{a_G} = 2x_w - x_a, \tag{4.2}$$

für die Normalengeschwindigkeit

$$v_{na_G} = 2V_{nw} - v_{na}, \tag{4.3}$$

für die Tangentialgeschwindigkeit

$$v_{ta_G} = v_{ta} \tag{4.4}$$

und für den Druck

$$p_{a_G} = p_a. \tag{4.5}$$

V ist die lokale Verschiebungsgeschwindigkeit der Boundary mit der aktuellen Position x_w. Im Fall der Rigid Wall, die sich nicht bewegt, ist $V = 0$, sodass für v_{na_G} gilt $v_{na_G} = -v_{na}$. Zu Beachten ist, dass diese Partikel in jedem Zeitschritt neu erzeugt werden müssen.

4.1.3 Penalty Based

Bei der Penalty-Basierten Methode nach Campell, Vignjevic und Libersky [CVL00] geht man davon aus, dass es einen gewissen Durchschlag gibt, wenn zwei Körper sich berühren. Die Wand wird hier durch statische Partikel modelliert, denen zusätzlich eine abstoßende Kraft gegeben wird, siehe Abbildung 4.1c. Wenn der Kontakt zwischen zwei Partikeln eintritt, kann ein Durchschlag erkannt werden durch

$$p = \frac{h_a + h_b}{2} - |r_{ab}| \geq 0. \tag{4.6}$$

Dabei ist $r_{ab} = r_b - r_a$ der Vektor vom Zentrum des Partikels a zum Zentrum des Partikels b, p der Durchschlag und h_a, h_b die Smoothinglängen von a bzw. b. Die Kontakt- oder Penaltykraft F wird entlang des Vektors r_{ij} angewendet. Berechnet werden kann diese wie bei Belytschko's Pinball Algorithmus [BY93] durch

$$F = K_p min\left(F_1, F_2\right) \tag{4.7}$$

mit

$$F_1 = \begin{cases} \dfrac{\rho_a \rho_b R_a^3 R_b^3}{\rho_a R_a^3 + \rho_b R_b^3} \cdot \dfrac{\dot{p}}{\Delta t} & \dot{p} > 0 \\ 0 & \dot{p} < 0 \end{cases} \tag{4.8}$$

und

$$F_2 = \left[\frac{G_a G_b}{G_a + G_b} \sqrt{\frac{R_a R_b}{R_a + R_b}} \right] p^{\frac{3}{2}}. \tag{4.9}$$

Dabei ist G der Schubmodulo, R der Radii und K_p ist ein Skalierungsfaktor. Auch andere Kontaktkräfte sind denkbar, siehe [CVL00].

4.1.4 Boundary Particles

Ein rein auf Partikeln basierender Ansatz sind die Boundary Partikel nach Violeau und Issa [VI07] und Lobovskỳ und Křen [LK07]. Dabei wird die Wand aus zwei Arten von Partikeln modelliert, siehe Abbildung 4.1d. Zum einen die Interface-Partikel, die die Oberfläche der Boundary markieren und zum anderen die Wall-Partikel, die den Körper der Boundary auffüllen. Dabei werden Druck und Dichte der Wall-Partikel regulär durch die geltenden Gleichungen aktualisiert, Druck und Dichte der Interface-Partikel jedoch nicht. Die Geschwindigkeit beider Partikel ist relativ zur Wand gesehen 0, sie bewegen sich nicht.

4.1.5 Boundary Force

Ein weiterer, neuerer Ansatz, der dem Penalty-Basierten sehr ähnelt, ist die Boundary Force nach Monaghan, Kos und Issa [MKI03], siehe Abbildung 4.1e. Demnach lässt sich die Kraft, die auf die Boundary-Partikel wirkt darstellen als:

$$f_b = \sum_a f_{ba} \tag{4.10}$$

wobei f_{ba} die Kraft ist, die jedes einzelne Fluid-Partikel a, das sich innerhalb der Smoothinglänge befindet, auf das Boundary Partikel b ausübt. Im Sinne der Boundary Force Methode ist die Kraft f_{ba} gegeben durch

$$f_{ba} = -\frac{m_a}{m_a + m_b} B(x,y) \vec{n}_b. \tag{4.11}$$

Dadurch ergibt sich nach Newtons drittem Gesetz, dass die Kraft f_{ab}, die jedes Boundary-Partikel b auf ein Fluid-Partikel a ausübt, gegeben ist durch

$$f_{ab} = \frac{m_b}{m_a + m_b} B(x,y) \vec{n}_b \tag{4.12}$$

mit $\vec{n}_b$ als Normalenvektor, der senkrecht zur Boundary an der Stelle des Boundary-Partikels b steht, x und y sind die tangentiale und normale Distanz zwischen Partikel a und b und $B(x,y)$ ist eine Funktion der lokalen Koordinaten x und y. Daraus ergibt sich, dass die Boundary Force f_a, die auf ein Fluid-Partikel a wirkt, berechnet werden kann durch die Summe

$$f_a = \sum_b f_{ab} \tag{4.13}$$

über alle Boundary-Partikel b, die innerhalb der Smoothinglänge von a liegen. Diese Kraft lässt sich dann zur SPH Approximation der Bewegungsgleichung hinzuaddieren. Für $B(x,y)$ gibt es nach Sutti [Sut14] mehrere Ansätze in der Literatur, wir beschränken uns jedoch auf den in [MKI03] vorgeschlagenen:

$$B\left(x,y\right) = \Gamma(y)\chi(x) \tag{4.14}$$

mit

$$\chi(x) = \begin{cases} \left(1 - \dfrac{x}{\delta p}\right) & 0 < x < \Delta p \\ 0 & sonst \end{cases} \tag{4.15}$$

und

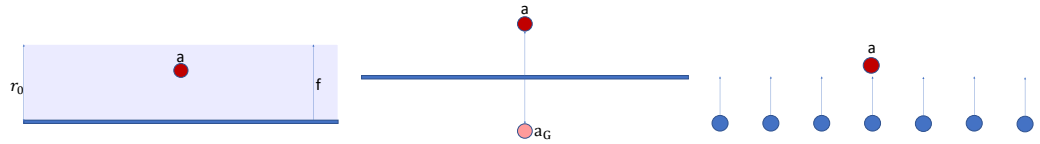
$$\Gamma\left(y\right) = \begin{cases} \dfrac{2}{3}\beta & 0 < q < \dfrac{2}{3} \\ \beta\left(2q - \dfrac{3}{2}q^2\right) & \dfrac{2}{3} < q < 1 \\ \dfrac{1}{2}\beta\left(2-q\right)^2 & 1 < q < 2 \\ 0 & sonst \end{cases} . \tag{4.16}$$

Dabei ist Δp der Abstand zwischen den Boundary-Partikeln, für den für gewöhnlich gilt $\Delta p = \frac{1}{2}r_0$ und $q = \frac{y}{h}$, mit $\beta = 0.02\frac{c_s^2}{y}$ und c_s ist die Schallgeschwindigkeit. Zur Vereinfachung der Methode, können die eigentlichen Partikel auch weggelassen werden, sodass die Methode einem repulsiven Ansatz gleicht. Dafür nehmen wir an, dass das Wand-Partikel b immer genau auf der Normalen der Wand zu Fluid-Partikel a ist. Resultierend daraus ist Gleichung (4.15) immer 1. Dementsprechend kann statt $B\left(x,y\right)$ direkt $\Gamma\left(y\right)$ verwendet werden.

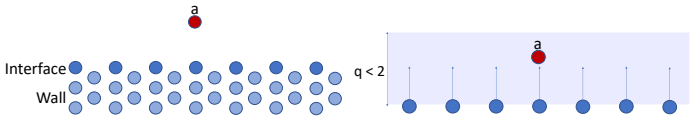
4.1.6 Vergleich

Die Ghost Partikel und Boundary Partikel erhalten die SPH Konsistenz, da die Partikel der Wand einfach in die bestehenden Gleichungen miteinbezogen werden können. Jedoch müssen die Partikel generiert werden, bei den Ghost Partikeln sogar in jedem Zeitschritt. Um die Implementierung der Wand einfach zu gestalten soll aber eine möglichst partikelfreie Methode gewählt werden. Damit Fallen die Ghost Partikel und Boundary Partikel weg. Der Penalty-Basierte, genau wie der Boundary

Force Ansatz, benötigen auch Partikel in der Simulation. Die Boundary Force lässt sich jedoch simpel partikelfrei umsetzen. Damit bleiben diese und die Lennard-Jones Repulsive Force übrig. Beide verhindern den Durchschlag von Partikeln. Nach [Sut14] hat die Lennard-Jones Repulsive Force jedoch das Problem, dass ein Partikel, das sich parallel zur Wand bewegt, nicht uniformer Normalenkraft und einer Tangetialkraft ungleich 0 ausgesetzt ist, was zu starken Störungen in der Strömung nahe der Wand führt. Die Kraft, die sich parallel zur Wand bewegende Partikel beim Boundary Force Ansatz erfahren, ist dagegen konstant. Das Problem, das diese Ansätze dagegen mit sich bringen, ist die Berechnung der Normalen, zumindest bei komplexeren Körpern. Da wir aber nur eine ebene Wand implementieren wollen, ist die Normale von vornherein klar. Daher scheint die Boundary Force der optimale Kompromiss zwischen Simplizität und Funktionalität und wurde als Teil dieser Arbeit implementiert. Für eine bessere Übersicht sind die Vor- und Nachteile in Tabelle 4.1 nochmal zusammengefasst.



(a) Repulsive Force **(b)** Ghost Partikel **(c)** Penalty-Basierend



(d) Boundary Partikel **(e)** Boundary Force

Abbildung 4.1: Skizzen der verglichenen Methoden für Randbedingungen, die für eine Rigid Wall verwendet werden können.

Methoden	**Vorteile**	**Nachteile**
Lennard-Jones Repulsive Force	Einfache Formel, keine zusätzlichen Partikel	Künstliche Kraft, ungleichmäßige Normalkraft führt zu starken Strömungen
Ghost Particles	Erhält die SPH Konsistenz	Generierung der Ghost Partikel in jedem Zeitschritt
Penalty Based	Wand existiert in Form von Partikeln	Zusätzliche Partikel, kompliziert, künstliche Kraft
Boundary Particles	Keine zusätzliche Formel, Wand existiert in Form von Partikeln	Viele zusätzliche Partikel
Boundary Force	Einfache Formel, keine zusätzlichen Partikel	Berechnung der Normalen, künstliche Kraft

Tabelle 4.1: Vergleich der möglichen Methoden für die Rigid Wall. Dabei werden die Vor- und Nachteile aufgelistet, nach denen die Entscheidung getroffen wurde.

4.2 Implementierung

Um die Implementierung zu vereinfachen wurde der bestehende Code syntaktisch angepasst. Zum einen wurden die in der Arbeit verwendeten Code-Abschnitte einem Refactoring unterzogen, um die Lesbarkeit zu erhöhen, zum anderen wurde auch das Build-System angepasst, sodass ein logischer, inkrementeller Build möglich ist. Außerdem wurde der Build-Befehl für die Verwendung mehrerer Kerne angepasst. Dies bringt eine enorme Zeitersparnis mit sich, wenn mehrfaches bauen nötig ist. Genaueres findet sich in Abschnitt 6.2

Für die eigentliche Implementierung der Wand ist es als erstes nötig die Daten der Wand aus der Konfigurationsdatei auszulesen. Ein Beispiel der finalen Version der Konfiguration der Rigid Wall, wie sie in der Input XML-Datei zu finden ist, ist in Listing 4.1 zu sehen. Wie zu Beginn des Kapitels beschrieben, soll die Wand als Ebenengleichung definiert sein. Außerdem ist noch die Masse der Wand definiert, die als m_b in Gleichung (4.11) verwendet wird und die `speed_of_sound` für das β der Γ-Funktion in Gleichung (4.16). Die `density` wird im Moment nicht verwendet und stattdessen ein von SiPER definierter Wert genutzt. Das `viscosity_alpha` wird erst in Kapitel 5 relevant und dort weiter beschrieben.

```
1  <GEOMETRY>
2    <RIGIDWALL>
3      <!--Specify the position of a rigid wall in form of ax + by + cz = d-->
4      <a>0</a>
5      <b>1</b>
6      <c>0</c>
7      <d>0.2</d>
8      <!--Mass of a virtual wall particle im kg-->
9      <mass>8e-6</mass>
10     <density>100</density>
11     <!--speed of sound at the virtual wall particle im m/s-->
12     <speed_of_sound>300</speed_of_sound>
13     <viscosity_alpha>1</viscosity_alpha>
14   </RIGIDWALL>
15 </GEOMETRY>
```

Listing 4.1: Konfigurationsabschnitt der Rigid Wall in der Input XML-Datei. In den Zeilen 3-7 ist die Definition der Ebenengleichung, die die Wand abbildet. Darunter sind weitere Attribute, die bei der Berechnung der Wand verwendet werden.

Nach dem die Daten in SiPER eingelesen sind, können sie nun verwendet werden. Dazu wurde im Ablauf (siehe Abbildung 3.3), direkt nach der Gravitation, die Berechnung der Rigid Wall Kräfte eingebaut. Die Berechnung ist in eine eigene Methode ausgelagert (siehe Listing 4.2) und wird dann direkt auf die von SiPER verwendete Predictor-Schritt Beschleunigung aufaddiert. Im Code zu sehen sind außerdem die in der Konfiguration definierten Werte der Masse und Schallgeschwindigkeit. Das `struct SPH_particle` enthält sämtliche Informationen aller Partikel, die von diesem Prozess verwaltet werden. Um auf die Werte des aktuellen Partikels zuzugreifen wird das `i` als Partikel-ID bzw. Index verwendet. Der Aufruf der Γ-Funktion (siehe Listing 4.3) wird direkt mit den Werten für β und q ausgeführt und ist ansonsten nach Gleichung (4.16) implementiert. Die Berechnung des Wandabstands y wird an einer zentralen Stelle zu Beginn eines Zeitschritts ausgeführt, indem für jedes Partikel die Funktion aus Listing 4.4 ausgeführt wird.

```
1  double* SPH_rigid_wall_forces_i(struct SPH_particle *part,
2    struct SPH_rigid_wall *rigid_wall, const int i, double *force) {
3    double y = part->rigid_wall_distance[i];
4    double q = y / part->smoothinglength;
5    vecSetAll(0.0, force);
6    if (q <= 0.0 || q >= 2.0) {
7      return force;
8    }
9
10   double c = rigid_wall->speed_of_sound;
11   double beta = 0.02 * c * c / y;
12   double gamma = Gamma(beta, q);
13   double multiplier = gamma * rigid_wall->mass / (rigid_wall->mass + part->m[i][0]) / y;
14
15   vecMultiply(rigid_wall->n, multiplier, force);
16   return force;
17 }
```

Listing 4.2: Implementierung der Funktion zur Berechnung der Rigid Wall Kraft für ein Partikel i. In Zeile 6 findet sich das Abbruchkriterium, sodass nur eine Kraft entsteht, wenn das Partikel i nah genug an der Wand ist. Ist das Partikel i nah genug an der Wand, so wird die absolute Kraft berechnet und mit dem Normalenvektor der Rigid Wall multipliziert.

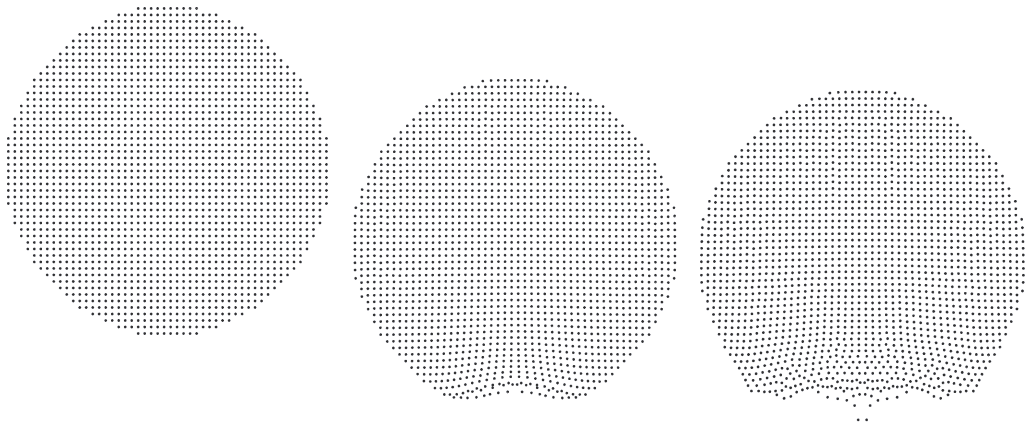
```
1  double Gamma(double beta, double q) {
2    if (q < 2.0 / 3.0) {
3      return 2.0 / 3.0 * beta;
4    }
5    if (q < 1.0) {
6      return beta * (2.0 * q - 3.0 / 2.0 * q * q);
7    }
8
9    return 1.0 / 2.0 * beta * (2.0 - q) * (2.0 - q);
10 }
```

Listing 4.3: Implementierung der Γ-Funktion nach Gleichung (4.16). β und q wurden bereits berechnet und werden als Parameter übergeben.

```
1  void update_rigid_wall_distance_i(const struct SPH_rigid_wall *rigid_wall,
2    const struct SPH_particle *part, int i) {
3    part->rigid_wall_distance[i] =
4      fabs(vecDistanceBetweenPointAndPlane(part->x[i], rigid_wall->n, rigid_wall->d));
5  }
```

Listing 4.4: Funktion zur Berechnung des Wandabstands eines Partikels i. Dafür werden die Partikel-Position x und die Ebenengleichung der Wand an eine Funktion zu Abstandsberechnung übergeben und das Ergebnis zentral abgelegt.

Das Ergebnis dieser sehr einfachen Implementierung ist in Abbildung 4.2 zu sehen. Wie daraus hervorgeht, reagieren die Partikel des Tropfens auf die Wand mit einem starken Abprallen. Da die Partikel durch die nachkommenden Partikel allerdings wieder nach unten gedrückt werden, entsteht eine Auf- und-Abwärtsbewegung. Die Amplitude dieser Bewegung wird sehr schnell größer, bis die Geschwindigkeit einiger Partikel mit über $10^{12}\frac{m}{s}$ zu groß wird und die Simulation abbricht. Abbildung 4.2c zeigt die Partikel bereits unmittelbar vor dem Abbruch. Die Simulation eines längeren Zeitraums konnte mit dieser Implementierung nicht erreicht werden. Eine simple Erklärung für dieses Verhalten ist der Schock, der beim Auftreffen auf die Wand entsteht. Dieser Schock wird nicht gedämpft und erzeugt auch sonst keine Reaktion im System, da die Wand weder in den Druck noch in die Viskosität miteinbezogen wird.



(a) t = 0ms **(b)** t = 212ms **(c)** t = 230ms

Abbildung 4.2: Der Tropfen prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die rein mit der Boundary Force modelliert wird und somit nicht sichtbar ist. Dabei ist ein starkes Abprallen zu beobachten und auch Partikel, die durch die Wand durchdringen.

4.3 Zusammenfassung

Zur Simulation des Hochgeschwindigkeitsaufpralls wurde zunächst eine einfache Rigid Wall, in Form einer Ebene im Raum, implementiert. Dazu wurde aus den fünf gefundenen Methoden, der Boundary Force Ansatz gewählt, da er gut funktionieren soll und in einfachster Form auch ohne zusätzliche Partikel auskommt. Die SiPER Konfigurationsdatei wurde erweitert, sodass auch die Rigid Wall darüber konfiguriert werden kann. Die Gleichungen der Boundary Force wurden implementiert und die Details der Implementierung aufgezeigt. Der Test der Implementierung mit dem zuvor erstellten Testfall zeigt ein starkes Abprallen der Partikel an der Wand und nach kurzer Zeit einen Simulationsabbruch. Diese Probleme werden auf die künstliche Kraft der Wand zurückgeführt, die keine Reaktion im System auslöst. Die nötigen Anpassungen, um eine Systemreaktion zu erzeugen werden in Kapitel 5 beschrieben.

5 Künstliche Viskosität

Um die Wand besser ins System zu integrieren und um einen Schock zu vermeiden, muss die künstliche Kraft mit weiteren künstlichen Reaktionen kombiniert werden. Die künstliche Viskosität ist ein verbreiteter Ansatz, um physikalisch nicht korrekte Schwingungen zu beseitigen oder zumindest zu reduzieren. Da eine künstliche Viskosität schnell zu stark werden kann, wird sie soweit wie möglich vermieden, wenn realistische Modelle simuliert werden sollen. Dies ist besonders relevant, wenn Turbulenzen oder Strömungen simuliert werden. In Fällen starker Kompression, also einem Schock, wie er an der Rigid Wall entsteht, kann die künstliche Viskosität aber notwendig sein, um die Stabilität der Simulation zu erhalten. Da wir bisher ein Abprallen der Partikel und einen Absturz der Simulation beobachten, wollen wir eine solche künstliche Viskosität in das System einführen. Außerdem soll die Wand auch in die SPH-Gleichungen miteinbezogen werden.

5.1 Rigid Wall Partikel

Als erste Anpassung wird die Boundary Force in ihrem ursprünglichen Penalty-Basierten Ansatz, mit Partikeln am Ort der Wand, verwendet, um diese Partikel in die SPH-Gleichungen miteinzubeziehen. Dafür werden innerhalb des Simulationsraums Partikel, auf der durch die Ebenengleichung definierten Ebene, generiert. Die Funktion wurde erweitert, sodass, wie bei den Boundary Partikeln in Abschnitt 4.1.4, zwei Partikelarten generiert werden, auch wenn wir es hier im Vergleich zu Abbildung 4.1d bei einem uniformen Gitter belassen. Die Berechnung der Rigid Wall Kräfte bleibt unverändert. Die Partikel sollen bewirken, dass die Wand einen Einfluss auf die Druckgleichung (siehe Gleichung (3.1)) nimmt. Da die Partikel, die in den Grenzbereich der Wand eindringen, direkt eine Druckveränderung erfahren, bleibt die Schockreaktion aus. Die in Listing 5.1 dargestellte Funktion ist die 2D Implementierung, die die SiPER Partikel-Art und -Position bestimmt. Die daraus erzeugten Partikel werden dann regulär in der bestehenden SiPER Implementierung verwendet.

```
1  double* calculate_rigid_wall_particle_grid(const struct SPH_rigid_wall *rigid_wall,
2   const struct SPH_cell *cell, const double grid_spacing, int *count) {
3    *count = 0;
4    double *positions = malloc(*count * (DIM + 1) * sizeof(double));
5    int index = 0;
6    for (int i = 0; i < 3; i++) {
7      double a1 = 0.0;
8      double a2 = (rigid_wall->d - i * grid_spacing) / rigid_wall->n[1];
9      double u1 = 1.0;
10     double u2 = -rigid_wall->n[0] / rigid_wall->n[1];
11
12     double lambda1 = -a1 / u1;
13     double lambda2 = -a2 / u2;
14     double min_lambda = fmin(lambda1, lambda2);
15
16     lambda1 = (cell->domain[0] - a1) / u1;
17     lambda2 = (cell->domain[1] - a2) / u2;
18     double max_lambda = fmax(lambda1, lambda2);
19
20     *count += MAX((int) ((max_lambda - min_lambda) / grid_spacing), 0);
21
22     positions = realloc(positions, *count * (DIM + 1) * sizeof(double));
23
24     for (double lambda = min_lambda; lambda < max_lambda; lambda += grid_spacing) {
25       positions[index++] = (double) (i == 0 ? WALL_BOUNDARY : WALL_REAL);
26       positions[index++] = a1 + lambda * u1;
27       positions[index++] = a2 + lambda * u2;
28     }
29   }
30   return positions;
31 }
```

Listing 5.1: Implementierung der Funktion zur Generierung von Wand-Partikel in 2D. Dafür wird die Ebenengleichung in 2D zu einer Strecke gekürzt, die innerhalb des Rechenraums liegt. Auf dieser Strecke werden dann im Initialabstand Partikel platziert. Dabei werden drei Schichten generiert, wobei die Erste vom Typ WALL_BOUNDARY ist, was den Interface-Partikeln entspricht. Die übrigen zwei Schichten sind vom Typ WALL_REAL, was die eigentlichen Wall-Partikel sind. Typ und Position werden in einem Array abgelegt und zurückgegeben.

5.2 Künstliche Viskosität

Eine weitere Anpassung ist die Einführung einer künstlichen Viskosität. Diese soll den Aufprall des Tropfens dämpfen, sodass die Partikel in der Nähe der Wand langsamer werden und die Wand eine Geschwindigkeit von 0 zur Viskosität beiträgt. Künstliche Viskosität wurde im ursprünglichen bereits von Monaghan [MG83] vorgeschlagen. Demnach wird auf die Druckgleichung der Viskositätsterm

$$\Pi_{ab} = -\alpha \frac{hc_a}{\rho_a} \frac{\vec{u}_{ab}}{\vec{r}_{ab}} \tag{5.1}$$

aufaddiert. Dabei ist α das in Abschnitt 4.2 bereits erwähnte `artificial_viscosity_alpha`, welches als Faktor der Viskosität agiert. Mit dieser Formulierung entstehen nach [MG83] zwei Probleme. Zum einen kann das $\vec{r}_{ab}$ im Nenner zu einer zu großen Viskosität führen und zum anderen ist $\Pi_{ab} \neq \Pi_{ba}$, was dazu führt, dass der Impuls nicht mehr implizit erhalten bleibt. Um diese Probleme anzugehen wird

$$\frac{1}{\vec{r}_{ab}} \text{ durch } \frac{\vec{r}_{ab}}{\vec{r}_{ab}^2 + \varepsilon h^2} \tag{5.2}$$

und ρ_a durch $\bar{\rho}_{ab}$ ersetzt, wobei der Überstrich den Mittelwert aus a und b beschreibt. Auch c_a sollte durch $\bar{c}_{ab}$ ersetzt werden, jedoch ist in SiPER keine Schallgeschwindigkeit definiert, weshalb wir die in der Rigid Wall Konfiguration definierte Schallgeschwindigkeit verwenden und c_a schlicht durch c ersetzten. Daraus resultiert die künstliche Viskosität, wie sie in dieser Arbeit verwendet wird als

$$\Pi_{ab} = \begin{cases} -\alpha \frac{hc}{\bar{\rho}_{ab}} \frac{\vec{u}_{ab}\vec{r}_{ab}}{\vec{r}_{ab}^2 + \varepsilon h^2} & \vec{u}_{ab}\vec{r}_{ab} < 0 \\ 0 & sonst \end{cases} . \tag{5.3}$$

Als ε verwenden wir das von Monaghan [MG83] vorgeschlagene $\varepsilon = 0.01$. Den Faktor α lassen wir an dieser Stelle erst einmal offen und führen Testreihen mit verschiedenen Werten durch. Da wir ISPH verwenden, kann Π_{ab} nicht einfach auf die Druckgleichung aufaddiert werden, da diese im Corrector-Schritt berechnet wird und wir die Viskosität schon im Predictor-Schritt benötigen. Deshalb werden wir den durch die Viskosität entstehenden Druck und die daraus entstehende Druckkraft explizit im Predictor-Schritt berechnen.

5.3 Implementierung

Zur Berechnung der künstlichen Viskosität wird die Funktion aus Listing 5.2 für jedes Partikel ausgeführt. Dabei iterieren wird über alle Nachbarpartikel und berechnen Gleichung (5.3).

```
1  double calculate_artificial_viscosity_pressure_i(const struct SPH_particle *part,
2   const struct SPH_rigid_wall *rigid_wall, const int i) {
3    double alpha = rigid_wall->viscosity_alpha;
4    double epsilon = 0.01;
5    double h = part->smoothinglength;
6    double c = rigid_wall->speed_of_sound;
7
8    double artificial_viscosity = 0.0;
9    for (int n = 0; n < part->n_nb[i]; n++) {
10     int j = part->nb[i][n];
11     double speed_diff[DIM];
12     double vr = vecDot(vecSubtract(part->v[i], part->v[j], speed_diff), part->r[i][n]);
13     if (vr >= 0.0) {
14       continue;
15     }
16     double density = 0.5 * (part->rho[i][0] + part->rho[j][0]);
17
18     artificial_viscosity += -alpha * h * c / density * vr /
19      (vecDot(part->r[i][n], part->r[i][n]) + epsilon * h * h);
20   }
21
22   return artificial_viscosity;
23 }
```

Listing 5.2: Implementierung der Funktion zur Berechnung des Drucks der künstlichen Viskosität eines Partikels i. Dabei wird über alle Nachbarn von i iteriert und jeweils Gleichung (5.3) gelöst und aufsummiert.

Den daraus resultierenden Druck wollen wir verwenden um die Druckkraft zu berechnen. Da die SiPER-Funktion zur Druckkraftberechnung diese direkt in Corrector-Beschleunigung ablegt und auch direkt den Druck der Partikel verwendet müssen wir einen Workaround verwenden. Dafür wird zunächst der Druck der künstlichen Viskosität im Druck-Attribut der Partikel ablegt, um aus diesem die Druckkraft berechnen zu lassen. Das Ergebnis der Druckkraftberechnung befindet sich dann in der Corrector-Beschleunigung. Also addieren wir die Corrector-Beschleunigung auf die Predictor-Beschleunigung. Wie in Listing 5.3 gezeigt, wird für alle Fluid-Partikel zunächst die in Abschnitt 4.2 beschriebene Rigid Wall Kraft auf die Predictor-Beschleunigung addiert. Dann wird der Druck der künstlichen Viskosität als Druck des jeweiligen Partikels gesetzt. Erst wenn der Druck aller Partikel gesetzt wurde, wird die Druckkraft durch eine bestehende SiPER-Funktion berechnet und das Ergebnis, das sich in `accCorrector` befindet, auf die Predictor-Beschleunigung `accPredictor` addiert. Damit ist die künstliche Viskosität im Predictor-Schritt implementiert.

```
void SPH_forces_predictor(PetscMPIInt rank, PetscMPIInt size, struct SPH_particle *part,
 struct SPH_cell *cell, struct SPH_time *time, struct SPH_comm *comm,
 struct SPH_phase *phases, struct SPH_boundary *boundary,
 struct SPH_rigid_wall *rigid_wall, struct SPH_physicalEOS *physEOS, double *g,
 struct SPH_model *models, double **sigma) {
  ...
  // Predictor loop
  for (int i = 0; i < part->nP_withoutHalo; i++) {
    ...
    if (part->C[i][0] <= FLUID_ONLY) {
      ...
      double force[DIM];
      vecIncrement(part->accPredictor[i],
       SPH_rigid_wall_forces_i(part, rigid_wall, i, force));
      part->p[i][0] = calculate_artificial_viscosity_pressure_i(part, rigid_wall, i);
    }
  }

  SPH_pressure_force(rank, size, part, cell, time, models, phases, boundary,
   rigid_wall, physEOS);
  for (int i = 0; i < part->nP_withoutHalo; i++) {
    vecIncrement(part->accPredictor[i], part->accCorrector[i]);
  }
  ...
}
```

Listing 5.3: Implementierung zur Berechnung der Kraft, die durch die künstliche Viskosität entsteht. In der ersten for-Schleife wird die Kraft der Rigid Wall auf die Predictor-Beschleunigung addiert und der Druck der Partikel durch die künstliche Viskosität gesetzt. Vor der zweiten for-Schleife wird dann die Druckkraft aller Partikel berechnet und implizit in der Corrector-Beschleunigung abgelegt. Diese wird dann in der zweiten for-Schleife auf die Predictor-Beschleunigung addiert.

5.4 Zusammenfassung

Um eine Systemreaktion hervorzurufen und den Schock im System zu dämpfen, wurde die Rigid Wall auf ihre ursprünglich angedachte Form erweitert. Dazu werden zu Beginn einer Simulation Partikel auf der Ebene der Rigid Wall generiert. Diese Boundary Partikel werden in alle SPH-Gleichungen miteinbezogen. Außerdem wurde eine künstliche Viskosität eingeführt, die den Schock dämpft, indem sie Partikel, die auf die Wand zufliegen, bremst. Die gezeigten Verbesserungen wurden implementiert und die genaue Implementierung aufgezeigt. Dabei wurden soweit wie möglich bestehende SiPER Implementierungen genutzt. Die Ergebnisse dieser Verbesserungen sind gleichzeitig die Ergebnisse der Arbeit und finden sich in Kapitel 6.

6 Ergebnis

Da diverse Komplikationen entstanden sind, werden die Ergebnisse im Folgenden in zwei Abschnitte unterteilt. Zum einen in Abschnitt 6.1 die Rigid Wall Implementierung und die daraus resultierenden Beobachtungen, zum anderen in Abschnitt 6.2 die Erkenntnisse aus der Arbeit mit SiPER.

6.1 Rigid Wall

Als Ergebnis betrachten wir die Simulation mit verschiedenen Werten für den Faktor der Viskosität α, also mit verschieden starker Viskosität. Dadurch soll gezeigt werden, welchen Einfluss die Stärke der Viskosität auf das Verhalten des Tropfens beim Aufprall hat. Dies ist wichtig, da das korrekte Aufprallverhalten nur durch ein passend gewähltes α erreicht werden kann. Außerdem sollte die künstliche Viskosität, als Dämpfung für den Schock der Partikel, den Absturz der Simulation verhindern. Die Tests zu den Werten $\alpha = 0$, $\alpha = 1$ und $\alpha = 100$ sind in Abbildung 6.2, Abbildung 6.3 und Abbildung 6.4 dargestellt. Wie in allen drei Fällen zu sehen, ist das in Abschnitt 4.2 beobachtete übertriebene Abprallen der Partikel verschwunden und Partikel kommen an der Wand zum Stehen. Allerdings bricht auch hier die Simulation unmittelbar nach dem letzten Zeitschritt der Abbildungen ab. Die Farbe der Partikel in den Abbildungen repräsentiert das Druck-Attribut mit der Skala aus Abbildung 6.1. In allen Fällen ist zu sehen, dass kein gleichmäßiges Druckfeld entsteht, sondern ein Durcheinander verschiedener Druckwerte. Da die Werte auch negative und extrem große Werte beinhalten, ist das vermutlich der Grund, der für den Absturz der Simulation sorgt. Um Implementierungsfehler auszuschließen, wurde der Test auch noch mit der existierenden SiPER No-Slip Domain Boundary durchgeführt, die den Rand des Rechenraums begrenzt, allerdings mit dem selbem Ergebnis. Eventuell entsteht dieses Fehlverhalten durch Fehler, die im Log von SiPER erscheinen, während die Simulation läuft. Demnach sind einige Werte in der Speichermatrix invalide. Die Ursache dafür konnte im Rahmen dieser Arbeit jedoch, auch in Kooperation mit dem ICVT, nicht ermittelt werden, resultierend darin, dass das Problem auch nicht behoben werden konnte.

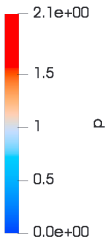
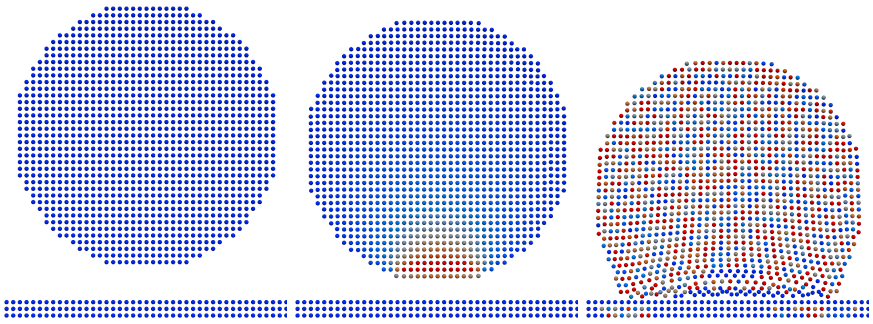
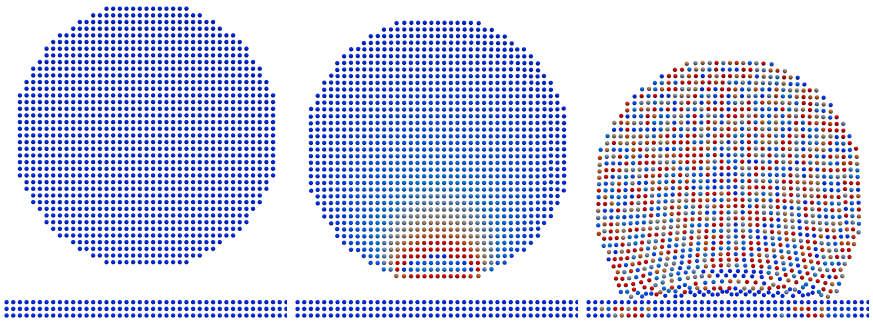


Abbildung 6.1: Die Druckskala, die in den Tests der Rigid Wall verwendet wird.



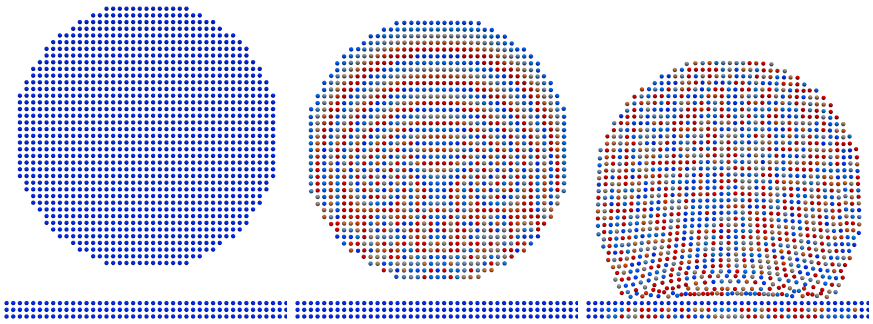
(a) t = 0ms **(b)** t = 41ms **(c)** t = 166ms

Abbildung 6.2: Der Tropfen mit 1245 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 0$ besitzt. Dabei ist kein Abprallen zu beobachten und initial ein Druckanstieg am unteren Rand des Tropfens.



(a) t = 0ms **(b)** t = 41ms **(c)** t = 166ms

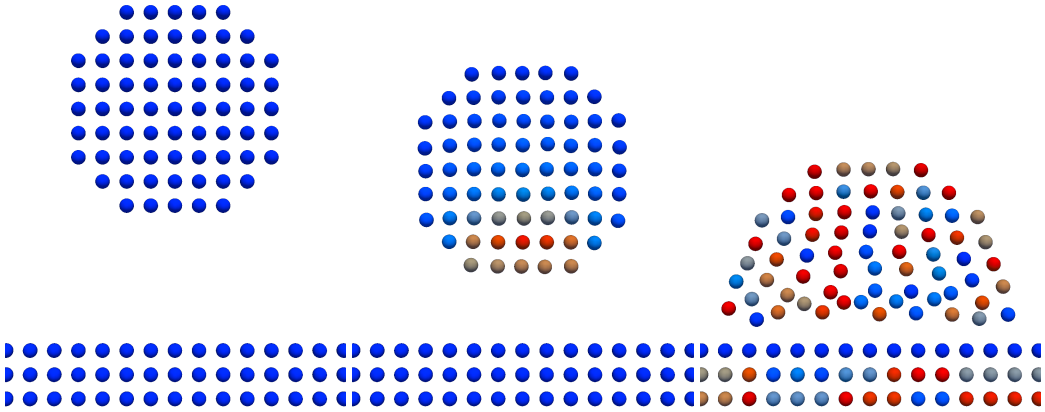
Abbildung 6.3: Der Tropfen mit 1245 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 1$ besitzt. Dabei ist kein Abprallen zu beobachten und initial ein Druckanstieg am unteren Rand des Tropfens.



(a) t = 0ms **(b)** t = 41ms **(c)** t = 166ms

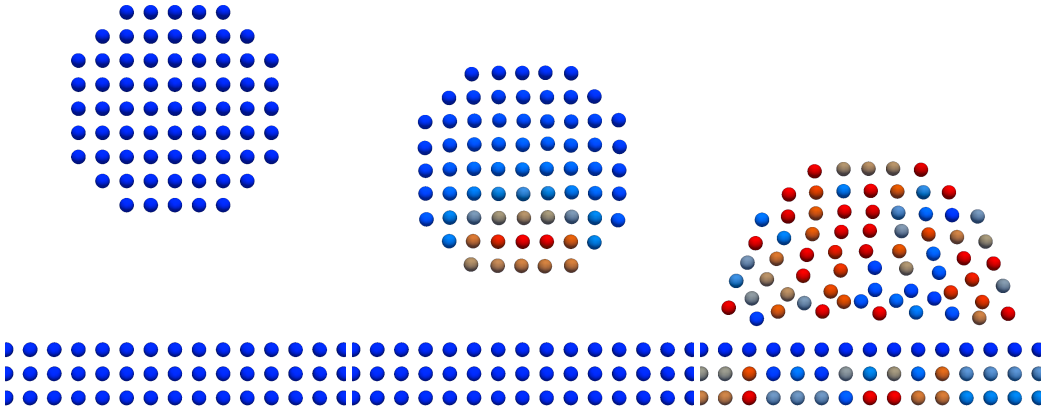
Abbildung 6.4: Der Tropfen mit 1245 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 100$ besitzt. Dabei ist kein Abprallen zu beobachten, jedoch ist der Druck schon früher nicht mehr geordnet.

Für eine bessere Überschaubarkeit wurde die gleiche Testreihe noch mit dem kleineren Tropfen aus 69 Partikeln durchgeführt, zu sehen in Abbildung 6.5, Abbildung 6.6 und Abbildung 6.7. Dies soll ermöglichen, nicht nur den gesamten Tropfen zu betrachten, sondern auch das Verhalten der einzelnen Partikel, vor allem das Druck Attribut. Wir beobachten dennoch dasselbe Verhalten: Der Tropfen erfährt im unteren Bereich einen Druckanstieg, wenn er sich in die Nähe der Wand bewegt. Auch der Druck der Wand-Partikel erhöht sich. Es gibt kein starkes Abprallen und der Tropfen scheint an der Wand zu zerlaufen. Kurz darauf zerspringt das Druckfeld jedoch wieder vollständig und die Simulation bricht ab. Die Druckskala ist die bereits in den vorherigen Abbildungen verwendete Skala aus Abbildung 6.1.



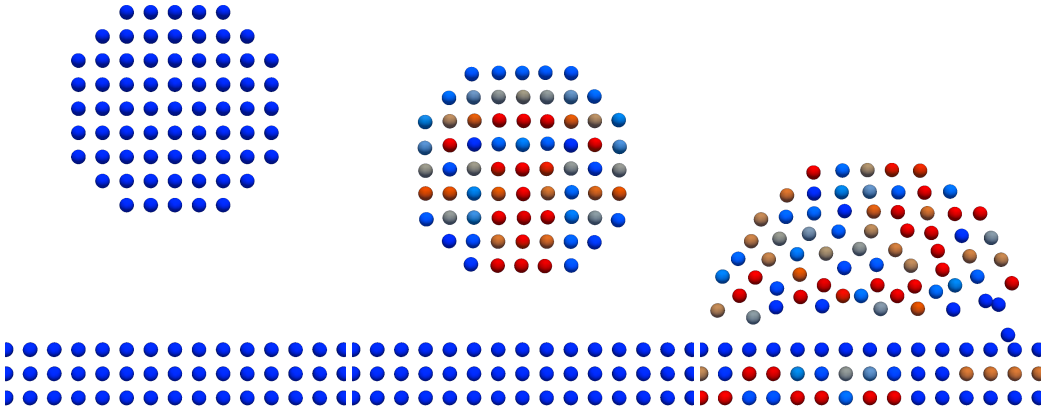
(a) t = 0ms **(b)** t = 50ms **(c)** t = 150ms

Abbildung 6.5: Der Tropfen mit 69 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 0$ besitzt. Dabei ist kein Abprallen zu beobachten und initial ein Druckanstieg am unteren Rand des Tropfens.



(a) t = 0ms **(b)** t = 50ms **(c)** t = 150ms

Abbildung 6.6: Der Tropfen mit 69 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 1$ besitzt. Dabei ist kein Abprallen zu beobachten und initial ein Druckanstieg am unteren Rand des Tropfens.



(a) t = 0ms **(b)** t = 50ms **(c)** t = 150ms

Abbildung 6.7: Der Tropfen mit 69 Partikeln prallt mit 1 $\frac{m}{s}$ gegen die Rigid Wall, die durch Boundary Partikel abgebildet ist und ein $\alpha = 100$ besitzt. Dabei ist kein Abprallen zu beobachten, jedoch hat sich der Druckanstieg schon früh durch den ganzen Tropfen ausgebreitet, sodass die Ordnung schon früh verloren geht.

6.2 SiPER

Das SiPER-Programm ist eine komplexe und mittlerweile sehr umfangreiche Software. Daher würde eine Dokumentation die Arbeit mit SiPER stark vereinfachen, vor allem wenn tiefere Eingriffe in den Code vorgenommen werden müssen. Auch wenn schon große Modelle mit SiPER simuliert worden sind, befindet sich die Software noch mitten in der Entwicklung. Zum Beispiel wurden die XML-Input-Dateien zum Zeitpunkt der Arbeit gerade eingeführt, was eine wirklich grundlegende Strukturänderung der Bedienung darstellt. Zuvor wurden noch einfache Textdateien verwendet.

Die starke Verwendung des Präprozessors für verschiedene Dimensionen, vgl. Listing 6.1, könnte durch allgemeine Vektorlösungen, vgl. Listing 6.2, stark reduziert werden. Damit würden Codeduplikate entfallen und die Lesbarkeit erhöht werden. Dieses Beispiel zeigt außerdem, dass der Code an einigen Stellen generell sehr viel einfacher und übersichtlicher gestaltet werden kann, indem nur sich unterscheidende Codeabschnitt aufgeteilt werden. Auch das bessere Beispiel der allgemeinen Vektorlösung bietet noch Potential zur Verbesserung. Die for-Schleife kann noch aufgelöst werden, wenn die Unterscheidung, welcher Kernel verwendet werden soll, an einer zentraleren Stelle beschrieben wird. Im Allgemeinen würde es die Codekomplexität reduzieren, wenn die Wahl des richtigen Kernels gekapselt werden würde. Auch Funktionen, die nur einen Wert einer Variablen zuweisen, können aufgelöst werden, um direkt erkenntlich zu machen was an der Stelle passiert. Solche Konstrukte aufzulösen und sprechende, kürzere Bezeichner zu verwenden würde nach McConnell [McC04] die Lesbarkeit stark erhöhen und die Codekomplexität reduzieren. Damit kann die Arbeit mit SiPER stark erleichtert und auch die Zuverlässigkeit und Ergebniskonsistenz erhöht werden. An neueren Codestellen wurden solche Maßnahmen teilweise auch schon umgesetzt. Die Erstellung einer Dokumentation wurde ebenfalls begonnen.

```
#ifdef DIM1
#ifdef CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dWS_NCSPH(part,i,l,0);
#else //CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dW(part,i,l,0);
#endif //CORRECTED_SPH
  v_x_r = (part->v[i][0]-part->v[j][0]) * part->r[i][l][0];
  v_x_r_wall = (part->v[i][0]) * part->r[i][l][0];
#endif //DIM1
#ifdef DIM2
#ifdef CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dWS_NCSPH(part,i,l,0)
   + part->r[i][l][1] * dWS_NCSPH(part,i,l,1);
#else //CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dW(part,i,l,0) + part->r[i][l][1] * dW(part,i,l,1);
#endif //CORRECTED_SPH
  v_x_r = (part->v[i][0]-part->v[j][0]) * part->r[i][l][0]
   + (part->v[i][1]-part->v[j][1]) * part->r[i][l][1];
  v_x_r_wall = (part->v[i][0]) * part->r[i][l][0] + (part->v[i][1]) * part->r[i][l][1];
#endif // DIM2
#ifdef DIM3
#ifdef CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dWS_NCSPH(part,i,l,0)
   + part->r[i][l][1] * dWS_NCSPH(part,i,l,1) + part->r[i][l][2] * dWS_NCSPH(part,i,l,2);
#else //CORRECTED_SPH
  r_x_dW = part->r[i][l][0] * dW(part,i,l,0)
   + part->r[i][l][1] * dW(part,i,l,1) + part->r[i][l][2] * dW(part,i,l,2);
#endif //CORRECTED_SPH
  v_x_r = (part->v[i][0]-part->v[j][0]) * part->r[i][l][0]
   + (part->v[i][1] - part->v[j][1]) * part->r[i][l][1]
   + (part->v[i][2] - part->v[j][2]) * part->r[i][l][2];
  v_x_r_wall = (part->v[i][0]) * part->r[i][l][0]
   + (part->v[i][1]) * part->r[i][l][1] + (part->v[i][2]) * part->r[i][l][2];
#endif //DIM3
```

Listing 6.1: Der Code zeigt die Berechnung der Variablen `r_x_dW`, `v_x_r` und `v_x_r_wall`, wie sie im ursprünglichen SiPER Code vorkommt. Dabei unterscheidet sich die Berechnung, wenn `CORRECTED_SPH` verwendet wird. Diese Unterscheidung wird mit `#ifdef` gelöst. Zusätzlich wird noch eine Unterscheidung für 1D, 2D und 3D gemacht.

```
1   r_x_dW = 0.0;
2   for (int d = 0; d < DIM; d++) {
3 #ifdef CORRECTED_SPH
4     r_x_dW += part->r[i][l][d] * dWS_NCSPH(part, i, l, d);
5 #else //CORRECTED_SPH
6     r_x_dW += part->r[i][l][d] * dW(part, i, l, d);
7 #endif //CORRECTED_SPH
8   }
9   double v_diff[DIM];
10  v_x_r = vecDot(vecSubtract(part->v[i], part->v[j], v_diff), part->r[i][l]);
11  v_x_r_wall = vecDot(part->v[i], part->r[i][l]);
```

Listing 6.2: Der Code zeigt die gleiche Berechnung der Variablen `r_x_dW`, `v_x_r` und `v_x_r_wall`, wie sie in Listing 6.1 gezeigt wird, nach einem Refactoring. Die Unterscheidung nach `CORRECTED_SPH` findet sich nun innerhalb einer for-Schleife, die über die Dimensionen iteriert. Die Berechnung der Skalarprodukte von `v_x_r` und `v_x_r_wall` ist nun in allgemeine Vektor-Funktionen gekapselt, die bereits mit der Dimensionsunterscheidung umgehen können.

Bereits angesprochen wurde das Build-System. Dieses wurde im Zuge der Arbeit überarbeitet, sodass jede c- und h-Datei nur wirklich benötigte Abhängigkeiten inkludiert und diese Abhängigkeiten automatisch bei einem Build berücksichtigt werden. Außerdem wurde das Build-Skript so angepasst, dass es automatisch alle verfügbaren Kerne zur Kompilierung verwendet. So wurde die Zeit eines Clean Builds auf dem verwendeten Ryzen R7 1800X System stark reduziert. Nach eigenen Messungen von 110 Sekunden auf 50 Sekunden. Bei einem inkrementellen Build kann der Build nun auch nur wenige Sekunden benötigen, im Vergleich zu einer Clean Build Zeit, wenn Header-Dateien verändert werden.

Ein weitere Verbesserungsmöglichkeit biete die Datenmatrix. Die Partikel in ein Partikel-Konstrukt zu kapseln, sodass alle Attribute eines Partikels an einem Ort liegen, würde es klar ersichtlich machen, wie die Attribute eines Partikels belegt sind. Dadurch könnte das Debuggen enorm erleichtert werden, da alle Werte des betrachteten Partikels direkt eingesehen werden können. Diese Struktur würde auch näher an der SPH-Betrachtungsweise liegen, sodass eine Menge von Partikeln mit Attributen gehalten wird. Anders als beim bisherigen Ansatz, bei dem es eine Menge von Attribut-Mengen gibt, die über die Partikel-ID einem Partikel zugeordnet werden können.

7 Zusammenfassung und Empfehlung

Die SPH-Methode, wurde als Alternative der Finite-Element-Methode, bei der Simulation eines Hochgeschwindigkeitsaufpralls betrachtet. Dabei soll vor allem die Korrektheit des Bruchverhaltens des Projektils betrachtet werden, dass bei der Finite-Element-Methode Schwierigkeiten bereitet. Daher wurde zum besseren Verständnis der gelösten Gleichungen eine quelloffene SPH Codebasis gesucht und mit dem Simulationsprogramm SiPER des ICVT gefunden. Da SiPER nur Fluide und Gase simulieren kann, muss das Programm entsprechend erweitert werden. Genau diese Erweiterungen wurden in dieser Arbeit betrachtet und deren Implementierung versucht. Zunächst sollte eine Rigid Wall implementiert werden, auf der der Aufprall stattfinden soll. Danach sollte ein Feststoffmodell implementiert werden, um das Projektil simulieren zu können. Abschließend sollte zur Validierung ein Taylor Bar Impact Test durchgeführt werden.

Als Erstes wurde die SiPER Entwicklungsumgebung aufgesetzt. Da die Arbeit in der ausgelieferten Virtual Machine mit Speicherproblemen Schwierigkeiten bereitet hat, wurde die Entwicklungsumgebung manuell in das WSL installiert. Außerdem wurden alle benötigten Pakete installiert und alle notwendigen Maßnahmen ergriffen, um ein grafisches Debuggen über die Eclipse CDT zu ermöglichen. Außerdem wurde das Build-System überarbeitet, um den Zeitaufwand des Bauens zu reduzieren. Um die Implementierung der Arbeit zu testen, wurde ein Testfall erstellt und die genaue Konfiguration betrachtet. Zu diesem Zweck wurde außerdem ein Programm entwickelt, mit dem sich Tropfen als Partikel-Gitter, das in SiPER eingelesen werden kann, generieren lassen. Um die Arbeit mit der Codebasis zu erleichtern, wurden die verwendeten Abschnitte einem Refactoring unterzogen.

Für die Modellierung der Rigid Wall wurden in der Literatur fünf Methoden gefunden und genauer betrachtet. Als bester Kompromiss zwischen Funktionalität und Simplizität, wurde die Boundary Force ausgewählt, deren Vorteil darin besteht, dass sie nicht zwingend zusätzliche Partikel benötigt und auch auf sich parallel zur Wand bewegende Partikel eine konstante Kraft ausübt. Diese Methode wurde implementiert und zunächst mit dem Aufprall eines Wassertropfens getestet. Dabei wurde ein starkes Abprallen der Partikel beobachtet, gefolgt von einem Absturz der Simulation. Diese Probleme wurden in der künstlichen Kraft der Boundary Force vermutet. Daher wurde die künstliche Viskosität als mögliche Lösung in Betracht gezogen. Diese wurde implementiert, indem die Boundary Force auf ihre ursprüngliche Form, mit Wand-Partikeln, erweitert wurde. Dafür werden auf der Ebene der Wand Partikel generiert. Außerdem wurde die künstliche Viskosität eingebaut, die einen Druck und somit eine Kraft auf die Partikel in der Nähe der Wand wirken lässt. Durch diese Verbesserungen konnte das Abprallen der Partikel verhindert werden, jedoch nicht der Absturz der Simulation. Dieser ist eventuell auf Fehler zurückzuführen, die durch nicht gesetzte Werte entstehen. Die Fehler konnten im Rahmen der Arbeit jedoch nicht behoben werden, weshalb die Implementierung eines Feststoffmodells und die Aufprallvalidierung mit Hilfe des Taylor Bar Impact Tests nicht mehr angegangen wurden. Ein funktionaler Stand, der einen vollständigen Aufprall simulieren kann, konnte nicht erreicht werden.

Während der Arbeit mit SiPER konnten einige Vorschläge zur Verbesserung der SiPER Codebasis gesammelt werden. Gerade eine Dokumentation und ein Refactoring könnten SiPER flexibler und leichter verständlich machen. Damit wären Erweiterungen leichter umsetzbar und SiPER könnte auch externen Entwicklern und Interessenten geöffnet werden.

Literaturverzeichnis

[BAA+17] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Kaushik et al. *„Petsc users manual revision 3.8“*. 2017 (zitiert auf S. 1).

[Bec18] M. Becker. „Numerical Ricochet Model of a 7.62mm Projectile Penetrating an Armor Steel Plate“. In: Precedings 15th International LS-DYNA Conference, 2018 (zitiert auf S. 1).

[BL99] J. Bonet, T.-S. Lok. „Variational and momentum preservation aspects of smooth particle hydrodynamic formulations“. In: *Computer Methods in applied mechanics and engineering* 180.1-2 (1999), S. 97–115 (zitiert auf S. 13, 14).

[BY93] T. Belytschko, I. Yeh. „The splitting pinball method for contact-impact problems“. In: *Computer methods in applied mechanics and engineering* 105.3 (1993), S. 375–393 (zitiert auf S. 20).

[CL03] A. Colagrossi, M. Landrini. „Numerical simulation of interfacial flows by smoothed particle hydrodynamics“. In: *Journal of computational physics* 191.2 (2003), S. 448–475 (zitiert auf S. 20).

[CR99] S. J. Cummins, M. Rudman. „An SPH projection method“. In: *Journal of computational physics* 152.2 (1999), S. 584–607 (zitiert auf S. 9).

[CVL00] J. Campbell, R. Vignjevic, L. Libersky. „A contact algorithm for smoothed particle hydrodynamics“. In: *Computer methods in applied mechanics and engineering* 184.1 (2000), S. 49–65 (zitiert auf S. 20, 21).

[CWRS16] S. Cooley, M. Wojciakowski, T. Raj, M. Satran. *„Windows Subsystem for Linux Documentation“*. Microsoft. 2016. URL: https://docs.microsoft.com/en-us/windows/wsl/about (zitiert auf S. 5).

[HKH+13] M. Hirschler, F. Keller, M. Huber, W. Säckel, U. Nieken. „Ein gitterfreies berechnungsverfahren zur simulation von koaleszenz in mehrphasensystemen“. In: *Chemie Ingenieur Technik* 7.85 (2013), S. 1099–1106 (zitiert auf S. 1).

[HKH+16] M. Hirschler, P. Kunz, M. Huber, F. Hahn, U. Nieken. „Open boundary conditions for ISPH and their application to micro-flow“. In: *Journal of Computational Physics* 307 (2016), S. 614–633 (zitiert auf S. 9, 10, 13).

[HONL17] M. Hirschler, G. Oger, U. Nieken, D. Le Touzé. „Modeling of droplet collisions using Smoothed Particle Hydrodynamics“. In: *International Journal of Multiphase Flow* 95 (2017), S. 175–187 (zitiert auf S. 14).

[LK07] L. Lobovskỳ, J. Křen. „Smoothed particle hydrodynamics modelling of fluids and solids“. In: (2007) (zitiert auf S. 21).

[McC04] S. McConnell. *„Code complete“*. Pearson Education, 2004 (zitiert auf S. 36).

[MG83] J. Monaghan, R. A. Gingold. „Shock simulation by the particle method SPH“. In: *Journal of computational physics* 52.2 (1983), S. 374–389 (zitiert auf S. 29).

[MKI03] J. Monaghan, A. Kos, N. Issa. „Fluid motion generated by impact“. In: *Journal of waterway, port, coastal, and ocean engineering* 129.6 (2003), S. 250–259 (zitiert auf S. 21, 22).

[Mon94] J. J. Monaghan. „Simulating free surface flows with SPH“. In: *Journal of computational physics* 110.2 (1994), S. 399–406 (zitiert auf S. 19).

[Rhe15] O. Rheinbach. *„PETSc in einer Virtualbox“*. 2015 (zitiert auf S. 4).

[Sut14] M. Sutti. *„SPH treatment of boundaries and application to moving objects“*. 2014 (zitiert auf S. 22, 23).

[VI07] D. Violeau, R. Issa. „Numerical modelling of complex turbulent free-surface flows with the SPH method: an overview“. In: *International Journal for Numerical Methods in Fluids* 53.2 (2007), S. 277–304 (zitiert auf S. 21).

[Wen96] H. Wendland. „Konstruktion und Untersuchung radialer Basisfunktionen mit kompaktem Träger“. Diss. 1996 (zitiert auf S. 11).

[XSL09] R. Xu, P. Stansby, D. Laurence. „Accuracy and stability in incompressible SPH (ISPH) based on the projection method and a new approach“. In: *Journal of computational Physics* 228.18 (2009), S. 6703–6725 (zitiert auf S. 12).

Alle URLs wurden zuletzt am 02. 04. 2019 geprüft.

Erklärung

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

Stuttgart, 04.04.2019, 

Ort, Datum, Unterschrift